Axelrod Documentation

Release 4.13.0

Vincent Knight

Apr 25, 2023

Contents

Here is quick overview of the current capabilities of the library:

- Over 230 strategies including many from the literature and exciting original contributions
  - Classic strategies like TiT-For-Tat, WSLS, and variants
  - Zero-Determinant and other Memory-One strategies
  - Many generic strategies that can be used to define an array of popular strategies, including finite state machines, strategies that hunt for patterns in other strategies, and strategies that combine the effects of many others
  - Strategy transformers that augment the abilities of any strategy
- Head-to-Head matches
- Round Robin tournaments with a variety of options, including:
  - noisy environments
  - spatial tournaments
  - probabilistically chosen match lengths
- Population dynamics
  - The Moran process
  - An ecological model
- Multi-processor support (not currently supported on Windows), caching for deterministic interactions, automatically generate figures and statistics

Every strategy is categorized on a number of dimensions, including:

- Deterministic or Stochastic
- How many rounds of history used
- Whether the strategy makes use of the game matrix, the length of the match, etc.

Furthermore the library is extensively tested with 100% coverage, ensuring validity and reproducibility of results!

CHAPTER 1

Quick start

Count the number of available players:

```
>>> import axelrod as axl
>>> len(axl.strategies)
240
```

Create matches between two players:

```
>>> import axelrod as axl
>>> players = (axl.Alternator(), axl.TitForTat())
>>> match = axl.Match(players, 5)
>>> interactions = match.play()
>>> interactions
[(C, C), (D, C), (C, D), (D, C), (C, D)]
```

Build full tournaments between groups of players:

```
>>> import axelrod as axl
>>> players = (axl.Cooperator(), axl.Alternator(), axl.TitForTat())
>>> tournament = axl.Tournament(players)
>>> results = tournament.play()
>>> results.ranked_names
['Alternator', 'Tit For Tat', 'Cooperator']
```

Study the evolutionary process using a Moran process:

```
>>> import axelrod as axl
>>> players = (axl.Cooperator(), axl.Alternator(), axl.TitForTat())
>>> mp = axl.MoranProcess(players)
>>> populations = mp.play()
>>> populations
[Counter({'Alternator': 1, 'Cooperator': 1, 'Tit For Tat': 1}),
 Counter({'Alternator': 1, 'Cooperator': 1, 'Tit For Tat': 1}),
 Counter({'Cooperator': 1, 'Tit For Tat': 2}),
```

(continues on next page)

(continued from previous page)

```
Counter({'Cooperator': 1, 'Tit For Tat': 2}),
Counter({'Tit For Tat': 3})]
```

As well as this, the library has a growing collection of strategies. The *Strategy index* gives a description of them.

For further details there is a library of *Tutorials* available and a *Community* page with information about how to get support and/or make contributions.

CHAPTER 2

Table of Contents

2.1 Tutorials

This section contains a variety of tutorials related to the Axelrod library.

Contents:

2.1.1 New to Game Theory and/or Python

This section contains a variety of tutorials that should help get you started with the Axelrod library.

Contents:

Installation

The library requires Python 3.5.

The simplest way to install the package is to obtain it from the PyPi repository:

```
$ pip install axelrod
```

If you want to have access to the manual Human strategy for interactive play, use the following command to also install *prompt_toolkit*:

```
$ pip install axelrod[Human]
```

You can also build it from source if you would like to:

```
$ git clone https://github.com/Axelrod-Python/Axelrod.git
$ cd Axelrod
$ python setup.py install
```

Creating Matches

You can create your own match between two players using the `Match` class. This is often useful when designing new strategies in order to study how they perform against specific opponents.

For example, to create a 5 turn match between `Cooperator` and `Alternator`:

```
>>> import axelrod as axl
>>> players = (axl.Cooperator(), axl.Alternator())
>>> match = axl.Match(players, 5)
>>> match.play()
[(C, C), (C, D), (C, C), (C, D), (C, C)]
```

By default, a match will not be noisy, but you can introduce noise if you wish. Noise is the probability with which any action dictated by a strategy will be swapped:

```
>>> match = axl.Match(players=players, turns=5, noise=0.2)
>>> match.play()
[(D, C), (C, D), (C, C), (C, D), (D, D)]
```

The result of the match is held as an attribute within the `Match` class. Each time `play()` is called, it will overwrite the content of that attribute:

```
>>> match.result
[(D, C), (C, D), (C, C), (C, D), (D, D)]
>>> match.play()
[(C, C), (C, C), (C, D), (C, C), (C, C)]
>>> match.result
[(C, C), (C, C), (C, D), (C, C), (C, C)]
```

The result of the match can also be viewed as sparklines where cooperation is shown as a solid block and defection as a space. Sparklines are a very concise way to view the result and can be useful for spotting patterns:

```
>>> import axelrod as axl
>>> players = (axl.Cooperator(), axl.Alternator())
>>> match = axl.Match(players, 25)
>>> match.play()
[(C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C,␣
↪C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C),␣
↪(C, D), (C, C), (C, D), (C, C)]
>>> print(match.sparklines())


```

The character for cooperation and a space for defection are default values but you can use any characters you like:

```
>>> import axelrod as axl
>>> players = (axl.Cooperator(), axl.Alternator())
>>> match = axl.Match(players, 25)
>>> match.play()
[(C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C,␣
↪C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C),␣
↪(C, D), (C, C), (C, D), (C, C)]
>>> print(match.sparklines(c_symbol='|', d_symbol='-'))
|||||||||||||||||||||||||
|-|-|-|-|-|-|-|-|-|-|-|-|
```

A *Match* class can also score the individual turns of a match. Just call *match.scores()* after play:

```
>>> import axelrod as axl
>>> players = (axl.Cooperator(), axl.Alternator())
>>> match = axl.Match(players, 25)
>>> match.play()
[(C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C,␣
↪C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C), (C, D), (C, C),␣
↪(C, D), (C, C), (C, D), (C, C)]
>>> match.scores()
[(3, 3), (0, 5), (3, 3), (0, 5), (3, 3), (0, 5), (3, 3), (0, 5), (3, 3), (0, 5), (3,␣
↪3), (0, 5), (3, 3), (0, 5), (3, 3), (0, 5), (3, 3), (0, 5), (3, 3), (0, 5), (3, 3),␣
↪(0, 5), (3, 3), (0, 5), (3, 3)]
```

There are various further methods:

```
>>> match.final_score()
(39, 99)
>>> match.final_score_per_turn()
(1.56, 3.96)
>>> match.winner()
Alternator
>>> match.cooperation()  # The count of cooperations
(25, 13)
>>> match.normalised_cooperation()  # The count of cooperations per turn
(1.0, 0.52)
```

Creating and running a simple tournament

The following lines of code creates a list players playing simple strategies:

```
>>> import axelrod as axl
>>> players = [axl.Cooperator(), axl.Defector(),
...            axl.TitForTat(), axl.Grudger()]
>>> players
[Cooperator, Defector, Tit For Tat, Grudger]
```

We can now create a tournament, play it, save the results and view the rank of each player:

```
>>> tournament = axl.Tournament(players)
>>> results = tournament.play()
>>> results.ranked_names
['Defector', 'Tit For Tat', 'Grudger', 'Cooperator']
```

We can also plot these results:

```
>>> plot = axl.Plot(results)
>>> p = plot.boxplot()
>>> p.show()
```

Note that in this case none of our strategies are stochastic so the boxplot shows that there is no variation. Take a look at the *Visualising results* section to see plots showing a stochastic effect.

Summarising tournament results

As shown in *Creating and running a simple tournament* let us create a tournament:

```
>>> import axelrod as axl
>>> players = [axl.Cooperator(), axl.Defector(),
...            axl.TitForTat(), axl.Grudger()]
>>> tournament = axl.Tournament(players, turns=10, repetitions=3)
>>> results = tournament.play()
```

The results set can return a list of named tuples, ordered by strategy rank that summarises the results of the tournament:

```
>>> summary = results.summarise()
>>> import pprint
>>> pprint.pprint(summary)
[Player(Rank=0, Name='Defector', Median_score=2.6..., Cooperation_rating=0.0, Wins=3.
↪0, Initial_C_rate=0.0, CC_rate=...),
 Player(Rank=1, Name='Tit For Tat', Median_score=2.3..., Cooperation_rating=0...,␣
↪Wins=0.0, Initial_C_rate=1.0, CC_rate=...),
 Player(Rank=2, Name='Grudger', Median_score=2.3..., Cooperation_rating=0..., Wins=0.
↪0, Initial_C_rate=1.0, CC_rate=...),
 Player(Rank=3, Name='Cooperator', Median_score=2.0..., Cooperation_rating=1.0,␣
↪Wins=0.0, Initial_C_rate=1.0, CC_rate=...)]
```

It is also possible to write this data directly to a csv file using the *write_summary* method:

```
>>> results.write_summary('summary.csv')
>>> import csv
>>> with open('summary.csv', 'r') as outfile:
...     csvreader = csv.reader(outfile)
...     for row in csvreader:
...         print(row)
['Rank', 'Name', 'Median_score', 'Cooperation_rating', 'Wins', 'Initial_C_rate', 'CC_
↪rate', 'CD_rate', 'DC_rate', 'DD_rate', 'CC_to_C_rate', 'CD_to_C_rate', 'DC_to_C_
↪rate', 'DD_to_C_rate']
['0', 'Defector', '2.6...', '0.0', '3.0', '0.0', '0.0', '0.0', '0.4...', '0.6...', '0
↪', '0', '0', '0']
['1', 'Tit For Tat', '2.3...', '0.7', '0.0', '1.0', '0.66...', '0.03...', '0.0', '0.3.
↪..', '1.0', '0', '0', '0']
['2', 'Grudger', '2.3...', '0.7', '0.0', '1.0', '0.66...', '0.03...', '0.0', '0.3...',
↪ '1.0', '0', '0', '0']
['3', 'Cooperator', '2.0...', '1.0', '0.0', '1.0', '0.66...', '0.33...', '0.0', '0.0',
↪ '1.0', '1.0', '0', '0']
```

The result set class computes a large number of detailed outcomes read about those in *Access tournament results*.

Visualising results

This tutorial will show you briefly how to visualise some basic results

Visualising the results of the tournament

As shown in *Creating and running a simple tournament*, let us create a tournament, but this time we will include a player that acts randomly:

```
>>> import axelrod as axl
>>> players = [axl.Cooperator(), axl.Defector(),
...            axl.TitForTat(), axl.Grudger()]
>>> players.append(axl.Random())
```

(continues on next page)

(continued from previous page)

```
>>> tournament = axl.Tournament(players)
>>> results = tournament.play()
```

We can view these results (which helps visualise the stochastic effects):

```
>>> plot = axl.Plot(results)
>>> p = plot.boxplot()
>>> p.show()
```

Visualising the distributions of wins

We can view the distributions of wins for each strategy:

```
>>> p = plot.winplot()
>>> p.show()
```

Visualising the payoff matrix

We can also easily view the payoff matrix described in *Access tournament results*, this becomes particularly useful when viewing the outputs of tournaments with a large number of strategies:

```
>>> p = plot.payoff()
>>> p.show()
```

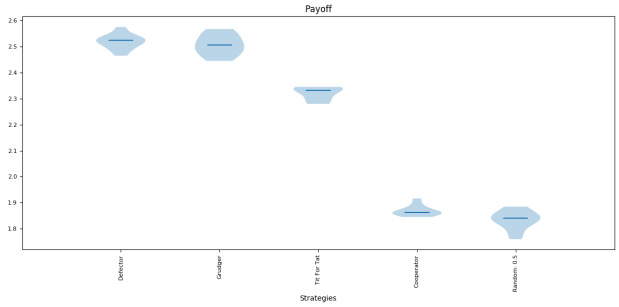
Saving all plots

The `axelrod.Plot` class has a method: `save_all_plots` that will save all the above plots to file.

Passing various objects to plot

The library give access to underlying matplotlib axes objects of each plot, thus the user can easily modify various aspects of a plot:

```
>>> import matplotlib.pyplot as plt
>>> _, ax = plt.subplots()
>>> title = ax.set_title('Payoff')
>>> xlabel = ax.set_xlabel('Strategies')
>>> p = plot.boxplot(ax=ax)
>>> p.show()
```



Moran Process

The strategies in the library can be pitted against one another in the Moran process, a population process simulating natural selection.

The process works as follows. Given an initial population of players, the population is iterated in rounds consisting of:

- matches played between each pair of players, with the cumulative total scores recorded
- a player is chosen to reproduce proportional to the player's score in the round
- a player is chosen at random to be replaced

The process proceeds in rounds until the population consists of a single player type. That type is declared the winner. To run an instance of the process with the library, proceed as follows:

```
>>> import axelrod as axl
>>> players = [axl.Cooperator(), axl.Defector(),
...            axl.TitForTat(), axl.Grudger()]
>>> mp = axl.MoranProcess(players, seed=1)
>>> populations = mp.play()
>>> mp.winning_strategy_name
'Tit For Tat'
```

You can access some attributes of the process, such as the number of rounds:

```
>>> len(mp)
15
```

The sequence of populations:

```
>>> import pprint
>>> pprint.pprint(populations)
[Counter({'Defector': 1, 'Tit For Tat': 1, 'Grudger': 1, 'Cooperator': 1}),
 Counter({'Defector': 1, 'Tit For Tat': 1, 'Grudger': 1, 'Cooperator': 1}),
 Counter({'Cooperator': 2, 'Defector': 1, 'Tit For Tat': 1}),
 Counter({'Defector': 2, 'Cooperator': 2}),
 Counter({'Cooperator': 3, 'Defector': 1}),
 Counter({'Cooperator': 3, 'Defector': 1}),
 Counter({'Defector': 2, 'Cooperator': 2}),
 Counter({'Defector': 3, 'Cooperator': 1}),
 Counter({'Defector': 3, 'Cooperator': 1}),
 Counter({'Defector': 3, 'Cooperator': 1}),
 Counter({'Defector': 3, 'Cooperator': 1}),
 Counter({'Defector': 3, 'Cooperator': 1}),
 Counter({'Defector': 3, 'Cooperator': 1}),
 Counter({'Defector': 3, 'Cooperator': 1}),
```

(continues on next page)

(continued from previous page)

```
 Counter({'Defector': 3, 'Cooperator': 1}),
 Counter({'Defector': 4})]
```

The scores in each round:

```
>>> for row in mp.score_history:
...     print([round(element, 1) for element in row])
[6.0, 7.0, 7.0, 7.0]
[7.0, 3.1, 7.0, 7.0]
[7.0, 3.1, 7.0, 7.0]
[7.0, 3.1, 7.0, 7.0]
[7.0, 3.1, 7.0, 7.0]
[3.0, 3.0, 5.0, 5.0]
[3.0, 3.0, 5.0, 5.0]
[3.1, 7.0, 7.0, 7.0]
[3.1, 7.0, 7.0, 7.0]
[9.0, 9.0, 9.0, 9.0]
[9.0, 9.0, 9.0, 9.0]
[9.0, 9.0, 9.0, 9.0]
[9.0, 9.0, 9.0, 9.0]
[9.0, 9.0, 9.0, 9.0]
```

We can plot the results of a Moran process with *mp.populations_plot()*. Let's use a larger population to get a bit more data:

```
>>> import random
>>> import matplotlib.pyplot as plt
>>> players = [axl.Defector(), axl.Defector(), axl.Defector(),
...            axl.Cooperator(), axl.Cooperator(), axl.Cooperator(),
...            axl.TitForTat(), axl.TitForTat(), axl.TitForTat(),
...            axl.Random()]
>>> mp = axl.MoranProcess(players=players, turns=200, seed=2)
>>> populations = mp.play()
>>> mp.winning_strategy_name
'Tit For Tat'
>>> ax = mp.populations_plot()
>>> plt.show()
```

Moran Process with Mutation

The `MoranProcess` class also accepts an argument for a mutation rate. Nonzero mutation changes the Markov process so that it no longer has absorbing states, and will iterate forever. To prevent this, iterate with a loop (or function like `takewhile` from `itertools`):

```
>>> import axelrod as axl
>>> players = [axl.Cooperator(), axl.Defector(),
...            axl.TitForTat(), axl.Grudger()]
>>> mp = axl.MoranProcess(players, mutation_rate=0.1, seed=10)
>>> for _ in mp:
...     if len(mp.population_distribution()) == 1:
...         break
>>> mp.population_distribution()
Counter({'Defector': 4})
```

It is possible to pass a fitness function that scales the utility values. A common one used in the literature, [Ohtsuki2006], is $f(s) = 1 - w + ws$ where w denotes the intensity of selection:

```
>>> players = (axl.Cooperator(), axl.Defector(), axl.Defector(), axl.Defector())
>>> w = 0.95
>>> fitness_transformation = lambda score: 1 - w + w * score
>>> mp = axl.MoranProcess(players, turns=10, fitness_transformation=fitness_
↪transformation, seed=3)
>>> populations = mp.play()
>>> mp.winning_strategy_name
'Defector'
```

Other types of implemented Moran processes:

- *Create Moran Processes On Graphs*
- *Create Approximate Moran Process*

Human Interaction

It is possible to play interactively using the Human strategy:

```
>>> import axelrod as axl
>>> me = axl.Human(name='me')
>>> players = [axl.TitForTat(), me]
>>> match = axl.Match(players, turns=3)
>>> match.play()
```

You will be prompted for the action to play at each turn:

```
Starting new match
Turn 1 action [C or D] for me: C

Turn 1: me played C, opponent played C
Turn 2 action [C or D] for me: D

Turn 2: me played D, opponent played C
Turn 3 action [C or D] for me: C
[(C, C), (C, D), (D, C)]
```

after this, the `match` object can be manipulated as described in *Creating Matches*

2.1.2 Running Axelrod's First Tournament

This tutorial will bring together topics from the previous tutorials to reproduce Axelrod's original tournament from [Axelrod1980].

Selecting our players

We will use the players from Axelrod's first tournament which are contained in the *axelrod.axelrod_first_strategies* list:

```
>>> import axelrod as axl
>>> first_tournament_participants_ordered_by_reported_rank = [s() for s in axl.
↪axelrod_first_strategies]
```

(continues on next page)

(continued from previous page)

```
>>> number_of_strategies = len(first_tournament_participants_ordered_by_reported_rank)
>>> for player in first_tournament_participants_ordered_by_reported_rank:
...     print(player)
Tit For Tat
First by Tideman and Chieruzzi: (D, D)
First by Nydegger
First by Grofman
First by Shubik
First by Stein and Rapoport: 0.05: (D, D)
Grudger
First by Davis: 10
First by Graaskamp: 0.05
First by Downing
First by Feld: 1.0, 0.5, 200
First by Joss: 0.9
First by Tullock
First by Anonymous
Random: 0.5
```

Creating the tournament

Now we create and run the tournament, we will set a seed to ensure reproducibility and 5 repetitions to smooth the random effects. We use 5 repetitions as this is what was done in [Axelrod1980]:

```
>>> tournament = axl.Tournament(
...      players=first_tournament_participants_ordered_by_reported_rank,
...      turns=200,
...      repetitions=5,
...      seed=1,
... )
>>> results = tournament.play()
```

Viewing the ranks of the participants

The results object contains the ranked names:

```
>>> for name in results.ranked_names:
...     print(name)
First by Stein and Rapoport: 0.05: (D, D)
First by Grofman
First by Shubik
Tit For Tat
First by Nydegger
First by Tideman and Chieruzzi: (D, D)
Grudger
First by Davis: 10
First by Graaskamp: 0.05
First by Downing
First by Feld: 1.0, 0.5, 200
First by Tullock
First by Joss: 0.9
First by Anonymous
Random: 0.5
```

We see that *TitForTat* does not win. In fact *TitForTat* typically does not win this tournament, possibly because our implementations differ from the original strategies as their code is not available.

We can plot the reported rank (from [Axelrod1980]) versus the reproduced one:

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(15, 6))
>>> plt.plot((0, 15), (0, 15), color="grey", linestyle="--")
>>> for original_rank, strategy in enumerate(first_tournament_participants_ordered_by_
↪reported_rank):
...     rank = results.ranked_names.index(str(strategy))
...     if rank == original_rank:
...         symbol = "+"
...         plt.plot((rank, rank), (rank, 0), color="grey")
...     else:
...         symbol = "o"
...     plt.scatter([rank], [original_rank], marker=symbol, color="black", s=50)
>>> plt.xticks(
...     range(number_of_strategies),
...     results.ranked_names,
...     rotation=90
... )
>>> plt.ylabel("Reported rank")
>>> plt.xlabel("Reproduced rank");
>>> plt.show()
```

Visualising the scores

We see that the first 6 strategies do not match the ranks of the original paper, we can take a look the variation in the scores:

```
>>> plot = axl.Plot(results)
>>> p = plot.boxplot()
>>> p.show()
```

The first 6 strategies have similar scores which could indicate that the original work by Axelrod was not run with sufficient repetitions. Another explanation is that all the strategies are implemented from the descriptions given in [Axelrod1980] and there is no source code to base this on. This leads to some strategies being ambiguous. These are all clearly explained in the strategy docstrings. For example:

```
>>> print(axl.FirstByAnonymous.__doc__)

    Submitted to Axelrod's first tournament by a graduate student whose name was
    withheld.

    The description written in [Axelrod1980]_ is:

    > "This rule has a probability of cooperating, P, which is initially 30% and
    > is updated every 10 moves. P is adjusted if the other player seems random,
    > very cooperative, or very uncooperative. P is also adjusted after move 130
    > if the rule has a lower score than the other player. Unfortunately, the
    > complex process of adjustment frequently left the probability of cooperation
    > in the 30% to 70% range, and therefore the rule appeared random to many
```

(continues on next page)

(continued from previous page)

```
    > other players."

    Given the lack of detail this strategy is implemented based on the final
    sentence of the description which is to have a cooperation probability that
    is uniformly random in the 30 to 70% range.

    Names:

    - (Name withheld): [Axelrod1980]_
```

Other outcomes

If we run the tournament with other seeds, the results are different. For example, with *1408* Tit For Tat wins:

```
>>> tournament = axl.Tournament(
...       players=first_tournament_participants_ordered_by_reported_rank,
...       turns=200,
...       repetitions=5,
...       seed=1408,
... )
>>> results = tournament.play()
>>> for name in results.ranked_names:
...     print(name)
Tit For Tat
First by Stein and Rapoport: 0.05: (D, D)
First by Grofman
First by Shubik
First by Tideman and Chieruzzi: (D, D)
First by Nydegger
Grudger
First by Davis: 10
First by Graaskamp: 0.05
First by Downing
First by Feld: 1.0, 0.5, 200
First by Tullock
First by Joss: 0.9
First by Anonymous
Random: 0.5
```

With *136* the strategy submitted by Grofman wins:

```
>>> tournament = axl.Tournament(
...       players=first_tournament_participants_ordered_by_reported_rank,
...       turns=200,
...       repetitions=5,
...       seed=136
... )
>>> results = tournament.play()
>>> for name in results.ranked_names:
...     print(name)
First by Grofman
First by Stein and Rapoport: 0.05: (D, D)
Tit For Tat
First by Shubik
First by Tideman and Chieruzzi: (D, D)
```

(continues on next page)

(continued from previous page)

```
First by Nydegger
Grudger
First by Davis: 10
First by Downing
First by Graaskamp: 0.05
First by Feld: 1.0, 0.5, 200
First by Joss: 0.9
First by Tullock
Random: 0.5
First by Anonymous
```

2.1.3 Create Heterogeneous Moran Processes

Axelrod Matches are homogeneous by nature but can be extended to utilize additional attributes of heterogeneous players. This tutorial indicates how the Axelrod `Match` class can be manipulated in order to play heterogeneous tournaments and Moran processes using mass as a score modifier similarly to the work of [Krapohl2020].

The following lines of code creates a list of players from the available demo strategies along with an ascending list of masses we will use for the players. This is equivalent in principle to the country masses discussed in [Krapohl2020]:

```
>>> import axelrod as axl
>>> players = [player() for player in axl.demo_strategies]
>>> masses = [i for i in range(len(players))]
>>> players
[Cooperator, Defector, Tit For Tat, Grudger, Random: 0.5]
```

Using the `setattr()` function, additional attributes can be passed to players to enable access during matches and tournaments without manual modification of individual strategies:

```
>>> def set_player_mass(players, masses):
...     """Add mass attribute to player strategy classes to be accessable via self.
↪mass"""
...     for player, mass in zip(players, masses):
...         setattr(player, "mass", mass)
...
>>> set_player_mass(players, masses)
```

The `Match` class can be partially altered to enable different behaviour (see *Use custom matches*). Here we extend `axl.Match` and overwrite its `final_score_per_turn()` function to utilize the player mass attribute as a multiplier for the final score:

```
>>> class MassBaseMatch(axl.Match):
...     """Axelrod Match object with a modified final score function to enable mass␣
↪to influence the final score as a multiplier"""
...     def final_score_per_turn(self):
...         base_scores = axl.Match.final_score_per_turn(self)
...         return [player.mass * score for player, score in zip(self.players, base_
↪scores)]
```

In [Krapohl2020] a non standard Moran process is used where the mass of individuals is not reproduced so we will use inheritance to create a new Moran process that keeps the mass of the individuals constant:

```
>>> class MassBasedMoranProcess(axl.MoranProcess):
...     """Axelrod MoranProcess class """
```

(continues on next page)

(continued from previous page)

```
...     def __next__(self):
...         set_player_mass(self.players, masses)
...         super().__next__()
...         return self

>>> mp = MassBasedMoranProcess(players, match_class=MassBaseMatch, seed=0)
>>> populations = mp.play()
>>> print(mp.winning_strategy_name)
Random: 0.5
```

Note that the snippets here only influence the final score of matches. The behavior of matches, and Moran processes can be more heavily influenced by partially overwriting other `match` functions or `birth` and `death` functions within `MoranProcess`.

2.1.4 Implement new games

Currently, the default `Strategy`, `Action` and `Game` implementations in Axelrod are centred around the Iterated Prisoners' Dilemma. The stage game can be changed as shown in *Use different stage games*.

However, just changing the stage game may not be sufficient. Take, for example, the game rock-paper-scissors:

```
>>> import axelrod as axl
>>> import numpy as np
>>> A = np.array([[0, -1, 1], [1, 0, -1], [-1, 1, 0]])
>>> rock_paper_scissors = axl.AsymmetricGame(A, -A)
>>> rock_paper_scissors
Axelrod game with matrices: (array([[ 0, -1,  1],
                                    [ 1,  0, -1],
                                    [-1,  1,  0]]),
                             array([[ 0,  1, -1],
                                    [-1,  0,  1],
                                    [ 1, -1,  0]]))
```

If we tried to run a rock-paper-scissors match with the `Tit-For-Tat` strategy, it wouldn't work properly. `Tit-For-Tat` only knows of two actions (cooperate and defect, corresponding to rows 1 and 2 respectively). If we tried to use it on rock-paper-scissors, it would interpret the game in the following way:

1. On the first turn, choose rock (option 1, cooperate)
2. If the opponent's last move is the Python object `axl.Action.D` (which it may never be unless the opponent also thinks it's playing IPD!), then choose paper (option 2, defection)

and so as we see, `Tit-For-Tat` would simply play Rock every turn, unless it were playing against another Prisoners' Dilemma strategy (then it plays rock unless the opponent last played paper, in which case it plays paper). In particular, it would never play scissors - it does not know that Scissors is something it can even do. This is not a bug, or an issue with the strategy itself; simply that `Tit-For-Tat` *thinks* it is playing the Iterated Prisoners' Dilemma and its `Action` set, regardless of what the actual game is.

Thus, if we wanted to implement new games we should also implement a new Action set, and some new strategies.

The Actions are relatively simple; they're an Enum class, with each action corresponding to a row/column (recall that Python starts counting from 0, rather than 1). We can also implement some methods that we think might be useful for viewing our actions and making strategies. (The Prisoners' Dilemma `Action` class, for example, has `flip`, which flips a `C` to a `D` and vice versa!)

A simple rock-paper-scissors action class would look like so:

```
>>> from enum import Enum
>>> class RPSAction(Enum):
...     """Actions for Rock-Paper-Scissors."""
...     R = 0  # rock
...     P = 1  # paper
...     S = 2  # scissors
...
...     def __repr__(self):
...         return self.name
...
...     def __str__(self):
...         return self.name
...
...     def rotate(self):
...         """
...         Cycles one step through the actions.
...         Maps R->P, P->S, S->R
...         """
...         rotations = {
...             RPSAction.R: RPSAction.P,
...             RPSAction.P: RPSAction.S,
...             RPSAction.S: RPSAction.R
...         }
...
...         return rotations[self]
```

We can then implement some strategies. Below we have the implementation of an Axelrod strategy into Python. These follow the same format;

- A subclass of the `Player` class, with three parts:
  - A `name` and a `classifier` dictionary. This is used for indexing strategies.
  - (Optionally) an `__init__` method, which allows the setting of initialisation variables (like probabilities of doing certain actions, or starting moves)
  - A `strategy` method, which takes the parameters `self` and `opponent`, representing both players in the match, and provides the algorithm for determining the player's next move.

If we want, we can also initialise some shorthand for the actions to avoid having to evoke their full names:

```
>>> R = RPSAction.R
>>> P = RPSAction.P
>>> S = RPSAction.S
```

Here are a couple of examples. One is a strategy which copies the opponent's previous move, and the other simply cycles through the moves. Both have an initialisation parameter for which move they start with:

```
>>> from axelrod.player import Player
>>> class Copycat(Player):
...     """
...     Starts with a chosen move,
...     and then copies their opponent's previous move.
...
...     Parameters
...     ----------
...     starting_move: RPSAction, default S
...         What move to play on the first round.
...     """
```

(continues on next page)

(continued from previous page)

```
...     name = "Copycat"
...     classifier = {
...         "memory_depth": 1,
...         "stochastic": False,
...         "long_run_time": False,
...         "inspects_source": False,
...         "manipulates_source": False,
...         "manipulates_state": False,
...     }
...
...     def __init__(self, starting_move=S):
...         self.starting_move = starting_move
...         super().__init__()
...
...     def strategy(self, opponent: Player) -> RPSAction:
...         """Actual strategy definition that determines player's action."""
...         if not self.history:
...             return self.starting_move
...         return opponent.history[-1]

>>> class Rotator(Player):
...     """
...     Cycles through the moves from a chosen starting move.
...
...     Parameters
...     ----------
...     starting_move: RPSAction, default S
...         What move to play on the first round.
...     """
...     name = "Rotator"
...     classifier = {
...         "memory_depth": 1,
...         "stochastic": False,
...         "long_run_time": False,
...         "inspects_source": False,
...         "manipulates_source": False,
...         "manipulates_state": False,
...     }
...
...     def __init__(self, starting_move=S):
...         self.starting_move = starting_move
...         super().__init__()
...
...     def strategy(self, opponent: Player) -> RPSAction:
...         """Actual strategy definition that determines player's action."""
...         if not self.history:
...             return self.starting_move
...         return self.history[-1].rotate()
```

We are now all set to run some matches and tournaments in our new game! Let's start with a match between our two new players:

```
>>> match = axl.Match(players=(Copycat(starting_move=P), Rotator()),
...                   turns=5,
...                   game=rock_paper_scissors)
>>> match.play()
[(P, S), (S, R), (R, P), (P, S), (S, R)]
```

and as with the Prisoners' Dilemma, we can run a tournament in the same way. Just make sure you specify the game when creating the tournament!:

```
>>> tournament = axl.Tournament(players, game=rock_paper_scissors)
>>> tournament.play()
```

where `players` is set to a list of Rock-Paper-Scissors strategies; hopefully more than two, else it isn't a very interesting tournament!

2.2 How to

This section contains short descriptions of how to accomplish specific tasks with the Axelrod library.

2.2.1 Include noise

A common variation on iterated prisoner's dilemma tournaments is to add stochasticity in the choice of actions, simply called noise. This noise is introduced by flipping plays between C and D with some probability that is applied to all plays after they are delivered by the player [Bendor1993].

The presence of this persistent background noise causes some strategies to behave substantially differently. For example, `TitForTat` can fall into defection loops with itself when there is noise. While `TitForTat` would usually cooperate well with itself:

```
C C C C C ...
C C C C C ...
```

Noise can cause a C to flip to a D (or vice versa), disrupting the cooperative chain:

```
C C C D C D C D D D ...
C C C C D C D D D D ...
```

To create a noisy tournament you simply need to add the *noise* argument:

```
>>> import axelrod as axl
>>> players = [axl.Cooperator(), axl.Defector(),
...            axl.TitForTat(), axl.Grudger()]
>>> noise = 0.1
>>> tournament = axl.Tournament(players, noise=noise)
>>> results = tournament.play()
>>> plot = axl.Plot(results)
>>> p = plot.boxplot()
>>> p.show()
```

Here is how the distribution of wins now looks:

```
>>> p = plot.winplot()
>>> p.show()
```

2.2.2 Include probabilistic endings

It is possible to create a tournament where the length of each Match is not constant for all encounters: after each turn the Match ends with a given probability, [Axelrod1980b]:

```
>>> import axelrod as axl
>>> players = [axl.Cooperator(), axl.Defector(),
...            axl.TitForTat(), axl.Grudger()]
>>> tournament = axl.Tournament(players, prob_end=0.5)
```

We can view the results in a similar way as described in *Access tournament results*:

```
>>> results = tournament.play()
>>> m = results.payoff_matrix
>>> for row in m:
...     print([round(ele, 1) for ele in row]) # Rounding output

[3.0, 0.0, 3.0, 3.0]
[5.0, 1.0, 3.7, 3.6]
[3.0, 0.3, 3.0, 3.0]
[3.0, 0.4, 3.0, 3.0]
```

We see that `Cooperator` always scores 0 against `Defector` but other scores seem variable as they are effected by the length of each match.

We can (as before) obtain the ranks for our players:

```
>>> results.ranked_names
['Defector', 'Tit For Tat', 'Grudger', 'Cooperator']
```

We can plot the results:

```
>>> plot = axl.Plot(results)
>>> p = plot.boxplot()
>>> p.show()
```

We can also view the length of the matches played by each player. The plot shows that the length of each match (for each player) is not the same. The median length is 4 which is the expected value with the probability of a match ending being `0.5`.

```
>>> p = plot.lengthplot()
>>> p.show()
```

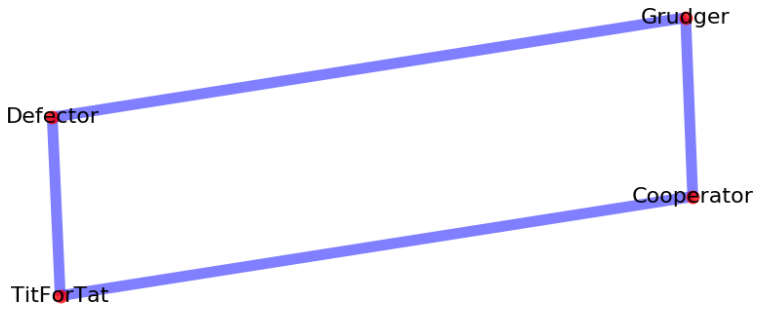
2.2.3 Create spatial tournaments

A spatial tournament is defined on a graph where the nodes correspond to players and edges define whether or not a given player pair will have a match.

The initial work on spatial tournaments was done by Nowak and May in a 1992 paper: [Nowak1992].

Additionally, Szabó and Fáth in their 2007 paper [Szabo2007] consider a variety of graphs, such as lattices, small world, scale-free graphs and evolving networks.

Let's create a tournament where `Cooperator` and `Defector` do not play each other and neither do `TitForTat` and `Grudger` :



Note that the edges have to be given as a list of tuples of player indices:

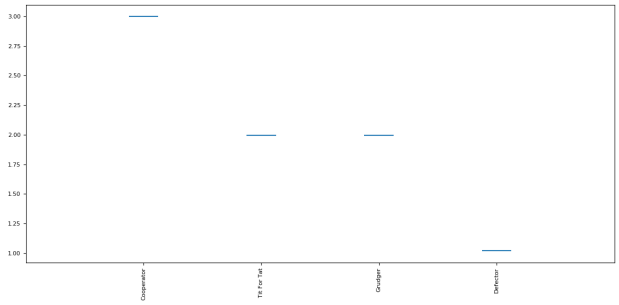
```
>>> import axelrod as axl
>>> players = [axl.Cooperator(), axl.Defector(),
...            axl.TitForTat(), axl.Grudger()]
>>> edges = [(0, 2), (0, 3), (1, 2), (1, 3)]
```

To create a spatial tournament you pass the `edges` to the `Tournament` class:

```
>>> spatial_tournament = axl.Tournament(players, edges=edges)
>>> results = spatial_tournament.play()
```

We can plot the results:

```
>>> plot = axl.Plot(results)
>>> p = plot.boxplot()
>>> p.show()
```



We can, like any other tournament, obtain the ranks for our players:

```
>>> results.ranked_names
['Cooperator', 'Tit For Tat', 'Grudger', 'Defector']
```

Let's run a small tournament of 2 `turns` and 2 `repetitions` and obtain the interactions:

```
>>> spatial_tournament = axl.Tournament(players ,turns=2, repetitions=2, edges=edges)
>>> results = spatial_tournament.play()
```

(continues on next page)

(continued from previous page)

```
>>> results.payoffs
[[[], [], [3.0, 3.0], [3.0, 3.0]], [[], [], [3.0, 3.0], [3.0, 3.0]], [[3.0, 3.0], [0.
↪5, 0.5], [], []], [[3.0, 3.0], [0.5, 0.5], [], []]]
```

As anticipated not all players interact with each other.

It is also possible to create a probabilistic ending spatial tournament:

```
>>> prob_end_spatial_tournament = axl.Tournament(players, edges=edges, prob_end=.1,␣
↪repetitions=1, seed=10)
>>> prob_end_results = prob_end_spatial_tournament.play()
```

We see that the match lengths are no longer all equal:

```
>>> prob_end_results.match_lengths
[[[0, 0, 20.0, 1.0], [0, 0, 46.0, 13.0], [20.0, 46.0, 0, 0], [1.0, 13.0, 0, 0]]]
```

2.2.4 Create Moran Processes On Graphs

The library also provides a graph-based Moran process [Shakarian2013] with `MoranProcess`. To use this feature you must supply at least one `Axelrod.graph.Graph` object, which can be initialized with just a list of edges:

```
edges = [(source_1, target1), (source2, target2), ...]
```

The nodes can be any hashable object (integers, strings, etc.). For example:

```
>>> import axelrod as axl
>>> from axelrod.graph import Graph
>>> edges = [(0, 1), (1, 2), (2, 3), (3, 1)]
>>> graph = Graph(edges)
```

Graphs are undirected by default but you can pass `directed=True` to create a directed graph. Various intermediates such as the list of neighbors are cached for efficiency by the graph object.

A Moran process can be invoked with one or two graphs. The first graph, the *interaction graph*, dictates how players are matched up in the scoring phase. Each player plays a match with each neighbor. The second graph dictates how players replace another during reproduction. When an individual is selected to reproduce, it replaces one of its neighbors in the *reproduction graph*. If only one graph is supplied to the process, the two graphs are assumed to be the same.

To create a graph-based Moran process, use a graph as follows:

```
>>> from axelrod.graph import Graph
>>> edges = [(0, 1), (1, 2), (2, 3), (3, 1)]
>>> graph = Graph(edges)
>>> players = [axl.Cooperator(), axl.Cooperator(), axl.Cooperator(), axl.Defector()]
>>> mp = axl.MoranProcess(players, interaction_graph=graph, seed=40)
>>> results = mp.play()
>>> mp.population_distribution()
Counter({'Defector': 4})
```

You can supply the `reproduction_graph` as a keyword argument. The standard Moran process is equivalent to using a complete graph with no loops for the `interaction_graph` and with loops for the `reproduction_graph`.

2.2.5 Create Approximate Moran Process

Due to the high computational cost of a single Moran process, an approximate Moran process is implemented that can make use of cached outcomes of games. The following code snippet will generate a Moran process in which the outcomes of the matches played by a `Random: 0.5` are sampled from one possible outcome against each opponent (`Defector` and `Random: 0.5`). First the cache is built by passing counter objects of outcomes:

```
>>> import axelrod as axl
>>> from collections import Counter
>>> cached_outcomes = {}
>>> cached_outcomes[("Random: 0.5", "Defector")] = axl.Pdf(Counter([(1, 1)]))
>>> cached_outcomes[("Random: 0.5", "Random: 0.5")] = axl.Pdf(Counter([(3, 3)]))
>>> cached_outcomes[("Defector", "Defector")] = axl.Pdf(Counter([(1, 1)]))
```

Now let us create an Approximate Moran Process:

```
>>> players = [axl.Defector(), axl.Random(), axl.Random()]
>>> amp = axl.ApproximateMoranProcess(players, cached_outcomes, seed=5)
>>> results = amp.play()
>>> amp.population_distribution()
Counter({'Random: 0.5': 3})
```

Note that by nature of being an approximation, it's possible that the results of an *ApproximateMoranProcess* may not always match the results of a standard *MoranProcess*, even for the same random seed. We see that, for this random seed, the `Random: 0.5` won this Moran process. This is not what happens in a standard Moran process where the `Random: 0.5` player will not win:

```
>>> mp = axl.MoranProcess(players, seed=5)
>>> results = mp.play()
>>> mp.population_distribution()
Counter({'Defector': 3})
```

2.2.6 Calculate Morality Metrics

Tyler Singer-Clark's June 2014 paper, "Morality Metrics On Iterated Prisoner's Dilemma Players" [Singer-Clark2014]), describes several interesting metrics which may be used to analyse IPD tournaments all of which are available within the ResultSet class. (Tyler's paper is available here: http://www.scottaaronson.com/morality.pdf).

Each metric depends upon the cooperation rate of the players, defined by Tyler Singer-Clark as:

$$CR(b) = \frac{C(b)}{TT}$$

where C(b) is the total number of turns where a player chose to cooperate and TT is the total number of turns played.

A matrix of cooperation rates is available within a tournament's ResultSet:

```
>>> import axelrod as axl
>>> players = [axl.Cooperator(), axl.Defector(),
...            axl.TitForTat(), axl.Grudger()]
>>> tournament = axl.Tournament(players)
>>> results = tournament.play()
>>> [[round(float(ele), 3) for ele in row] for row in results.normalised_cooperation]
[[1.0, 1.0, 1.0, 1.0], [0.0, 0.0, 0.0, 0.0], [1.0, 0.005, 1.0, 1.0], [1.0, 0.005, 1.0,
↪ 1.0]]
```

There is also a 'good partner' matrix showing how often a player cooperated at least as much as its opponent:

```
>>> results.good_partner_matrix
[[0, 10, 10, 10], [0, 0, 0, 0], [10, 10, 0, 10], [10, 10, 10, 0]]
```

Each of the metrics described in Tyler's paper is available as follows (here they are rounded to 2 digits):

```
>>> [round(ele, 2) for ele in results.cooperating_rating]
[1.0, 0.0, 0.67..., 0.67...]
>>> [round(ele, 2) for ele in results.good_partner_rating]
[1.0, 0.0, 1.0, 1.0]
>>> [round(ele, 2) for ele in results.eigenjesus_rating]
[0.58, 0.0, 0.58, 0.58]
>>> [round(ele, 2) for ele in results.eigenmoses_rating]
[0.37, -0.37, 0.6, 0.6]
```

2.2.7 Run Axelrod's Ecological Variant

In Axelrod's original work an ecological approach based on the payoff matrix of the tournament was used to study the evolutionary stability of each strategy. Whilst this bears some comparison to the *Moran Process*, the latter is much more widely used in the literature.

To study the evolutionary stability of each strategy it is possible to create an ecosystem based on the payoff matrix of a tournament:

```
>>> import axelrod as axl
>>> players = [axl.Cooperator(), axl.Defector(),
...            axl.TitForTat(), axl.Grudger(),
...            axl.Random()]
>>> tournament = axl.Tournament(players)
>>> results = tournament.play()
>>> eco = axl.Ecosystem(results)
>>> eco.reproduce(100) # Evolve the population over 100 time steps
```

Here is how we obtain a nice stackplot of the system evolving over time:

```
>>> plot = axl.Plot(results)
>>> p = plot.stackplot(eco)
>>> p.show()
```

2.2.8 Fingerprint

Ashlock Fingerprints

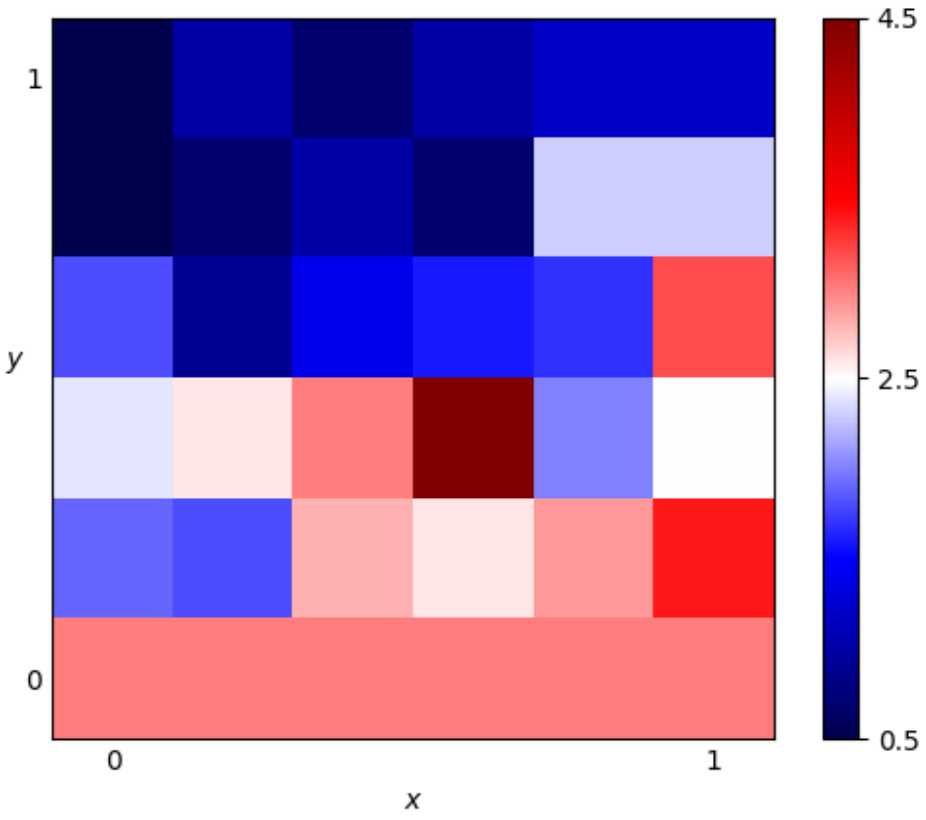
In [Ashlock2008], [Ashlock2009] a methodology for obtaining visual representation of a strategy's behaviour is described. The basic method is to play the strategy against a probe strategy with varying noise parameters. These noise parameters are implemented through the `JossAnnTransformer`. The Joss-Ann of a strategy is a new strategy which has a probability `x` of cooperating, a probability `y` of defecting, and otherwise uses the response appropriate to the original strategy. We can then plot the expected score of the strategy against `x` and `y` and obtain a heat plot over the unit square. When `x + y >= 1` the `JossAnn` is created with parameters `(1-y, 1-x)` and plays against the Dual of the probe instead. A full definition and explanation is given in [Ashlock2008], [Ashlock2009].

Here is how to create a fingerprint of `WinStayLoseShift` using `TitForTat` as a probe:

```
>>> import axelrod as axl
>>> strategy = axl.WinStayLoseShift
>>> probe = axl.TitForTat
>>> af = axl.AshlockFingerprint(strategy, probe)
>>> data = af.fingerprint(turns=10, repetitions=2, step=0.2, seed=1)
>>> data
{...
>>> data[(0, 0)]
3.0
```

The `fingerprint` method returns a dictionary mapping coordinates of the form `(x, y)` to the mean score for the corresponding interactions. We can then plot the above to get:

```
>>> p = af.plot()
>>> p.show()
```

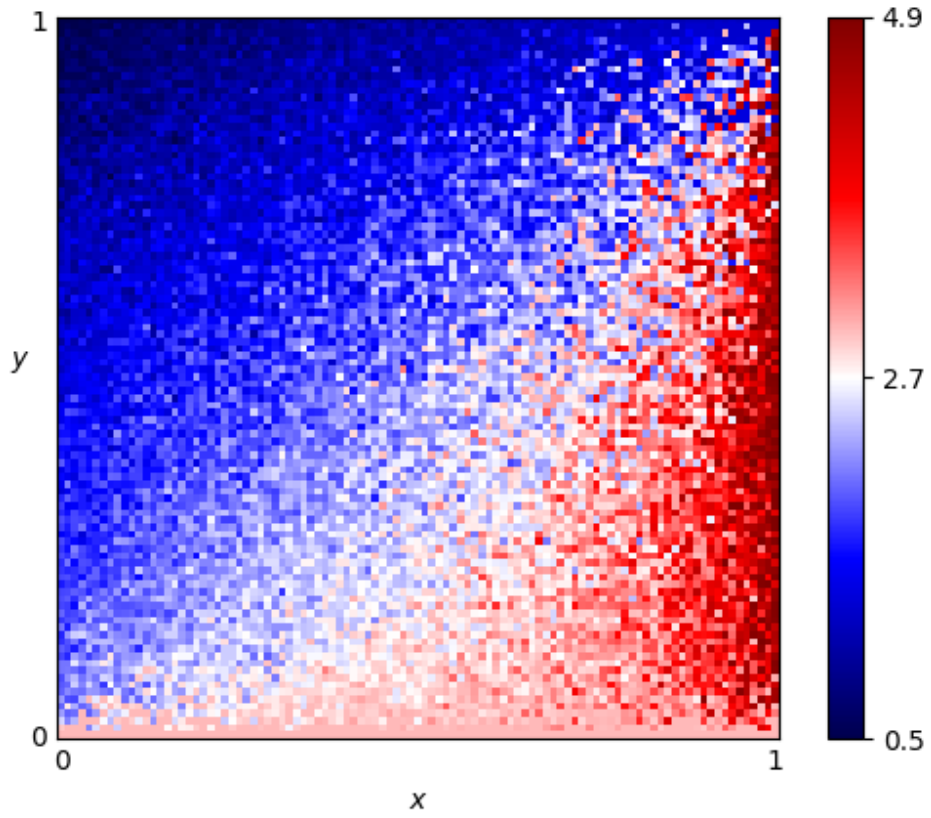


In reality we would need much more detail to make this plot useful.

Running the above with the following parameters:

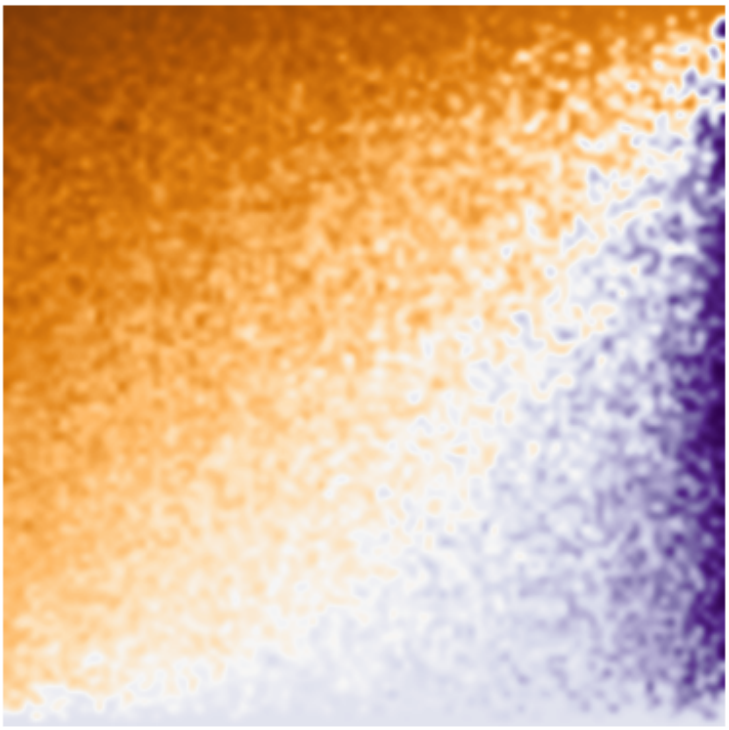
```
>>> af.fingerprint(turns=50, repetitions=2, step=0.01)
```

We get the plot:



We are also able to specify a matplotlib colour map, interpolation and can remove the colorbar and axis labels:

```
>>> p = af.plot(cmap='PuOr', interpolation='bicubic', colorbar=False, labels=False)
>>> p.show()
```



Note that it is also possible to pass a player instance to be fingerprinted and/or as a probe. This allows for the fingerprinting of parametrized strategies:

```
>>> player = axl.Random(p=.1)
>>> probe = axl.GTFT(p=.9)
>>> af = axl.AshlockFingerprint(player, probe)
>>> data = af.fingerprint(turns=10, repetitions=2, step=0.2, seed=2)
>>> data
{...
>>> data[(0, 0)]
3.75
```

Transitive Fingerprint

Another implemented fingerprint is the transitive fingerprint. The transitive fingerprint represents the cooperation rate of a strategy against a set of opponents over a number of turns.

By default the set of opponents consists of `50` Random players that cooperate with increasing probability. This is how to obtain the transitive fingerprint for `TitForTat`:

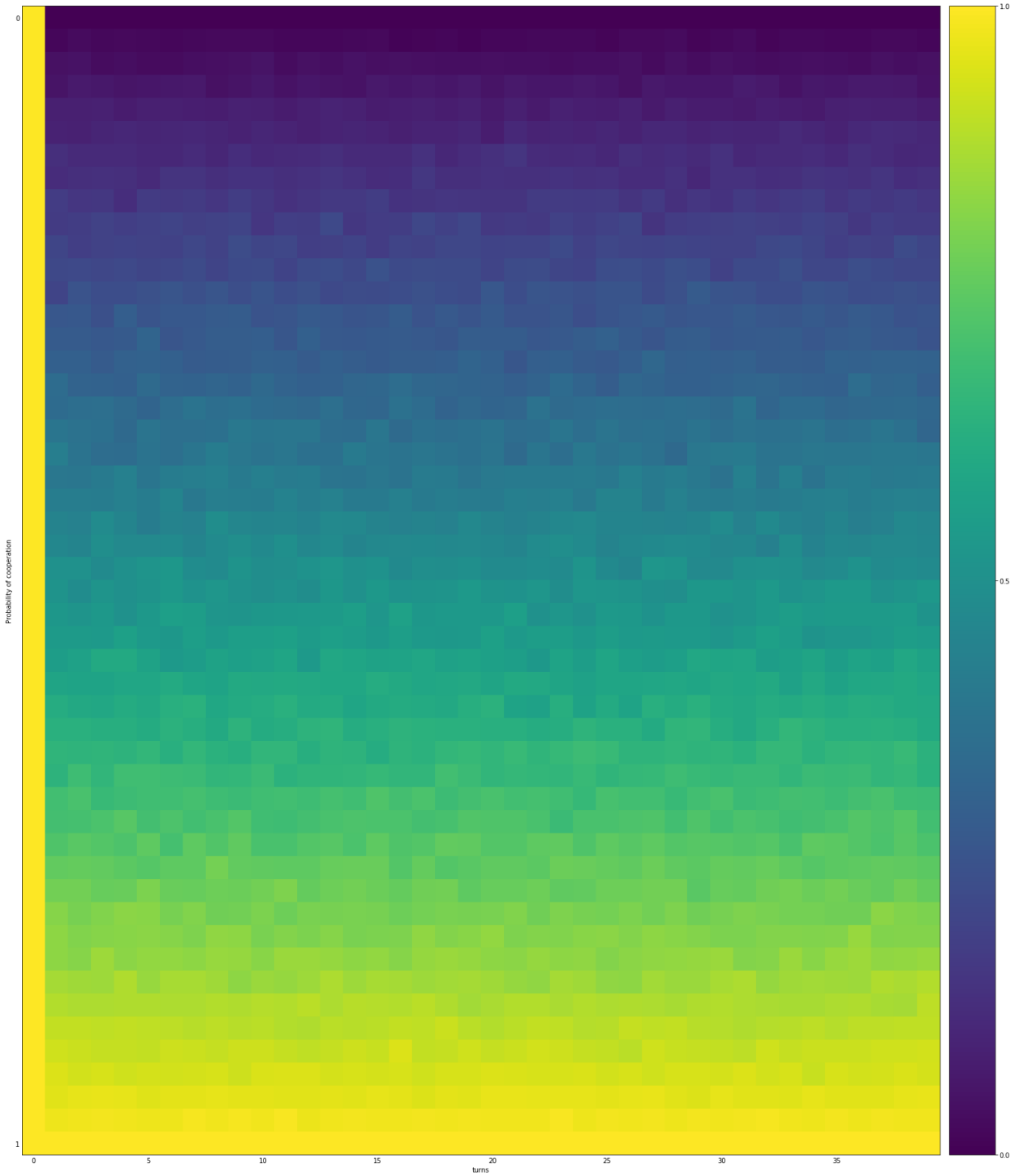
```
>>> player = axl.TitForTat()
>>> tf = axl.TransitiveFingerprint(player)
>>> data = tf.fingerprint(turns=40, seed=3)
```

The data produced is a `numpy` array showing the cooperation rate against a given opponent (row) in a given turn (column):

```
>>> data.shape
(50, 40)
```

It is also possible to visualise the fingerprint:

```
>>> p = tf.plot()
>>> p.show()
```

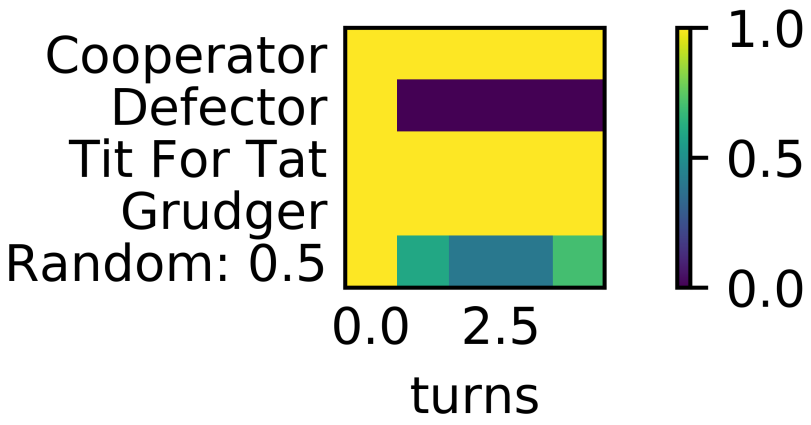


It is also possible to fingerprint against a given set of opponents:

```
>>> opponents = [s() for s in axl.demo_strategies]
>>> tf = axl.TransitiveFingerprint(player, opponents=opponents)
>>> data = tf.fingerprint(turns=5, repetitions=10, seed=4)
```

The name of the opponents can be displayed in the plot:

```
>>> p = tf.plot(display_names=True)
>>> p.show()
```



2.2.9 Evolve Players

Several strategies in the library derive from `EvolvablePlayer` which specifies methods allowing evolutionary or particle swarm algorithms to be used with these strategies. The Axelrod Dojo library [Axelrod1980] contains implementations of both algorithms for use with the Axelrod library. Examples include FSMPlayers, ANN (neural networks), and LookerUp and Gambler (lookup tables).

New `EvolvablePlayer` subclasses can be added to the library. Any strategy that can define `mutation` and `crossover` methods can be used with the evolutionary algorithm and the atomic mutation version of the Moran process. To use the particle swarm algorithms, methods to serialize the strategy to and from a vector of floats must be defined.

Moran Process: Atomic Mutation for Evolvable Players

Additionally, the Moran process implementation supports a second style of mutation suitable for evolving new strategies utilizing the `EvolvablePlayer` class via its `mutate` method. This is in contrast to the transitional mutation that selects one of the other player types rather than (possibly) generating a new player variant. To use this mutation style set *mutation_method=atomic* in the initialisation of the Moran process:

```
>>> import axelrod as axl
>>> C = axl.Action.C
>>> players = [axl.EvolvableFSMPlayer(num_states=2, initial_state=1, initial_
↪action=C) for _ in range(5)]
>>> mp = axl.MoranProcess(players, turns=10, mutation_method="atomic", seed=1)
>>> population = mp.play()
```

Note that this may cause the Moran process to fail to converge, if the mutation rates are very high or the population size very large. See *Moran Process* for more information.

Reproducible Seeding

`EvolvablePlayers` are inherently stochastic. For reproducibility of results, they can be seeded. When using the Moran process, a process level seed is sufficient. Child seeds will be created and propagated in a reproducible way. If

initialized without a seed, an `EvolvablePlayer` will be given a random seed in a non-reproducible way.

2.2.10 Access collections of strategies

All of the strategies are accessible from the main name space of the library. For example:

```
>>> import axelrod as axl
>>> axl.TitForTat()
Tit For Tat
>>> axl.Cooperator()
Cooperator
```

The **main strategies** which obey the rules of Axelrod's original tournament can be found in a list: *axelrod.strategies*:

```
>>> axl.strategies
[...
```

This makes creating a full tournament very straightforward:

```
>>> players = [s() for s in axl.strategies]
>>> tournament = axl.Tournament(players)
```

There are a list of various other strategies in the library to make it easier to create a variety of tournaments:

```
>>> axl.demo_strategies  # 5 simple strategies useful for demonstration.
[...
>>> axl.basic_strategies  # A set of basic strategies.
[...
>>> axl.long_run_time_strategies  # These have a high computational cost
[...
```

Furthermore there are some strategies that 'cheat' (for example by modifying their opponents source code). These can be found in `axelrod.cheating_strategies`:

```
>>> axl.cheating_strategies
[...
```

All of the strategies in the library are contained in: `axelrod.all_strategies`:

```
>>> axl.all_strategies
[...
```

All strategies are also classified, you can read more about that in *Classification of strategies*.

2.2.11 Classification of strategies

Due to the large number of strategies, every class and instance of the class has a `classifier` attribute which classifies that strategy according to various dimensions.

Here is the `classifier` for the `Cooperator` strategy:

```
>>> import axelrod as axl
>>> expected_dictionary = {
...     'manipulates_state': False,
...     'long_run_time': False,
```

(continues on next page)

(continued from previous page)

```
...     'stochastic': False,
...     'manipulates_source': False,
...     'inspects_source': False,
...     'memory_depth': 0
... }  # Order of this dictionary might be different on your machine
>>> axl.Cooperator.classifier == expected_dictionary
True
```

Note that instances of the class also have this classifier:

```
>>> s = axl.Cooperator()
>>> s.classifier == expected_dictionary
True
```

The instance starts with a copy of the class's classifier dictionary, but is allowed to change this classifier dictionary at any point, and many strategies do so upon initialization.

In addition to the classifier dictionary, each classifier is defined with some logic that maps classifier definitions to values. To learn the classification of a strategy, we first look in the strategy's classifier dictionary, then if the key is not present, then we refer to this logic. This logic must be defined for a class, and not specific instances.

To lookup the classifier of a strategy, using the classifier dict, or the strategy's logic as default, we use `Classifiers[<classifier>]( <strategy>)`:

```
>>> from axelrod import Classifiers
>>> Classifiers['memory_depth'](axl.TitForTat())
1
>>> Classifiers['stochastic'](axl.Random())
True
```

We can use this classification to generate sets of strategies according to filters which we define in a 'filterset' dictionary and then pass to the 'filtered_strategies' function. For example, to identify all the stochastic strategies:

```
>>> filterset = {
...     'stochastic': True
... }
>>> strategies = axl.filtered_strategies(filterset)
>>> len(strategies)
88
```

Or, to find out how many strategies only use 1 turn worth of memory to make a decision:

```
>>> filterset = {
...     'memory_depth': 1
... }
>>> strategies = axl.filtered_strategies(filterset)
>>> len(strategies)
33
```

Multiple filters can be specified within the filterset dictionary. To specify a range of memory_depth values, we can use the 'min_memory_depth' and 'max_memory_depth' filters:

```
>>> filterset = {
...     'min_memory_depth': 1,
...     'max_memory_depth': 4
... }
>>> strategies = axl.filtered_strategies(filterset)
```

(continues on next page)

(continued from previous page)

```
>>> len(strategies)
57
```

We can also identify strategies that make use of particular properties of the tournament. For example, here is the number of strategies that make use of the length of each match of the tournament:

```
>>> filterset = {
...     'makes_use_of': ['length']
... }
>>> strategies = axl.filtered_strategies(filterset)
>>> len(strategies)
22
```

Note that in the filterset dictionary, the value for the 'makes_use_of' key must be a list. Here is how we might identify the number of strategies that use both the length of the tournament and the game being played:

```
>>> filterset = {
...     'makes_use_of': ['length', 'game']
... }
>>> strategies = axl.filtered_strategies(filterset)
>>> len(strategies)
15
```

Some strategies have been classified as having a particularly long run time:

```
>>> filterset = {
...     'long_run_time': True
... }
>>> strategies = axl.filtered_strategies(filterset)
>>> len(strategies)
18
```

Strategies that `manipulate_source`, `manipulate_state` and/or `inspect_source` return `False` for the `Classifier.obey_axelrod` function:

```
>>> s = axl.Darwin()
>>> axl.Classifiers.obey_axelrod(s)
False
>>> s = axl.TitForTat()
>>> axl.Classifiers.obey_axelrod(s)
True
```

2.2.12 Strategy Transformers

What is a Strategy Transformer?

A strategy transformer is a function that modifies an existing strategy. For example, `FlipTransformer` takes a strategy and flips the actions from C to D and D to C:

```
>>> import axelrod as axl
>>> from axelrod.strategy_transformers import *
>>> FlippedCooperator = FlipTransformer()(axl.Cooperator)
>>> player = FlippedCooperator()
>>> opponent = axl.Cooperator()
```

(continues on next page)

(continued from previous page)

```
>>> player.strategy(opponent)
D
>>> opponent.strategy(player)
C
```

Our player was switched from a `Cooperator` to a `Defector` when we applied the transformer. The transformer also changed the name of the class and player:

```
>>> player.name
'Flipped Cooperator'
>>> FlippedCooperator.name
'Flipped Cooperator'
```

This behavior can be suppressed by setting the `name_prefix` argument:

```
>>> FlippedCooperator = FlipTransformer(name_prefix=None)(axl.Cooperator)
>>> player = FlippedCooperator()
>>> player.name
'Cooperator'
```

Note carefully that the transformer returns a class, not an instance of a class. This means that you need to use the Transformed class as you would normally to create a new instance:

```
>>> from axelrod.strategy_transformers import NoisyTransformer
>>> player = NoisyTransformer(0.5)(axl.Cooperator)()
```

rather than `NoisyTransformer(0.5)(axl.Cooperator())` or just `NoisyTransformer(0.5)(axl.Cooperator)`.

Included Transformers

The library includes the following transformers:

- `ApologyTransformer`: Apologizes after a round of `(D, C)`:

```
 >>> ApologizingDefector = ApologyTransformer([D], [C])(axl.Defector)
 >>> player = ApologizingDefector()

You can pass any two sequences in. In this example the player would apologize
after two consequtive rounds of `(D, C)`::

    >>> ApologizingDefector = ApologyTransformer([D, D], [C, C])(axl.Defector)
    >>> player = ApologizingDefector()
```

- `DeadlockBreakingTransformer`: Attempts to break `(D, C) -> (C, D)` deadlocks by cooperating:

```
>>> DeadlockBreakingTFT = DeadlockBreakingTransformer()(axl.TitForTat)
>>> player = DeadlockBreakingTFT()
```

- `DualTransformer`: The Dual of a strategy will return the exact opposite set of moves to the original strategy when both are faced with the same history. [Ashlock2008]:

```
>>> DualWSLS = DualTransformer()(axl.WinStayLoseShift)
>>> player = DualWSLS()
```

- `FlipTransformer`: Flips all actions:

```
>>> FlippedCooperator = FlipTransformer()(axl.Cooperator)
>>> player = FlippedCooperator()
```

- `FinalTransformer(seq=None)`: Ends the tournament with the moves in the sequence `seq`, if the tournament_length is known. For example, to obtain a cooperator that defects on the last two rounds:

```
>>> FinallyDefectingCooperator = FinalTransformer([D, D])(axl.Cooperator)
>>> player = FinallyDefectingCooperator()
```

- `ForgiverTransformer(p)`: Flips defections with probability `p`:

```
>>> ForgivinDefector = ForgiverTransformer(0.1)(axl.Defector)
>>> player = ForgivinDefector()
```

- `GrudgeTransformer(N)`: Defections unconditionally after more than N defections:

```
>>> GrudgingCooperator = GrudgeTransformer(2)(axl.Cooperator)
>>> player = GrudgingCooperator()
```

- `InitialTransformer(seq=None)`: First plays the moves in the sequence `seq`, then plays as usual. For example, to obtain a defector that cooperates on the first two rounds:

```
>>> InitiallyCooperatingDefector = InitialTransformer([C, C])(axl.Defector)
>>> player = InitiallyCooperatingDefector()
```

- `JossAnnTransformer(probability)`: Where `probability = (x, y)`, the Joss-Ann of a strategy is a new strategy which has a probability `x` of choosing the move C, a probability `y` of choosing the move D, and otherwise uses the response appropriate to the original strategy. [Ashlock2008]:

```
>>> JossAnnTFT = JossAnnTransformer((0.2, 0.3))(axl.TitForTat)
>>> player = JossAnnTFT()
```

- `MixedTransformer`: Randomly plays a mutation to another strategy (or set of strategies. Here is the syntax to do this with a set of strategies:

```
>>> strategies = [axl.Grudger, axl.TitForTat]
>>> probability = [.2, .3]  # .5 chance of mutated to one of above
>>> player =  MixedTransformer(probability, strategies)(axl.Cooperator)
```

  Here is the syntax when passing a single strategy:

```
>>> strategy = axl.Grudger
>>> probability = .2
>>> player =  MixedTransformer(probability, strategy)(axl.Cooperator)
```

- `NiceTransformer()`: Prevents a strategy from defecting if the opponent has not yet defected:

```
>>> NiceDefector = NiceTransformer()(axl.Defector)
>>> player = NiceDefector()
```

- `NoisyTransformer(noise)`: Flips actions with probability `noise`:

```
>>> NoisyCooperator = NoisyTransformer(0.5)(axl.Cooperator)
>>> player = NoisyCooperator()
```

- `RetaliationTransformer(N)`: Retaliation N times after a defection:

```
>>> TwoTitsForTat = RetaliationTransformer(2)(axl.Cooperator)
>>> player = TwoTitsForTat()
```

- `RetaliateUntilApologyTransformer()`: adds TitForTat-style retaliation:

```
>>> TFT = RetaliateUntilApologyTransformer()(axl.Cooperator)
>>> player = TFT()
```

- `TrackHistoryTransformer`: Tracks History internally in the `Player` instance in a variable `_recorded_history`. This allows a player to e.g. detect noise.:

```
>>> player = TrackHistoryTransformer()(axl.Random)()
```

Composing Transformers

Transformers can be composed to form new composers, in two ways. You can simply chain together multiple transformers:

```
>>> cls1 = FinalTransformer([D,D])(InitialTransformer([D,D])(axl.Cooperator))
>>> p1 = cls1()
```

This defines a strategy that cooperates except on the first two and last two rounds. Alternatively, you can make a new class using `compose_transformers`:

```
>>> cls1 = compose_transformers(FinalTransformer([D, D]), InitialTransformer([D, D]))
>>> p1 = cls1(axl.Cooperator)()
>>> p2 = cls1(axl.Defector)()
```

Usage as Class Decorators

Transformers can also be used to decorate existing strategies. For example, the strategy `BackStabber` defects on the last two rounds. We can encode this behavior with a transformer as a class decorator:

```
@FinalTransformer([D, D]) # End with two defections
class BackStabber(Player):
    """
    Forgives the first 3 defections but on the fourth
    will defect forever. Defects on the last 2 rounds unconditionally.
    """

    name = 'BackStabber'
    classifier = {
        'memory_depth': float('inf'),
        'stochastic': False,
        'inspects_source': False,
        'manipulates_source': False,
        'manipulates_state': False
    }

    def strategy(self, opponent):
        if not opponent.history:
            return C
        if opponent.defections > 3:
```

(continues on next page)

(continued from previous page)

```
            return D
        return C
```

Writing New Transformers

To make a new transformer, you need to define a strategy wrapping function with the following signature:

```
def strategy_wrapper(player, opponent, proposed_action, *args, **kwargs):
    """
    Strategy wrapper functions should be of the following form.

    Parameters
    ----------
    player: Player object or subclass (self)
    opponent: Player object or subclass
    proposed_action: an axelrod.Action, C or D
        The proposed action by the wrapped strategy
        proposed_action = Player.strategy(...)
    args, kwargs:
        Any additional arguments that you need.

    Returns
    -------
    action: an axelrod.Action, C or D

    """

    # This example just passes through the proposed_action
    return proposed_action
```

The proposed action will be the outcome of:

```
self.strategy(player)
```

in the underlying class (the one that is transformed). The strategy_wrapper still has full access to the player and the opponent objects and can have arguments.

To make a transformer from the `strategy_wrapper` function, use `StrategyTransformerFactory`, which has signature:

```
def StrategyTransformerFactory(strategy_wrapper, name_prefix=""):
    """Modify an existing strategy dynamically by wrapping the strategy
    method with the argument `strategy_wrapper`.

    Parameters
    ----------
    strategy_wrapper: function
        A function of the form `strategy_wrapper(player, opponent, proposed_action,␣
↪*args, **kwargs)`
        Can also use a class that implements
            def __call__(self, player, opponent, action)
    name_prefix: string, "Transformed "
        A string to prepend to the strategy and class name
    """
```

So we use `StrategyTransformerFactory` with `strategy_wrapper`:

```
TransformedClass = StrategyTransformerFactory(generic_strategy_wrapper)
Cooperator2 = TransformedClass(*args, **kwargs)(axl.Cooperator)
```

If your wrapper requires no arguments, you can simply proceed as follows:

```
>>> TransformedClass = StrategyTransformerFactory(generic_strategy_wrapper)()
>>> Cooperator2 = TransformedClass(axl.Cooperator)
```

For more examples, see `axelrod/strategy_transformers.py`.

2.2.13 Access tournament results

This tutorial will show you how to access the various results of a tournament:

- Wins: the number of matches won by each player
- Match lengths: the number of turns of each match played by each player (relevant for tournaments with probabilistic ending).
- Scores: the total scores of each player.
- Normalised scores: the scores normalised by matches played and turns.
- Ranking: ranking of players based on median score.
- Ranked names: names of players in ranked order.
- Payoffs: average payoff per turn of each player.
- Payoff matrix: the payoff matrix showing the payoffs of each row player against each column player.
- Payoff standard deviation: the standard deviation of the payoffs matrix.
- Score differences: the score difference between each player.
- Payoff difference means: the mean score differences.
- Cooperation counts: the number of times each player cooperated.
- Normalised cooperation: cooperation count per turn.
- Normalised cooperation: cooperation count per turn.
- State distribution: the count of each type of state of a match
- Normalised state distribution: the normalised count of each type of state of a match
- State to action distribution: the count of each type of state to action pair of a match
- Normalised state distribution: the normalised count of each type of state to action pair of a match
- Initial cooperation count: the count of initial cooperation by each player.
- Initial cooperation rate: the rate of initial cooperation by each player.
- Cooperation rating: cooperation rating of each player
- Vengeful cooperation: a morality metric from the literature (see *Calculate Morality Metrics*).
- Good partner matrix: a morality metric from [Singer-Clark2014].
- Good partner rating: a morality metric from [Singer-Clark2014].
- Eigenmoses rating: a morality metric from [Singer-Clark2014].
- Eigenjesus rating: a morality metric from [Singer-Clark2014].

As shown in *Creating and running a simple tournament* let us create a tournament:

```
>>> import axelrod as axl
>>> players = [axl.Cooperator(), axl.Defector(),
...            axl.TitForTat(), axl.Grudger()]
>>> tournament = axl.Tournament(players, turns=10, repetitions=3)
>>> results = tournament.play()
```

Wins

This gives the number of wins obtained by each player:

```
>>> results.wins
[[0, 0, 0], [3, 3, 3], [0, 0, 0], [0, 0, 0]]
```

The `Defector` is the only player to win any matches (all other matches are ties).

Match lengths

This gives the length of the matches played by each player:

```
>>> import pprint  # Nicer formatting of output
>>> pprint.pprint(results.match_lengths)
[[[10.0, 10.0, 10.0, 10.0],
  [10.0, 10.0, 10.0, 10.0],
  [10.0, 10.0, 10.0, 10.0],
  [10.0, 10.0, 10.0, 10.0]],
 [[10.0, 10.0, 10.0, 10.0],
  [10.0, 10.0, 10.0, 10.0],
  [10.0, 10.0, 10.0, 10.0],
  [10.0, 10.0, 10.0, 10.0]],
 [[10.0, 10.0, 10.0, 10.0],
  [10.0, 10.0, 10.0, 10.0],
  [10.0, 10.0, 10.0, 10.0],
  [10.0, 10.0, 10.0, 10.0]]]
```

Every player plays 10 turns against every other player (including themselves) for every repetition of the tournament.

Scores

This gives all the total tournament scores (per player and per repetition):

```
>>> results.scores
[[60, 60, 60], [78, 78, 78], [69, 69, 69], [69, 69, 69]]
```

Normalised scores

This gives the scores, averaged per opponent and turns:

```
>>> results.normalised_scores
[[2.0, 2.0, 2.0], [2.6, 2.6, 2.6], [2.3, 2.3, 2.3], [2.3, 2.3, 2.3]]
```

We see that Cooperator got on average a score of 2 per turn per opponent:

```
>>> results.normalised_scores[0]
[2.0, 2.0, 2.0]
```

Ranking

This gives the ranked index of each player:

```
>>> results.ranking
[1, 2, 3, 0]
```

The first player has index 1 (`Defector`) and the last has index 0 (`Cooperator`).

Ranked names

This gives the player names in ranked order:

```
>>> results.ranked_names
['Defector', 'Tit For Tat', 'Grudger', 'Cooperator']
```

Payoffs

This gives for each player, against each opponent every payoff received for each repetition:

```
>>> pprint.pprint(results.payoffs)
[[[3.0, 3.0, 3.0], [0.0, 0.0, 0.0], [3.0, 3.0, 3.0], [3.0, 3.0, 3.0]],
 [[5.0, 5.0, 5.0], [1.0, 1.0, 1.0], [1.4, 1.4, 1.4], [1.4, 1.4, 1.4]],
 [[3.0, 3.0, 3.0], [0.9, 0.9, 0.9], [3.0, 3.0, 3.0], [3.0, 3.0, 3.0]],
 [[3.0, 3.0, 3.0], [0.9, 0.9, 0.9], [3.0, 3.0, 3.0], [3.0, 3.0, 3.0]]]
```

Payoff matrix

This gives the mean payoff of each player against every opponent:

```
>>> pprint.pprint(results.payoff_matrix)
[[3.0, 0.0, 3.0, 3.0],
 [5.0, 1.0, 1.4, 1.4],
 [3.0, 0.9, 3.0, 3.0],
 [3.0, 0.9, 3.0, 3.0]]
```

We see that the `Cooperator` gets a mean score of 3 against all players except the `Defector`:

```
>>> results.payoff_matrix[0]
[3.0, 0.0, 3.0, 3.0]
```

Payoff standard deviation

This gives the standard deviation of the payoff of each player against every opponent:

```
>>> pprint.pprint(results.payoff_stddevs)
[[0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 2.2, 2.2],
 [0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0]]
```

We see that there is no variation for the payoff for `Cooperator`:

```
>>> results.payoff_stddevs[0]
[0.0, 0.0, 0.0, 0.0]
```

Score differences

This gives the score difference for each player against each opponent for every repetition:

```
>>> pprint.pprint(results.score_diffs)
[[[0.0, 0.0, 0.0], [-5.0, -5.0, -5.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
 [[5.0, 5.0, 5.0], [0.0, 0.0, 0.0], [0.5, 0.5, 0.5], [0.5, 0.5, 0.5]],
 [[0.0, 0.0, 0.0], [-0.5, -0.5, -0.5], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
 [[0.0, 0.0, 0.0], [-0.5, -0.5, -0.5], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]]
```

We see that `Cooperator` has no difference in score with all players except against the `Defector`:

```
>>> results.score_diffs[0][1]
[-5.0, -5.0, -5.0]
```

Payoff difference means

This gives the mean payoff differences over each repetition:

```
>>> pprint.pprint(results.payoff_diffs_means)
[[0.0, -5.0, 0.0, 0.0],
 [5.0, 0.0, 0.49999999999999983, 0.49999999999999983],
 [0.0, -0.49999999999999983, 0.0, 0.0],
 [0.0, -0.49999999999999983, 0.0, 0.0]]
```

Here is the mean payoff difference for the `Cooperator` strategy, shows that it has no difference with all players except against the `Defector`:

```
>>> results.payoff_diffs_means[0]
[0.0, -5.0, 0.0, 0.0]
```

Cooperation counts

This gives a total count of cooperation for each player against each opponent:

```
>>> results.cooperation
[[30, 30, 30, 30], [0, 0, 0, 0], [30, 3, 30, 30], [30, 3, 30, 30]]
```

Normalised cooperation

This gives the average rate of cooperation against each opponent:

```
>>> pprint.pprint(results.normalised_cooperation)
[[1.0, 1.0, 1.0, 1.0],
 [0.0, 0.0, 0.0, 0.0],
 [1.0, 0.1, 1.0, 1.0],
 [1.0, 0.1, 1.0, 1.0]]
```

We see that `Cooperator` for all the rounds (as expected):

```
>>> results.normalised_cooperation[0]
[1.0, 1.0, 1.0, 1.0]
```

State distribution counts

This gives a total state count against each opponent. A state corresponds to 1 turn of a match and can be one of `(C, C)`, `(C, D)`, `(D, C)`, `(D, D)` where the first element is the action of the player in question and the second the action of the opponent:

```
>>> pprint.pprint(results.state_distribution)
[[Counter(),
  Counter({(C, D): 30}),
  Counter({(C, C): 30}),
  Counter({(C, C): 30})],
 [Counter({(D, C): 30}),
  Counter(),
  Counter({(D, D): 27, (D, C): 3}),
  Counter({(D, D): 27, (D, C): 3})],
 [Counter({(C, C): 30}),
  Counter({(D, D): 27, (C, D): 3}),
  Counter(),
  Counter({(C, C): 30})],
 [Counter({(C, C): 30}),
  Counter({(D, D): 27, (C, D): 3}),
  Counter({(C, C): 30}),
  Counter()]]
```

Normalised state distribution

This gives the average rate state distribution against each opponent. A state corresponds to 1 turn of a match and can be one of `(C, C)`, `(C, D)`, `(D, C)`, `(D, D)` where the first element is the action of the player in question and the second the action of the opponent:

```
>>> pprint.pprint(results.normalised_state_distribution)
[[Counter(),
  Counter({(C, D): 1.0}),
  Counter({(C, C): 1.0}),
  Counter({(C, C): 1.0})],
 [Counter({(D, C): 1.0}),
  Counter(),
  Counter({(D, D): 0.9..., (D, C): 0.1...}),
  Counter({(D, D): 0.9..., (D, C): 0.1...})],
 [Counter({(C, C): 1.0}),
  Counter({(D, D): 0.9..., (C, D): 0.1...}),
  Counter(),
  Counter({(C, C): 1.0})],
```

(continues on next page)

(continued from previous page)

```
 [Counter({(C, C): 1.0}),
  Counter({(D, D): 0.9..., (C, D): 0.1...}),
  Counter({(C, C): 1.0}),
  Counter()]]
```

State to action distribution counts

This gives a total state action pair count against each opponent. A state corresponds to 1 turn of a match and can be one of `(C, C)`, `(C, D)`, `(D, C)`, `(D, D)` where the first element is the action of the player in question and the second the action of the opponent:

```
>>> pprint.pprint(results.state_to_action_distribution)
[[Counter(),
  Counter({((C, D), C): 27}),
  Counter({((C, C), C): 27}),
  Counter({((C, C), C): 27})],
 [Counter({((D, C), D): 27}),
  Counter(),
  Counter({((D, D), D): 24, ((D, C), D): 3}),
  Counter({((D, D), D): 24, ((D, C), D): 3})],
 [Counter({((C, C), C): 27}),
  Counter({((D, D), D): 24, ((C, D), D): 3}),
  Counter(),
  Counter({((C, C), C): 27})],
 [Counter({((C, C), C): 27}),
  Counter({((D, D), D): 24, ((C, D), D): 3}),
  Counter({((C, C), C): 27}),
  Counter()]]
```

Normalised state to action distribution

This gives the average rate state to action pair distribution against each opponent. A state corresponds to 1 turn of a match and can be one of `(C, C)`, `(C, D)`, `(D, C)`, `(D, D)` where the first element is the action of the player in question and the second the action of the opponent:

```
>>> pprint.pprint(results.normalised_state_to_action_distribution)
[[Counter(),
  Counter({((C, D), C): 1.0}),
  Counter({((C, C), C): 1.0}),
  Counter({((C, C), C): 1.0})],
 [Counter({((D, C), D): 1.0}),
  Counter(),
  Counter({((D, C), D): 1.0, ((D, D), D): 1.0}),
  Counter({((D, C), D): 1.0, ((D, D), D): 1.0})],
 [Counter({((C, C), C): 1.0}),
  Counter({((C, D), D): 1.0, ((D, D), D): 1.0}),
  Counter(),
  Counter({((C, C), C): 1.0})],
 [Counter({((C, C), C): 1.0}),
  Counter({((C, D), D): 1.0, ((D, D), D): 1.0}),
  Counter({((C, C), C): 1.0}),
  Counter()]]
```

Initial cooperation counts

This gives the count of cooperations made by each player during the first turn of every match:

```
>>> results.initial_cooperation_count
[9, 0, 9, 9]
```

Each player plays an opponent a total of 9 times (3 opponents and 3 repetitions). Apart from the `Defector`, they all cooperate on the first turn.

Initial cooperation rates

This gives the rate of which a strategy cooperates during the first turn:

```
>>> results.initial_cooperation_rate
[1.0, 0.0, 1.0, 1.0]
```

Morality Metrics

The following morality metrics are available, they are calculated as a function of the cooperation rating:

```
>>> results.cooperating_rating
[1.0, 0.0, 0.7, 0.7]
>>> pprint.pprint(results.vengeful_cooperation)
[[1.0, 1.0, 1.0, 1.0],
 [-1.0, -1.0, -1.0, -1.0],
 [1.0, -0.8, 1.0, 1.0],
 [1.0, -0.78 1.0, 1.0]]
>>> pprint.pprint(results.good_partner_matrix)
[[0, 3, 3, 3], [0, 0, 0, 0], [3, 3, 0, 3], [3, 3, 3, 0]]
>>> pprint.pprint(results.good_partner_rating)
[1.0, 0.0, 1.0, 1.0]
>>> results.eigenmoses_rating
[0.37..., -0.37..., 0.59..., 0.59...]
>>> results.eigenjesus_rating
[0.57..., 0.0, 0.57..., 0.57...]
```

For more information about these see *Calculate Morality Metrics*.

2.2.14 Read and write interactions from/to file

When dealing with large tournaments it might be desirable to separate the analysis from the actual running of the tournaments. This can be done by passing a `filename` argument to the `play` method of a tournament:

```
>>> import axelrod as axl
>>> players = [s() for s in axl.basic_strategies]
>>> tournament = axl.Tournament(players, turns=4, repetitions=2)
>>> results = tournament.play(filename="basic_tournament.csv")
```

This will create a file *basic_tournament.csv* with data that looks something like:

```
Interaction index,Player index,Opponent index,Repetition,Player name,Opponent name,
↪Actions,Score,Score difference,Turns,Score per turn,Score difference per turn,Win,
↪Initial cooperation,Cooperation count,CC count,CD count,DC count,DD count,CC to C␣
↪count,CC to D count,CD to C count,CD to D count,DC to C count,DC to D count,DD to C␣
↪count,DD to D count,Good partner
0,0,0,0,Alternator,Alternator,CDCD,8,0,4,2.0,0.0,0,True,2,2,0,0,2,0,2,0,0,0,0,1,0,1
0,0,0,0,Alternator,Alternator,CDCD,8,0,4,2.0,0.0,0,True,2,2,0,0,2,0,2,0,0,0,0,1,0,1
1,0,0,1,Alternator,Alternator,CDCD,8,0,4,2.0,0.0,0,True,2,2,0,0,2,0,2,0,0,0,0,1,0,1
1,0,0,1,Alternator,Alternator,CDCD,8,0,4,2.0,0.0,0,True,2,2,0,0,2,0,2,0,0,0,0,1,0,1
2,0,1,0,Alternator,Anti Tit For Tat,CDCD,8,0,4,2.0,0.0,0,True,2,2,0,0,2,0,2,0,0,0,0,1,
↪0,1
2,1,0,0,Anti Tit For Tat,Alternator,CDCD,8,0,4,2.0,0.0,0,True,2,2,0,0,2,0,2,0,0,0,0,1,
↪0,1
3,0,1,1,Alternator,Anti Tit For Tat,CDCD,8,0,4,2.0,0.0,0,True,2,2,0,0,2,0,2,0,0,0,0,1,
↪0,1
3,1,0,1,Anti Tit For Tat,Alternator,CDCD,8,0,4,2.0,0.0,0,True,2,2,0,0,2,0,2,0,0,0,0,1,
↪0,1
4,0,2,0,Alternator,Bully,CDCD,5,-5,4,1.25,-1.25,0,True,2,1,1,0,2,0,1,0,1,0,0,1,0,1
4,2,0,0,Bully,Alternator,DDCD,10,5,4,2.5,1.25,1,False,1,1,0,1,2,0,1,0,0,0,1,1,0,0
5,0,2,1,Alternator,Bully,CDCD,5,-5,4,1.25,-1.25,0,True,2,1,1,0,2,0,1,0,1,0,0,1,0,1
5,2,0,1,Bully,Alternator,DDCD,10,5,4,2.5,1.25,1,False,1,1,0,1,2,0,1,0,0,0,1,1,0,0
6,0,3,0,Alternator,Cooperator,CDCD,16,10,4,4.0,2.5,1,True,2,2,0,2,0,0,2,0,0,1,0,0,0,0
6,3,0,0,Cooperator,Alternator,CCCC,6,-10,4,1.5,-2.5,0,True,4,2,2,0,0,2,0,1,0,0,0,0,0,1
7,0,3,1,Alternator,Cooperator,CDCD,16,10,4,4.0,2.5,1,True,2,2,0,2,0,0,2,0,0,1,0,0,0,0
7,3,0,1,Cooperator,Alternator,CCCC,6,-10,4,1.5,-2.5,0,True,4,2,2,0,0,2,0,1,0,0,0,0,0,1
8,0,4,0,Alternator,Cycler DC,CDCD,10,0,4,2.5,0.0,0,True,2,0,2,2,0,0,0,0,2,1,0,0,0,1
8,4,0,0,Cycler DC,Alternator,DCDC,10,0,4,2.5,0.0,0,False,2,0,2,2,0,0,0,0,1,2,0,0,0,1
9,0,4,1,Alternator,Cycler DC,CDCD,10,0,4,2.5,0.0,0,True,2,0,2,2,0,0,0,0,2,1,0,0,0,1
9,4,0,1,Cycler DC,Alternator,DCDC,10,0,4,2.5,0.0,0,False,2,0,2,2,0,0,0,0,1,2,0,0,0,1
10,0,5,0,Alternator,Defector,CDCD,2,-10,4,0.5,-2.5,0,True,2,0,2,0,2,0,0,0,2,0,0,1,0,1
10,5,0,0,Defector,Alternator,DDDD,12,10,4,3.0,2.5,1,False,0,0,0,2,2,0,0,0,0,0,2,0,1,0
11,0,5,1,Alternator,Defector,CDCD,2,-10,4,0.5,-2.5,0,True,2,0,2,0,2,0,0,0,2,0,0,1,0,1
11,5,0,1,Defector,Alternator,DDDD,12,10,4,3.0,2.5,1,False,0,0,0,2,2,0,0,0,0,0,2,0,1,0
12,0,6,0,Alternator,Grudger,CDCD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,0,1,0,1,1,0,0,0,1
12,6,0,0,Grudger,Alternator,CCDD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,1,0,0,1,0,1,0,0,1
13,0,6,1,Alternator,Grudger,CDCD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,0,1,0,1,1,0,0,0,1
13,6,0,1,Grudger,Alternator,CCDD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,1,0,0,1,0,1,0,0,1
14,0,7,0,Alternator,Suspicious Tit For Tat,CDCD,10,0,4,2.5,0.0,0,True,2,0,2,2,0,0,0,0,
↪2,1,0,0,0,1
14,7,0,0,Suspicious Tit For Tat,Alternator,DCDC,10,0,4,2.5,0.0,0,False,2,0,2,2,0,0,0,
↪0,1,2,0,0,0,1
15,0,7,1,Alternator,Suspicious Tit For Tat,CDCD,10,0,4,2.5,0.0,0,True,2,0,2,2,0,0,0,0,
↪2,1,0,0,0,1
15,7,0,1,Suspicious Tit For Tat,Alternator,DCDC,10,0,4,2.5,0.0,0,False,2,0,2,2,0,0,0,
↪0,1,2,0,0,0,1
16,0,8,0,Alternator,Tit For Tat,CDCD,13,5,4,3.25,1.25,1,True,2,1,1,2,0,0,1,0,1,1,0,0,
↪0,0
16,8,0,0,Tit For Tat,Alternator,CCDC,8,-5,4,2.0,-1.25,0,True,3,1,2,1,0,1,0,0,1,1,0,0,
↪0,1
17,0,8,1,Alternator,Tit For Tat,CDCD,13,5,4,3.25,1.25,1,True,2,1,1,2,0,0,1,0,1,1,0,0,
↪0,0
17,8,0,1,Tit For Tat,Alternator,CCDC,8,-5,4,2.0,-1.25,0,True,3,1,2,1,0,1,0,0,1,1,0,0,
↪0,1
18,0,9,0,Alternator,Win-Shift Lose-Stay: D,CDCD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,0,1,0,
↪1,1,0,0,0,1
18,9,0,0,Win-Shift Lose-Stay: D,Alternator,DCCD,9,0,4,2.25,0.0,0,False,2,1,1,1,1,0,1,
↪1,0,1,0,0,0,1
```

(continues on next page)

(continued from previous page)

```
19,0,9,1,Alternator,Win-Shift Lose-Stay: D,CDCD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,0,1,0,
↪1,1,0,0,0,1
19,9,0,1,Win-Shift Lose-Stay: D,Alternator,DCCD,9,0,4,2.25,0.0,0,False,2,1,1,1,1,0,1,
↪1,0,1,0,0,0,1
20,0,10,0,Alternator,Win-Stay Lose-Shift: C,CDCD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,0,1,
↪0,1,1,0,0,0,1
20,10,0,0,Win-Stay Lose-Shift: C,Alternator,CCDD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,1,0,
↪0,1,0,1,0,0,1
21,0,10,1,Alternator,Win-Stay Lose-Shift: C,CDCD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,0,1,
↪0,1,1,0,0,0,1
21,10,0,1,Win-Stay Lose-Shift: C,Alternator,CCDD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,1,0,
↪0,1,0,1,0,0,1
22,1,1,0,Anti Tit For Tat,Anti Tit For Tat,CDCD,8,0,4,2.0,0.0,0,True,2,2,0,0,2,0,2,0,
↪0,0,0,1,0,1
22,1,1,0,Anti Tit For Tat,Anti Tit For Tat,CDCD,8,0,4,2.0,0.0,0,True,2,2,0,0,2,0,2,0,
↪0,0,0,1,0,1
23,1,1,1,Anti Tit For Tat,Anti Tit For Tat,CDCD,8,0,4,2.0,0.0,0,True,2,2,0,0,2,0,2,0,
↪0,0,0,1,0,1
23,1,1,1,Anti Tit For Tat,Anti Tit For Tat,CDCD,8,0,4,2.0,0.0,0,True,2,2,0,0,2,0,2,0,
↪0,0,0,1,0,1
24,1,2,0,Anti Tit For Tat,Bully,CCCC,0,-20,4,0.0,-5.0,0,True,4,0,4,0,0,0,0,3,0,0,0,0,
↪0,1
24,2,1,0,Bully,Anti Tit For Tat,DDDD,20,20,4,5.0,5.0,1,False,0,0,0,4,0,0,0,0,0,0,3,0,
↪0,0
25,1,2,1,Anti Tit For Tat,Bully,CCCC,0,-20,4,0.0,-5.0,0,True,4,0,4,0,0,0,0,3,0,0,0,0,
↪0,1
25,2,1,1,Bully,Anti Tit For Tat,DDDD,20,20,4,5.0,5.0,1,False,0,0,0,4,0,0,0,0,0,0,3,0,
↪0,0
26,1,3,0,Anti Tit For Tat,Cooperator,CDDD,18,15,4,4.5,3.75,1,True,1,1,0,3,0,0,1,0,0,0,
↪2,0,0,0
26,3,1,0,Cooperator,Anti Tit For Tat,CCCC,3,-15,4,0.75,-3.75,0,True,4,1,3,0,0,1,0,2,0,
↪0,0,0,0,1
27,1,3,1,Anti Tit For Tat,Cooperator,CDDD,18,15,4,4.5,3.75,1,True,1,1,0,3,0,0,1,0,0,0,
↪2,0,0,0
27,3,1,1,Cooperator,Anti Tit For Tat,CCCC,3,-15,4,0.75,-3.75,0,True,4,1,3,0,0,1,0,2,0,
↪0,0,0,0,1
28,1,4,0,Anti Tit For Tat,Cycler DC,CCDC,7,-5,4,1.75,-1.25,0,True,3,2,1,0,1,0,1,1,0,0,
↪0,1,0,1
28,4,1,0,Cycler DC,Anti Tit For Tat,DCDC,12,5,4,3.0,1.25,1,False,2,2,0,1,1,0,1,0,0,1,
↪0,1,0,0
29,1,4,1,Anti Tit For Tat,Cycler DC,CCDC,7,-5,4,1.75,-1.25,0,True,3,2,1,0,1,0,1,1,0,0,
↪0,1,0,1
29,4,1,1,Cycler DC,Anti Tit For Tat,DCDC,12,5,4,3.0,1.25,1,False,2,2,0,1,1,0,1,0,0,1,
↪0,1,0,0
30,1,5,0,Anti Tit For Tat,Defector,CCCC,0,-20,4,0.0,-5.0,0,True,4,0,4,0,0,0,0,3,0,0,0,
↪0,0,1
30,5,1,0,Defector,Anti Tit For Tat,DDDD,20,20,4,5.0,5.0,1,False,0,0,0,4,0,0,0,0,0,0,3,
↪0,0,0
31,1,5,1,Anti Tit For Tat,Defector,CCCC,0,-20,4,0.0,-5.0,0,True,4,0,4,0,0,0,0,3,0,0,0,
↪0,0,1
31,5,1,1,Defector,Anti Tit For Tat,DDDD,20,20,4,5.0,5.0,1,False,0,0,0,4,0,0,0,0,0,0,3,
↪0,0,0
32,1,6,0,Anti Tit For Tat,Grudger,CDDC,9,0,4,2.25,0.0,0,True,2,1,1,1,1,0,1,0,0,0,1,1,
↪0,1
32,6,1,0,Grudger,Anti Tit For Tat,CCDD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,1,0,0,1,0,0,0,
↪1,1
33,1,6,1,Anti Tit For Tat,Grudger,CDDC,9,0,4,2.25,0.0,0,True,2,1,1,1,1,0,1,0,0,0,1,1,
↪0,1
```

(continues on next page)

(continued from previous page)

```
33,6,1,1,Grudger,Anti Tit For Tat,CCDD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,1,0,0,1,0,0,0,
↪1,1
34,1,7,0,Anti Tit For Tat,Suspicious Tit For Tat,CCDD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,
↪0,1,1,0,0,1,0,0,1
34,7,1,0,Suspicious Tit For Tat,Anti Tit For Tat,DCCD,9,0,4,2.25,0.0,0,False,2,1,1,1,
↪1,1,0,0,1,1,0,0,0,1
35,1,7,1,Anti Tit For Tat,Suspicious Tit For Tat,CCDD,9,0,4,2.25,0.0,0,True,2,1,1,1,1,
↪0,1,1,0,0,1,0,0,1
35,7,1,1,Suspicious Tit For Tat,Anti Tit For Tat,DCCD,9,0,4,2.25,0.0,0,False,2,1,1,1,
↪1,1,0,0,1,1,0,0,0,1
...
```

Note that depending on the order in which the matches have been played, the rows could also be in a different order.

It is possible to read in this data file to obtain interactions:

```
>>> interactions = axl.interaction_utils.read_interactions_from_file("basic_
↪tournament.csv")
```

This gives a dictionary mapping pairs of player indices to interaction histories:

```
>>> interactions[(0, 1)]
[[(C, C), (D, D), (C, C), (D, D)], [(C, C), (D, D), (C, C), (D, D)]]
```

This should allow for easy manipulation of data outside of the capabilities within the library.

Note that you can supply *build_results=False* as a keyword argument to *tournament.play()* to prevent keeping or loading interactions in memory, since the total memory footprint can be large for various combinations of parameters. The memory usage scales as $O(\text{players}^2 \times \text{turns} \times \text{repetitions})$.

2.2.15 Use Parallel processing

When dealing with large tournaments on a multi core machine it is possible to run the tournament in parallel **although this is not currently supported on Windows**. Using `processes=0` will simply use all available cores:

```
>>> import axelrod as axl
>>> players = [s() for s in axl.basic_strategies]
>>> tournament = axl.Tournament(players, turns=4, repetitions=2)
>>> results = tournament.play(processes=0)
```

2.2.16 Use a cache

Whilst for stochastic strategies, every repetition of a Match will give a different result, for deterministic strategies, when there is no noise there is no need to re run the match. The library has a `DeterministicCache` class that allows us to quickly replay matches.

Caching a Match

To illustrate this, let us time the play of a match **without** a cache:

```
>>> import axelrod as axl
>>> import timeit
>>> def run_match():
```

(continues on next page)

(continued from previous page)

```
...     p1, p2 = axl.GoByMajority(), axl.Alternator()
...     match = axl.Match((p1, p2), turns=200)
...     return match.play()
>>> time_with_no_cache = timeit.timeit(run_match, number=500)
>>> time_with_no_cache
2.2295279502868652
```

Here is how to create a new empty cache:

```
>>> cache = axl.DeterministicCache()
>>> len(cache)
0
```

Let us rerun the above match but using the cache:

```
>>> p1, p2 = axl.GoByMajority(), axl.Alternator()
>>> match = axl.Match((p1, p2), turns=200, deterministic_cache=cache)
>>> match.play()
[(C, C), ..., (C, D)]
```

We can take a look at the cache:

```
>>> cache
{('Soft Go By Majority', 'Alternator'): [(C, C), ..., (C, D)]}
>>> len(cache)
1
>>> len(cache[(axl.GoByMajority(), axl.Alternator())])
200
```

This maps a triplet of 2 player names and the match length to the resulting interactions. We can rerun the code and compare the timing:

```
>>> def run_match_with_cache():
...     p1, p2 = axl.GoByMajority(), axl.Alternator()
...     match = axl.Match((p1, p2), turns=200, deterministic_cache=cache)
...     return match.play()
>>> time_with_cache = timeit.timeit(run_match_with_cache, number=500)
>>> time_with_cache
0.04215192794799805
>>> time_with_cache < time_with_no_cache
True
```

We can write the cache to file:

```
>>> cache.save("cache.txt")
True
```

Caching a Tournament

Tournaments will automatically create caches as needed on a match by match basis.

Caching a Moran Process

A prebuilt cache can also be used in a Moran process (by default a new cache is used):

```
>>> cache = axl.DeterministicCache("cache.txt")
>>> players = [axl.GoByMajority(), axl.Alternator(),
...            axl.Cooperator(), axl.Grudger()]
>>> mp = axl.MoranProcess(players, deterministic_cache=cache)
>>> populations = mp.play()
>>> mp.winning_strategy_name
Defector
```

We see that the cache has been augmented, although note that this particular number will depend on the stochastic behaviour of the Moran process:

```
>>> len(cache)
18
```

2.2.17 Use different stage games

As described in *Play Contexts and Generic Prisoner's Dilemma* the default game used for the Prisoner's Dilemma is given by:

```
>>> import axelrod as axl
>>> pd = axl.game.Game()
>>> pd
Axelrod game: (R,P,S,T) = (3, 1, 0, 5)
>>> pd.RPST()
(3, 1, 0, 5)
```

These `Game` objects are used to score *matches*, *tournaments* and *Moran processes*:

```
>>> pd.score((axl.Action.C, axl.Action.C))
(3, 3)
>>> pd.score((axl.Action.C, axl.Action.D))
(0, 5)
>>> pd.score((axl.Action.D, axl.Action.C))
(5, 0)
>>> pd.score((axl.Action.D, axl.Action.D))
(1, 1)
```

It is possible to run a matches, tournaments and Moran processes with a different game. For example here is the game of chicken:

```
>>> chicken = axl.game.Game(r=0, s=-1, t=1, p=-10)
>>> chicken
Axelrod game: (R,P,S,T) = (0, -10, -1, 1)
>>> chicken.RPST()
(0, -10, -1, 1)
```

Here is a simple tournament run with this game:

```
>>> players = [axl.Cooperator(), axl.Defector(), axl.TitForTat()]
>>> tournament = axl.Tournament(players, game=chicken)
>>> results = tournament.play()
>>> results.ranked_names
['Cooperator', 'Defector', 'Tit For Tat']
```

The default Prisoner's dilemma has different results:

```
>>> tournament = axl.Tournament(players)
>>> results = tournament.play()
>>> results.ranked_names
['Defector', 'Tit For Tat', 'Cooperator']
```

Asymmetric games can also be implemented via the AsymmetricGame class with two Numpy arrays for payoff matrices:

```
>>> import numpy as np
>>> A = np.array([[3, 1], [1, 3]])
>>> B = np.array([[1, 3], [2, 1]])
>>> asymmetric_game = axl.AsymmetricGame(A, B)
>>> asymmetric_game
Axelrod game with matrices: (array([[3, 1],
                                    [1, 3]]),
                             array([[1, 3],
                                    [2, 1]]))
```

Asymmetric games can also be different sizes (even if symmetric; regular games can currently only be 2x2), such as Rock Paper Scissors:

```
>>> A = np.array([[0, -1, 1], [1, 0, -1], [-1, 1, 0]])
>>> rock_paper_scissors = axl.AsymmetricGame(A, -A)
>>> rock_paper_scissors
Axelrod game with matrices: (array([[ 0, -1,  1],
                                    [ 1,  0, -1],
                                    [-1,  1,  0]]),
                             array([[ 0,  1, -1],
                                    [-1,  0,  1],
                                    [ 1, -1,  0]]))
```

NB: Some features of Axelrod, such as strategy transformers, are specifically created for use with the iterated Prisoner's Dilemma; they may break with games of other sizes. Note also that most strategies in Axelrod are Prisoners' Dilemma strategies, so behave as though they are playing the Prisoners' Dilemma; in the rock-paper-scissors example above, they will certainly never choose scissors (because their strategy action set is two actions!)

For a more detailed tutorial on how to implement another game into Axelrod, *here is a tutorial using rock paper scissors as an example.*

2.2.18 Use custom matches

The Moran process supports custom match classes. Below creates a new class of a match where both players end with a score of 2:

```
>>> import axelrod as axl
>>> class MassBaseMatch(axl.Match):
...     """Axelrod Match object with a modified final score function to enable mass␣
↪to influence the final score as a multiplier"""
...     def final_score_per_turn(self):
...         return 2, 2
```

We then create a Moran process with the custom match class by passing our custom `MassBaseMatch` to the Moran process with the `match_class` keyword argument:

```
>>> players = [axl.Cooperator(), axl.Defector(), axl.TitForTat(), axl.Grudger()]
>>> mp = axl.MoranProcess(players=players, match_class=MassBaseMatch, seed=0)
>>> population = mp.play()
>>> print(mp.winning_strategy_name)
Defector
```

2.2.19 Set a random seed

The library has a variety of strategies whose behaviour is stochastic. To ensure reproducible results a random seed should be set. The library abstracts away the propagation of seeds in matches and tournaments, so you typically only need to supply a seed to those objects.

Matches

For a match, set a seed by passing a parameter to *Match*

```
>>> import axelrod as axl
>>> players = (axl.Random(), axl.MetaMixer())  # Two stochastic strategies
>>> match = axl.Match(players, turns=3, seed=101)
>>> results = match.play()
```

We obtain the same results if it is played with the same seed:

```
>>> match2 = axl.Match(players, turns=3, seed=101)
>>> result2 = match2.play()
>>> results == result2
True
```

For noisy matches, a seed also needs to be set for reproducibility, even if the players are deterministic.

```
>>> import axelrod as axl
>>> players = (axl.Cooperator(), axl.Defector())  # Two deterministic strategies
>>> match = axl.Match(players, turns=200, seed=101, noise=0.25)
>>> results = match.play()
>>> match2 = axl.Match(players, turns=200, seed=101, noise=0.25)
>>> results2 = match2.play()
>>> results == results2
True
```

Tournaments

For tournaments, an initial seed is used to generate subsequent seeds for each match in a a manner that will yield identical results. Note that if the tournament is run with multiple processes, the order of the matches may be computed differently, but the seeds used for each match will be the same.

To seed a tournament we also pass a seed to the tournament at creation time:

```
>>> import axelrod as axl
>>> seed = 201
>>> players = (axl.Random(), axl.Cooperator(), axl.MetaMixer())
>>> tournament = axl.Tournament(players, turns=5, repetitions=5, seed=seed)
>>> results = tournament.play(processes=1)
>>> tournament2 = axl.Tournament(players, turns=5, repetitions=5, seed=seed)
```

(continues on next page)

(continued from previous page)

```
>>> results2 = tournament2.play(processes=1)
>>> results.ranked_names == results2.ranked_names
True
```

For parallel processing, the ordering of match results may differ, but the actual results, and the final rankings, will be the same.

```
>>> import axelrod as axl
>>> players = (axl.Random(), axl.Cooperator(), axl.MetaMixer())
>>> tournament = axl.Tournament(players, turns=5, repetitions=5, seed=201)
>>> results = tournament.play(processes=2)
>>> tournament2 = axl.Tournament(players, turns=5, repetitions=5, seed=201)
>>> results2 = tournament2.play(processes=2)
>>> results.ranked_names == results2.ranked_names
True
```

Moran Process

Similarly, a Moran process is essentially another type of tournament. The library's implementation will propagate child seeds to each match to ensure reproducibility. See also the documentation on `EvolvablePlayers`.

Fingerprints

Since fingerprint generation depends on tournaments, fingerprints can also be given a seed for reproducibility.

2.2.20 Set Player information

It is possible to determine what information players know about their matches. By default all known information is given. For example let us create a match with 5 turns between `FirstBySteinAndRapoport` and `Alternator`. The latter of these two always defects on the last 2 turns:

```
>>> import axelrod as axl
>>> players = (axl.Alternator(), axl.FirstBySteinAndRapoport())
>>> axl.Match(players, turns=5).play()
[(C, C), (D, C), (C, C), (D, D), (C, D)]
```

We can play the same match but let us tell the players that the match lasts 6 turns:

```
>>> axl.Match(players, turns=5, match_attributes={"length": 6}).play()
[(C, C), (D, C), (C, C), (D, C), (C, D)]
```

We can also pass this information to a tournament. Let us create a tournament with 5 turns but ensure the players believe the match length is infinite (unknown):

```
>>> tournament = axl.Tournament(players, turns=5,
...                             match_attributes={"length": float('inf')})
```

The `match_attributes` dictionary can also be used to pass `game` and `noise`.

2.2.21 Check Player equality

It is possible to test for player equality using ==:

```
>>> import axelrod as axl
>>> p1, p2, p3 = axl.Alternator(), axl.Alternator(), axl.TitForTat()
>>> p1 == p2
True
>>> p1 == p3
False
```

Note that this checks all the attributes of an instance:

```
>>> p1.name = "John Nash"
>>> p1 == p2
False
```

This however does not check if the players will behave in the same way. For example here are two equivalent players:

```
>>> p1 = axl.Alternator()
>>> p2 = axl.Cycler("CD")
>>> p1 == p2
False
```

To check if player strategies are equivalent you can use *Fingerprint*.

2.2.22 Cite the library

We would be delighted if anyone wanted to use and/or reference this library for their own research.

If you do please let us know and reference the library: as described in the CITATION.rst file on the library repository.

2.2.23 Contribute

This section contains a variety of tutorials that should help you contribute to the library.

Contents:

Guidelines

All contributions to this repository are welcome via pull request on the github repository.

The project follows the following guidelines:

1. Use the base Python library unless completely necessary. A few external libraries (such as numpy) have been included in requirements.txt – feel free to use these as needed.
2. Try as best as possible to follow PEP8 which includes **using descriptive variable names**.
3. Code Format: Use the Black formatter to format all code and the isort utility to sort import statements. You can run black on all code with:

   ```
   $ python -m black -l 80 .
   ```

4. Commits: Please try to use commit messages that give a meaningful history for anyone using git's log features. Try to use messages that complete sentence, "This commit will…" There is some excellent guidance on the subject from Chris Beams
5. Testing: the project uses the unittest library and has a nice testing suite that makes some things easy to write tests for. Please try to increase the test coverage on pull requests.

6. Merging pull-requests: We require two of the (currently three) core-team maintainers to merge. Opening a PR for early feedback or to check test coverage is OK, just indicate that the PR is not ready to merge (and update when it is).

By submitting a pull request, you are agreeing that your work may be distributed under the terms of the project's licence and you will become one of the project's joint copyright holders.

Setting up the environment

Installing all dependencies

All development dependencies can be installed by running:

```
$ pip install -r requirements/development.txt
```

It is recommended to do this using a virtual environment tool of your choice.

For example, when using the virtual environment library `venv`:

```
$ python -m venv axelrod_development
$ source axelrod_development/bin/activate
$ pip install -r requirements/development.txt
```

Alternatively, you can specify the development variant rather than the path:

```
$ python -m venv axelrod_development
$ source axelrod_development/bin/activate
$ pip install .[development]
```

The git workflow

There are two important branches in this repository:

- `dev`: The most up to date branch with no failing tests. This is the default branch on github.
- `release`: The latest release.

When working on a new contribution branch from the latest `dev` branch and open a Pull Request on github from your branch to the `dev` branch.

The procedure for a new release (this is carried out by one of core maintainers):

1. Create a Pull Request from `dev` to `release` which should include an update to `axelrod/version.py` and `CHANGES.md`
2. Create a git tag.
3. Push to github.
4. Create a release on github.
5. Push to PyPi: `python setup.py sdist bdist_wheel upload`

Contributing a strategy

This section contains a variety of tutorials that should help you contribute a new strategy to the library.

Contents:

Instructions

Here is the file structure for the Axelrod repository:

```
.
├── axelrod
│   └── __init__.py
│   └── ecosystem.py
│   └── game.py
│   └── player.py
│   └── plot.py
│   └── result_set.py
│   └── round_robin.py
│   └── tournament.py
│   └── /strategies/
│       └── __init__.py
│       └── _strategies.py
│       └── cooperator.py
│       └── defector.py
│       └── grudger.py
│       └── titfortat.py
│       └── gobymajority.py
│       └── ...
│   └── /tests/
│       └── integration
│       └── strategies
│       └── unit
│           └── test_*.py
└── README.md
```

To contribute a strategy you need to follow as many of the following steps as possible:

1. Fork the github repository.
2. Add a `<strategy>.py` file to the strategies directory or add a strategy to a pre existing `<strategy>.py` file.
3. Update the `./axelrod/strategies/_strategies.py` file.
4. If you created a new `<strategy>.py` file add it to `.docs/reference/all_strategies.rst`.
5. Write some unit tests in the `./axelrod/tests/strategies/` directory.
6. This one is also optional: ping us a message and we'll add you to the Contributors team. This would add an Axelrod-Python organisation badge to your profile.
7. Send us a pull request.

If you would like a hand with any of the above please do get in touch: we're always delighted to have new strategies.

Writing the new strategy

Identify a new strategy

If you're not sure if you have a strategy that has already been implemented, you can search the *Strategy index* to see if they are implemented. If you are still unsure please get in touch: via the gitter room or open an issue.

Several strategies are special cases of other strategies. For example, both `Cooperator` and `Defector` are special cases of `Random`, `Random(1)` and `Random(0)` respectively. While we could eliminate `Cooperator` in its current form, these strategies are intentionally left as is as simple examples for new users and contributors. Nevertheless, please feel free to update the docstrings of strategies like `Random` to point out such cases.

The code

There are a couple of things that need to be created in a strategy.py file. Let us take a look at the *TitForTat* class (located in the `axelrod/strategies/titfortat.py` file):

```
class TitForTat(Player):
    """
    A player starts by cooperating and then mimics previous move by
    opponent.

    Note that the code for this strategy is written in a fairly verbose
    way. This is done so that it can serve as an example strategy for
    those who might be new to Python.

    Names

    - Rapoport's strategy: [Axelrod1980]_
    - TitForTat: [Axelrod1980]_
    """

    # These are various properties for the strategy
    name = 'Tit For Tat'
    classifier = {
        'memory_depth': 1,  # Four-Vector = (1.,0.,1.,0.)
        'stochastic': False,
        'inspects_source': False,
        'manipulates_source': False,
        'manipulates_state': False
    }

    def strategy(self, opponent):
        """This is the actual strategy"""
        # First move
        if len(self.history) == 0:
            return C
        # React to the opponent's last move
        if opponent.history[-1] == D:
            return D
        return C
```

The first thing that is needed is a docstring that explains what the strategy does:

```
"""A player starts by cooperating and then mimics previous move by opponent."""
```

Secondly, any alternate names should be included and if possible references provided (this helps when trying to identify if a strategy has already been implemented or not):

```
- Rapoport's strategy: [Axelrod1980]_
- TitForTat: [Axelrod1980]_
```

These references can be found in the *Bibliography*. If a required references is not there please feel free to add it or just get in touch and we'd be happy to help.

After that simply add in the string that will appear as the name of the strategy:

```
name = 'Tit For Tat'
```

Note that this is mainly used in plots by `matplotlib` so you can use LaTeX if you want to. For example there is strategy with π as a name:

```
name = '$\pi$'
```

Following that you can add in the `classifier` dictionary:

```
classifier = {
    'memory_depth': 1,  # Four-Vector = (1.,0.,1.,0.)
    'stochastic': False,
    'inspects_source': False,
    'manipulates_source': False,
    'manipulates_state': False
}
```

This helps classify the strategy as described in *Classification of strategies*.

After that the only thing required is to write the `strategy` method which takes an opponent as an argument. In the case of *TitForTat* the strategy checks if it has any history (`if len(self.history) == 0`). If it does not (ie this is the first play of the match) then it returns `C`. If not, the strategy simply repeats the opponent's last move (`return opponent.history[-1]`):

```
def strategy(opponent):
    """This is the actual strategy"""
    # First move
    if len(self.history) == 0:
        return C
    # Repeat the opponent's last move
    return opponent.history[-1]
```

The variables `C` and `D` represent the cooperate and defect actions respectively.

Some strategies make specific use of the variables of a match to create their own attributes. In principle these attributes could change throughout a match or tournament if the match properties (like the game matrix) change, so we require that this logic live in the `receive_match_attributes` method for correct dynamic updating. Here is how this is done for *Stalker*:

```
def receive_match_attributes(self)
    R, P, S, T = self.match_attributes["game"].RPST()
    self.very_good_score = R
    self.very_bad_score = P
    self.wish_score = (R + P) / 2
```

There are various examples of helpful functions and properties that make writing strategies easier. Do not hesitate to get in touch with the Axelrod-Python team for guidance.

Writing docstrings

The project takes pride in its documentation for the strategies and its corresponding bibliography. The docstring is a string which describes a method, module or class. The docstrings help the user in understanding the working of the strategy and the source of the strategy. The docstring must be written in the following way, i.e.:

```
 """This is a docstring.

It can be written over multiple lines.

"""
```

Sections

The Sections of the docstring are:

1. **Working of the strategy**

   A brief summary on how the strategy works, E.g.:

   ```
   class TitForTat(Player):
   """
   A player starts by cooperating and then mimics the
   previous action of the opponent.
   """
   ```

2. **Bibliography/Source of the strategy**

   A section to mention the source of the strategy or the paper from which the strategy was taken. The section must start with the Names section. For E.g.:

   ```
   class TitForTat(Player):
   """
   A player starts by cooperating and then mimics the
   previous action of the opponent.

   Names:

   - Rapoport's strategy: [Axelrod1980]_
   - TitForTat: [Axelrod1980]_
   """
   ```

   Here, the info written under the Names section tells about the source of the TitforTat strategy. `[Axelrod1980]_` corresponds to the bibliographic item in `docs/reference/bibliography.rst`. If you are using a source that is not in the bibliography please add it.

Adding the new strategy

To get the strategy to be recognised by the library we need to add it to the files that initialise when someone types `import axelrod`. This is done in the `axelrod/strategies/_strategies.py` file.

To classify the new strategy, run rebuild_classifier_table:

```
python rebuild_classifier_table.py
```

This will update `axelrod/strategies/_strategies.py`. Check that the recorded classifications for the strategies are what you expected.

If you have added your strategy to a file that already existed (perhaps you added a new variant of `titfortat` to the `titfortat.py` file), simply add your strategy to the list of strategies already imported from `<file_name>.py`:

```
from <file_name> import <list-of-strategies>
```

If you have added your strategy to a new file then simply add a line similar to above with your new strategy.

Once you have done that, you need to add the class itself to the `all_strategies` list (in `axelrod/strategies/_strategies.py`). You will also need to increment the doctest in `axelrod/docs/index.rst`.

Finally, if you have created a new module (a new `<strategy.py>` file) please add it to the `docs/references/all_strategies.rst` file so that it will automatically be documented.

Classifying the new strategy

Every strategy class has a classifier dictionary that gives some classification of the strategy according to certain dimensions. Some of the classifiers have formulas that try to compute the value for different strategies. Where these exist, they're overridden by the values defined in this dictionary. When creating a new strategy, you should try to fill out all of the dictionary.

Let us take a look at the dimensions available by looking at `TitForTat`:

```
>>> import axelrod
>>> classifier = axelrod.TitForTat.classifier
>>> for key in sorted(classifier.keys()):
...     print(key)
inspects_source
long_run_time
manipulates_source
manipulates_state
memory_depth
stochastic
```

You can read more about this in the *Classification of strategies* section but here are some tips about filling this part in correctly.

Note that when an instance of a class is created it gets it's own copy of the default classifier dictionary from the class. This might sometimes be modified by the initialisation depending on input parameters. A good example of this is the `Joss` strategy:

```
>>> joss = axelrod.FirstByJoss()
>>> boring_joss = axelrod.FirstByJoss(p=1)
>>> axelrod.Classifiers["stochastic"](joss)
True
>>> axelrod.Classifiers["stochastic"](boring_joss)
False
```

A classifier value defined on the instance overrides the value defined for the class.

There are currently three important dimensions that help identify if a strategy obeys axelrod's original tournament rules.

1. `inspects_source` - does the strategy 'read' any source code that it would not normally have access to. An example of this is `Geller`.
2. `manipulates_source` - does the strategy 'write' any source code that it would not normally be able to. An example of this is `Mind Bender`.
3. `manipulates_state` - does the strategy 'change' any attributes that it would not normally be able to. An example of this is `Mind Reader`.

These dimensions are currently relevant to the *obey_axelrod* function which checks if a strategy obeys Axelrod's original rules.

Writing tests for the new strategy

To write tests you either need to create a file called `test_<library>.py` where `<library>.py` is the name of the file you have created or similarly add tests to the test file that is already present in the `axelrod/tests/strategies/` directory.

Typically we want to test the following:

- That the strategy behaves as intended on the first move and subsequent moves, triggering any expected actions
- That the strategy initializes correctly

A `TestPlayer` class has been written that has a member function `versus_test` which can be used to test how the player plays against a given opponent. It takes an optional keyword argument `seed` (useful and necessary for stochastic strategies, `None` by default):

```
self.versus_test(opponent=axelrod.MockPlayer(actions=[C, D]),
                 expected_actions=[(D, C), (C, D), (C, C)], seed=None)
```

In this case the player is tested against an opponent that will cycle through `C, D`. The `expected_actions` are the actions played by both the tested player and the opponent in the match. In this case we see that the player is expected to play `D, C, C` against `C, D, C`.

Note that you can either user a `MockPlayer` that will cycle through a given sequence or you can use another strategy from the Axelrod library.

The function `versus_test` also accepts a dictionary parameter of attributes to check at the end of the match. For example this test checks if the player's internal variable `opponent_class` is set to `"Cooperative"`:

```
actions = [(C, C)] * 6
self.versus_test(axelrod.Cooperator(), expected_actions=actions
                 attrs={"opponent_class": "Cooperative"})
```

Note here that instead of passing a sequence of actions as an opponent we are passing an actual player from the axelrod library.

The function `versus_test` also accepts a dictionary parameter of match attributes that dictate the knowledge of the players. For example this test assumes that players do not know the length of the match:

```
actions = [(C, C), (C, D), (D, C), (C, D)]
self.versus_test(axelrod.Alternator(), expected_actions=actions,
                 match_attributes={"length": float("inf")})
```

The function `versus_test` also accepts a dictionary parameter of keyword arguments that dictate how the player is initiated. For example this tests how the player plays when initialised with `p=1`:

```
actions = [(C, C), (C, D), (C, C), (C, D)]
self.versus_test(axelrod.Alternator(), expected_actions=actions,
                 init_kwargs={"p": 1})
```

As an example, the tests for Tit-For-Tat are as follows:

```
import axelrod
from test_player import TestPlayer

```

(continues on next page)

(continued from previous page)

```
C, D = axelrod.Action.C, axelrod.Action.D

class TestTitForTat(TestPlayer):
    """
    Note that this test is referred to in the documentation as an example on
    writing tests.  If you modify the tests here please also modify the
    documentation.
    """

    name = "Tit For Tat"
    player = axelrod.TitForTat
    expected_classifier = {
        'memory_depth': 1,
        'stochastic': False,
        'makes_use_of': set(),
        'inspects_source': False,
        'manipulates_source': False,
        'manipulates_state': False
    }

    def test_strategy(self):
        self.first_play_test(C)
        self.second_play_test(rCC=C, rCD=D, rDC=C, rDD=D)

        # Play against opponents
        actions = [(C, C), (C, D), (D, C), (C, D)]
        self.versus_test(axelrod.Alternator(), expected_actions=actions)

        actions = [(C, C), (C, C), (C, C), (C, C)]
        self.versus_test(axelrod.Cooperator(), expected_actions=actions)

        actions = [(C, D), (D, D), (D, D), (D, D)]
        self.versus_test(axelrod.Defector(), expected_actions=actions)

        # This behaviour is independent of knowledge of the Match length
        actions = [(C, C), (C, D), (D, C), (C, D)]
        self.versus_test(axelrod.Alternator(), expected_actions=actions,
                         match_attributes={"length": float("inf")})

        # We can also test against random strategies
        actions = [(C, D), (D, D), (D, C), (C, C)]
        self.versus_test(axelrod.Random(), expected_actions=actions,
                         seed=0)

        actions = [(C, C), (C, D), (D, D), (D, C)]
        self.versus_test(axelrod.Random(), expected_actions=actions,
                         seed=1)

        #  If you would like to test against a sequence of moves you should use
        #  a MockPlayer
        opponent = axelrod.MockPlayer(actions=[C, D])
        actions = [(C, C), (C, D), (D, C), (C, D)]
        self.versus_test(opponent, expected_actions=actions)

        opponent = axelrod.MockPlayer(actions=[C, C, D, D, C, D])
        actions = [(C, C), (C, C), (C, D), (D, D), (D, C), (C, D)]
        self.versus_test(opponent, expected_actions=actions)
```

There are other examples of using this testing framework in `axelrod/tests/strategies/test_titfortat.py`.

The `expected_classifier` dictionary tests that the classification of the strategy is as expected (the tests for this is inherited in the `init` method). Please be sure to classify new strategies according to the already present dimensions but if you create a new dimension you do not **need** to re classify all the other strategies (but feel free to! :)), but please do add it to the `default_classifier` in the `axelrod/player.py` parent class.

Contributing to the library

All contributions (docs, tests, etc) are very welcome, if there is a specific functionality that you would like to add then please open an issue (or indeed take a look at the ones already there and jump in the conversation!).

If you want to work on documentation please keep in mind that doctests are encouraged to help keep the documentation up to date.

Running tests

Basic test runners

The project has an extensive test suite which is run each time a new contribution is made to the repository. If you want to check that all the tests pass before you submit a pull request you can run the tests yourself:

```
$ python -m unittest discover
```

If you are developing new tests for the suite, it is useful to run a single test file so that you don't have to wait for the entire suite each time. For example, to run only the tests for the Grudger strategy:

```
$ python -m unittest axelrod.tests.strategies.test_grudger
```

The test suite is divided into three categories: strategy tests, unit tests and integration tests. Each can be run individually:

```
$ python -m unittest discover -s axelrod.tests.strategies
$ python -m unittest discover -s axelrod.tests.unit
$ python -m unittest discover -s axelrod.tests.integration
```

Testing coverage of tests

The library has 100% test coverage. This can be tested using the Python `coverage` package. Once installed (`pip install coverage`), to run the tests and check the coverage for the entire library:

```
$ coverage run --source=axelrod -m unittest discover
```

You can then view a report of the coverage:

```
$ coverage report -m
```

You can also run the coverage on a subset of the tests. For example, to run the tests with coverage for the Grudger strategy:

```
$ coverage run --source=axelrod -m unittest axelrod.tests.strategies.test_grudger
```

Testing the documentation

The documentation is doctested, to run those tests you can run the script:

```
$ python doctests.py
```

You can also run the doctests on any given file. For example, to run the doctests for the `docs/tutorials/getting_started/match.rst` file:

```
$ python -m doctest docs/tutorials/getting_started/match.rst
```

Type checking

The library makes use of type hinting, this can be checked using the Python `mypy` package. Once installed (`pip install mypy`), to run the type checker:

```
$ python run_mypy.py
```

You can also run the type checker on a given file. For example, to run the type checker on the Grudger strategy:

```
$ mypy --ignore-missing-imports --follow-imports skip axelrod/strategies/grudger.py
```

2.3 Discussion

This section is a discussion of various aspects of Game Theory related to the Axelrod library.

Contents:

2.3.1 Background to Axelrod's Tournament

In the 1980s, professor of Political Science Robert Axelrod ran a tournament inviting strategies from collaborators all over the world for the Iterated Prisoner's Dilemma.

Another nice write up of Axelrod's work and this tournament on github was put together by Artem Kaznatcheev here.

The Prisoner's Dilemma

The Prisoner's dilemma is the simple two player game shown below:

	Cooperate	Defect
Cooperate	(3,3)	(0,5)
Defect	(5,0)	(1,1)

If both players cooperate they will each go to prison for 2 years and receive an equivalent utility of 3. If one cooperates and the other defects: the defector does not go to prison and the cooperator goes to prison for 5 years, the cooperator receives a utility of 0 and the defector a utility of 5. If both defect: they both go to prison for 4 years and receive an equivalent utility of 1.

Note: Years in prison doesn't equal to utility directly. The formula is U = 5 - Y for Y in [0, 5], where `U` is the utility, `Y` are years in prison. The reason is to follow the original Axelrod's scoring.

By simply investigating the best responses against both possible actions of each player it is immediate to see that the Nash equilibrium for this game is for both players to defect.

The Iterated Prisoner's Dilemma

We can use the basic Prisoner's Dilemma as a *stage* game in a repeated game. Players now aim to maximise the utility (corresponding to years in prison) over a repetition of the game. Strategies can take in to account both players history and so can take the form:

> "I will cooperate unless you defect 3 times in a row at which point I will defect forever."

Axelrod ran such a tournament (twice) and invited strategies from anyone who would contribute. The tournament was a round robin and the winner was the strategy who had the lowest total amount of time in prison.

This tournament has been used to study how cooperation can evolve from a very simple set of rules. This is mainly because the winner of both tournaments was 'tit for tat': a strategy that would never defect first (referred to as a 'nice' strategy).

2.3.2 Strategy Archetypes

Finite State Machines

A finite state machine (FSM) is a general computation model. In the context of Axelrod, it's a set of states and "transitions." A transition for a given state/previous-opponent-action combination says how the strategy will respond, both in what action it will take and in what state it will transitions to. That is a transition will specify that in state a, the strategy will respond to action X by taking action Y and moving to state b (which will tell us which transitions to use in later moves). We may write this transition (a, X, b, Y). For Axelrod, a FSM must have a full set of transitions, which specifies a unique response for each state/previous-opponent-action combination.

See [Harper2017] for a more-detailed explanation.

Representing a strategy as a finite state machine has been useful in some research (see [Harper2017] or [Ashlock2006b]). Though it's theoretically possible to represent all strategies as FSMs, this is impractical for most strategies. However, some strategies lend themselves naturally to a FSM representation. For example, for the Iterated Prisoner's Dilemma, we could consider a strategy that cooperates (C) until the opponent defects (D) twice in a row, then defect forever thereafter. (We'll call this strategy grudger_2 for the example.) We could call state 1, the state where the opponent hasn't started a defect streak; state 2, the state where the opponent is on a 1-defect streak; and state 3, the state where the opponent has defected twice in a row at some point. Then the transitions would be:

```
>>> from axelrod import Action
>>> C, D = Action.C, Action.D
>>> grudger_2_transitions = (
...     (1, C, 1, C),
...     (1, D, 2, C),
...     (2, C, 1, C),
...     (2, D, 3, D),
...     (3, C, 3, D),
...     (3, D, 3, D)
... )
```

The Axelrod library includes a FSM meta-strategy player, which will you let you specify a player's strategy by this transition matrix, along with an initial state and initial action. The syntax for this is:

```
>>> from axelrod.strategies.finite_state_machines import FSMPlayer
>>> grudger_2 = FSMPlayer(transitions=grudger_2_transitions,
...                       initial_state=1, initial_action=C)
```

The library also includes the functionality to compute the memory from the set of transitions. In the grudger_2 example, the memory would be 2. Because either the strategy's own previous move was a defect (in which case, continue to defect) or we just need to check if the last two opponent moves were defects or not. Though this function takes the transitions in a slightly different format:

```
>>> transition_dict = {
...     (t[0], t[1]): (t[2], t[3]) for t in grudger_2_transitions
... }
>>> from axelrod.compute_finite_state_machine_memory import *
>>> get_memory_from_transitions(transitions=transition_dict,
...                             initial_state=1)
2
```

2.3.3 Overview of past Tournaments

Axelrod's first tournament

Axelrod's first tournament is described in his 1980 paper entitled 'Effective choice in the Prisoner's Dilemma' [Axelrod1980]. This tournament included 14 strategies (plus a random "strategy") and they are listed below, (ranked in the order in which they appeared).

An indication is given as to whether or not this strategy is implemented in the `axelrod` library. If this strategy is not implemented please do send us a pull request.

Table 1: Strategies in Axelrod's first tournament

Name	Author	Axelrod Library Name
Tit For Tat	Anatol Rapoport	*TitForTat*
Tideman and Chieruzzi	T Nicolaus Tideman and Paula Chieruzz	*TidemanAndChieruzzi*
Nydegger	Rudy Nydegger	*Nydegger*
Grofman	Bernard Grofman	*Grofman*
Shubik	Martin Shubik	*Shubik*
Stein and Rapoport	Stein and Anatol Rapoport	*SteinAndRapoport*
Grudger	James W Friedman	*Grudger*
Davis	Morton Davis	*Davis*
Graaskamp	Jim Graaskamp	*Graaskamp*
FirstByDowning	Leslie Downing	*RevisedDowning*
Feld	Scott Feld	*Feld*
Joss	Johann Joss	*Joss*
Tullock	Gordon Tullock	*Tullock*
(Name withheld)	Unknown	UnnamedStrategy
Random	Unknownd	*Random*

Axelrod's second tournament

The code for Axelrod's second touranment was originally published by the University of Michigan Center for the Study of Complex Systems and is now available from Robert Axelrod's personal website subject to a disclaimer which states:

> "All materials in this archive are copyright (c) 1996, Robert Axelrod, unless otherwise noted. You are free to download these materials and use them without restriction."

The Axelrod-Python organisation has published a modified version of the original code. In the following table, links to original code point to the Axelrod-Python repository.

Table 2: Strategies in Axelrod's second tournament

Original Code	Author	Axelrod Library Name
GRASR	Unknown	Not Implemented
K31R	Gail Grisell	*GoByMajority*
K32R	Charles Kluepfel	*SecondByKluepfel*
K33R	Harold Rabbie	Not Implemented
K34R	James W Friedman	*Grudger*
K35R	Abraham Getzler	Not Implemented
K36R	Roger Hotz	Not Implemented
K37R	George Lefevre	Not Implemented
K38R	Nelson Weiderman	Not Implemented
K39R	Tom Almy	Not Implemented
K40R	Robert Adams	Not Implemented
K41R	Herb Weiner	*SecondByWeiner*
K42R	Otto Borufsen	*SecondByBorufsen*
K43R	R D Anderson	Not Implemented
K44R	William Adams	*SecondByWmAdams*
K45R	Michael F McGurrin	Not Implemented
K46R	Graham J Eatherley	*SecondByEatherley*
K47R	Richard Hufford	*SecondByRichardHufford*
K48R	George Hufford	Not Implemented
K49R	Rob Cave	*SecondByCave*
K50R	Rik Smoody	Not Implemented
K51R	John Willaim Colbert	Not Implemented
K52R	David A Smith	Not Implemented
K53R	Henry Nussbacher	Not Implemented
K54R	William H Robertson	Not Implemented
K55R	Steve Newman	Not Implemented
K56R	Stanley F Quayle	Not Implemented
K57R	Rudy Nydegger	Not Implemented
K58R	Glen Rowsam	*SecondByRowsam*
K59R	Leslie Downing	*RevisedDowning*
K60R	Jim Graaskamp and Ken Katzen	*SecondByGraaskampKatzen*
K61R	Danny C Champion	*SecondByChampion*
K62R	Howard R Hollander	Not Implemented
K63R	George Duisman	Not Implemented
K64R	Brian Yamachi	*SecondByYamachi*
K65R	Mark F Batell	Not Implemented
K66R	Ray Mikkelson	Not Implemented
K67R	Craig Feathers	*SecondByTranquilizer*

Continued on next page

Table 2 – continued from previous page

Original Code	Author	Axelrod Library Name
K68R	Fransois Leyvraz	`SecondByLeyvraz`
K69R	Johann Joss	Not Implemented
K70R	Robert Pebly	Not Implemented
K71R	James E Hall	Not Implemented
K72R	Edward C White Jr	`SecondByWhite`
K73R	George Zimmerman	Not Implemented
K74R	Edward Friedland	Not Implemented
K74RXX	Edward Friedland	Not Implemented
K75R	Paul D Harrington	`SecondByHarrington`
K76R	David Gladstein	`SecondByGladstein`
K77R	Scott Feld	Not Implemented
K78R	Fred Mauk	Not Implemented
K79R	Dennis Ambuehl and Kevin Hickey	Not Implemented
K80R	Robyn M Dawes and Mark Batell	Not Implemented
K81R	Martyn Jones	Not Implemented
K82R	Robert A Leyland	Not Implemented
K83R	Paul E Black	`SecondByWhite`
K84R	T Nicolaus Tideman and Paula Chieruzzi	`SecondByTidemanChieruzzi`
K85R	Robert B Falk and James M Langsted	Not Implemented
K86R	Bernard Grofman	Not Implemented
K87R	E E H Schurmann	Not Implemented
K88R	Scott Appold	`SecondByAppold`
K89R	Gene Snodgrass	Not Implemented
K90R	John Maynard Smith	`TitFor2Tats`
K91R	Jonathan Pinkley	Not Implemented
K92R	Anatol Rapoport	`TitForTat`
K93R	Unknown	Not Implemented
KPAVLOVC	Unknown	`WinStayLoseShift`
KRANDOMC	Unknown	`Random`
KTF2TC	Unknown	`TitFor2Tats`
KTITFORTATC	Unknown	`TitForTat`

Stewart and Plotkin's Tournament (2012)

In 2012, Alexander Stewart and Joshua Plotkin ran a variant of Axelrod's tournament with 19 strategies to test the effectiveness of the then newly discovered Zero-Determinant strategies.

The paper is identified as *doi: 10.1073/pnas.1208087109* and referred to as [Stewart2012] below. Unfortunately the details of the tournament and the implementation of strategies is not clear in the manuscript. We can, however, make reasonable guesses to the implementation of many strategies based on their names and classical definitions.

The following classical strategies are included in the library:

Table 3: Strategies in Stewart and Plotkin's tournament

S&P Name	Long Name	Axelrod Library Name
ALLC	Always Cooperate	*Cooperator*
ALLD	Always Defect	*Defector*
EXTORT-2	Extort-2	ZDExtort2
HARD_MAJO	Hard majority	*HardGoByMajority*
HARD_JOSS	Hard Joss	Joss
HARD_TFT	Hard tit for tat	*HardTitForTat*
HARD_TF2T	Hard tit for 2 tats	*HardTitFor2Tats*
TFT	Tit-For-Tat	*TitForTat*
GRIM	Grim	*Grudger*
GTFT	Generous Tit-For-Tat	*GTFT*
TF2T	Tit-For-Two-Tats	*TitFor2Tats*
WSLS	Win-Stay-Lose-Shift	*WinStayLoseShift*
RANDOM	Random	*Random*
ZDGTFT-2	ZDGTFT-2	ZDGTFT2

ALLC, ALLD, TFT and RANDOM are defined above. The remaining classical strategies are defined below. The tournament also included two Zero Determinant strategies, both implemented in the library. The full table of strategies and results is available online.

Memory one strategies

In 2012 Press and Dyson [Press2012] showed interesting results with regards to so called memory one strategies. Stewart and Plotkin implemented a number of these. A memory one strategy is simply a probabilistic strategy that is defined by 4 parameters. These four parameters dictate the probability of cooperating given 1 of 4 possible outcomes of the previous round:

- $P(C \mid CC) = p_1$
- $P(C \mid CD) = p_2$
- $P(C \mid DC) = p_3$
- $P(C \mid DD) = p_4$

The memory one strategy class is used to define a number of strategies below.

GTFT

Generous-Tit-For-Tat plays Tit-For-Tat with occasional forgiveness, which prevents cycling defections against itself.

GTFT is defined as a memory-one strategy as follows:

- $P(C \mid CC) = 1$
- $P(C \mid CD) = p$
- $P(C \mid DC) = 1$
- $P(C \mid DD) = p$

where $p = \min\left(1 - \frac{T-R}{R-S}, \frac{R-P}{T-P}\right)$.

GTFT came 2nd in average score and 18th in wins in S&P's tournament.

TF2T

Tit-For-Two-Tats is like Tit-For-Tat but only retaliates after two defections rather than one. This is not a memory-one strategy.

TF2T came 3rd in average score and last (?) in wins in S&P's tournament.

WSLS

Win-Stay-Lose-Shift is a strategy that shifts if the highest payoff was not earned in the previous round. WSLS is also known as "Win-Stay-Lose-Switch" and "Pavlov". It can be seen as a memory-one strategy as follows:

- $P(C \mid CC) = 1$
- $P(C \mid CD) = 0$
- $P(C \mid DC) = 0$
- $P(C \mid DD) = 1$

WSLS came 7th in average score and 13th in wins in S&P's tournament.

RANDOM

Random is a strategy that was defined in *Axelrod's first tournament*, note that this is also a memory-one strategy:

- $P(C \mid CC) = 0.5$
- $P(C \mid CD) = 0.5$
- $P(C \mid DC) = 0.5$
- $P(C \mid DD) = 0.5$

RANDOM came 8th in average score and 8th in wins in S&P's tournament.

ZDGTFT-2

This memory-one strategy is defined by the following four conditional probabilities based on the last round of play:

- $P(C \mid CC) = 1$
- $P(C \mid CD) = 1/8$
- $P(C \mid DC) = 1$
- $P(C \mid DD) = 1/4$

This strategy came 1st in average score and 16th in wins in S&P's tournament.

EXTORT-2

This memory-one strategy is defined by the following four conditional probabilities based on the last round of play:

- $P(C \mid CC) = 8/9$
- $P(C \mid CD) = 1/2$
- $P(C \mid DC) = 1/3$

- $P(C \mid DD) = 0$

This strategy came 18th in average score and 2nd in wins in S&P's tournament.

GRIM

Grim is not defined in [Stewart2012] but it is defined elsewhere as follows. GRIM (also called "Grim trigger"), cooperates until the opponent defects and then always defects thereafter. In the library this strategy is called *Grudger*.

GRIM came 10th in average score and 11th in wins in S&P's tournament.

HARD_JOSS

HARD_JOSS is not defined in [Stewart2012] but is otherwise defined as a strategy that plays like TitForTat but cooperates only with probability 0.9. This is a memory-one strategy with the following probabilities:

- $P(C \mid CC) = 0.9$
- $P(C \mid CD) = 0$
- $P(C \mid DC) = 1$
- $P(C \mid DD) = 0$

HARD_JOSS came 16th in average score and 4th in wins in S&P's tournament.

HARD_JOSS as described above is implemented in the library as *Joss* and is the same as the Joss strategy from *Axelrod's first tournament*.

HARD_MAJO

HARD_MAJO is not defined in [Stewart2012] and is presumably the same as "Go by Majority", defined as follows: the strategy defects on the first move, defects if the number of defections of the opponent is greater than or equal to the number of times it has cooperated, and otherwise cooperates,

HARD_MAJO came 13th in average score and 5th in wins in S&P's tournament.

HARD_TFT

Hard TFT is not defined in [Stewart2012] but is elsewhere defined as follows. The strategy cooperates on the first move, defects if the opponent has defected on any of the previous three rounds, and otherwise cooperates.

HARD_TFT came 12th in average score and 10th in wins in S&P's tournament.

HARD_TF2T

Hard TF2T is not defined in [Stewart2012] but is elsewhere defined as follows. The strategy cooperates on the first move, defects if the opponent has defected twice (successively) of the previous three rounds, and otherwise cooperates.

HARD_TF2T came 6th in average score and 17th in wins in S&P's tournament.

Calculator

This strategy is not unambiguously defined in [Stewart2012] but is defined elsewhere. Calculator plays like Joss for 20 rounds. On the 21 round, Calculator attempts to detect a cycle in the opponents history, and defects unconditionally thereafter if a cycle is found. Otherwise Calculator plays like TFT for the remaining rounds.

Prober

PROBE is not unambiguously defined in [Stewart2012] but is defined elsewhere as Prober. The strategy starts by playing D, C, C on the first three rounds and then defects forever if the opponent cooperates on rounds two and three. Otherwise Prober plays as TitForTat would.

Prober came 15th in average score and 9th in wins in S&P's tournament.

Prober2

PROBE2 is not unambiguously defined in [Stewart2012] but is defined elsewhere as Prober2. The strategy starts by playing D, C, C on the first three rounds and then cooperates forever if the opponent played D then C on rounds two and three. Otherwise Prober2 plays as TitForTat would.

Prober2 came 9th in average score and 12th in wins in S&P's tournament.

Prober3

PROBE3 is not unambiguously defined in [Stewart2012] but is defined elsewhere as Prober3. The strategy starts by playing D, C on the first two rounds and then defects forever if the opponent cooperated on round two. Otherwise Prober3 plays as TitForTat would.

Prober3 came 17th in average score and 7th in wins in S&P's tournament.

HardProber

HARD_PROBE is not unambiguously defined in [Stewart2012] but is defined elsewhere as HardProber. The strategy starts by playing D, D, C, C on the first four rounds and then defects forever if the opponent cooperates on rounds two and three. Otherwise Prober plays as TitForTat would.

HardProber came 5th in average score and 6th in wins in S&P's tournament.

NaiveProber

NAIVE_PROBER is a modification of Tit For Tat strategy which with a small probability randomly defects. Default value for a probability of defection is 0.1.

Beaufils et al.'s tournament (1997)

In 1997, [Beaufils1997] the authors used a tournament to describe a new strategy of their called "Gradual". The description given in the paper of "Gradual" is:

> This strategy acts as tit-for-tat, except when it is time to forgive and remember the past. It uses cooperation on the first move and then continues to do so as long as the other player cooperates. Then after the first defection of the other player, it defects one time and cooperates two times; after the second defection of the opponent, it defects two times and cooperates two times, ... after the nth defection it reacts with n consecutive defections and then calms down its opponent with two cooperations.

This is the only description of the strategy however the paper does include a table of results of the tournament. The scores of "Gradual" against the opponents (including itself) are:

Table 4: Score of Gradual reported in [Beaufils1997]

Name	Name used in [Beaufils1997]	Score (1000 turns)
Cooperator	coop	3000
Defector	def	915
Random	rand	2815
Tit For Tat	tft	3000
Grudger	spite	3000
Cycler DDC	p_nst	2219
Cycler CCD	p_kn	3472
Go By Majority	sft_mj	3000
Suspicious Tit For Tat	mist	2996
Prober	prob	2999
Gradual	grad	3000
Win Stay Lose Shift	pav	3000

The following code reproduces the above:

```
>>> import axelrod as axl
>>> players = [axl.Cooperator(),
...            axl.Defector(),
...            axl.Random(),
...            axl.TitForTat(),
...            axl.Grudger(),
...            axl.CyclerDDC(),
...            axl.CyclerCCD(),
...            axl.GoByMajority(),
...            axl.SuspiciousTitForTat(),
...            axl.Prober(),
...            axl.OriginalGradual(),
...            axl.WinStayLoseShift(),
...            ]
>>> turns = 1000
>>> tournament = axl.Tournament(players, turns=turns, repetitions=1, seed=75)
>>> results = tournament.play(progress_bar=False)
>>> for average_score_per_turn in results.payoff_matrix[-2]:
...     print(round(average_score_per_turn * turns, 1))
3000.0
915.0
2763.0
3000.0
3000.0
2219.0
3472.0
3000.0
2996.0
2999.0
```

(continues on next page)

(continued from previous page)

```
3000.0
3000.0
```

The `OriginalGradual` strategy implemented has the following description:

> A player that punishes defections with a growing number of defections but after punishing for *punishment_limit* number of times enters a calming state and cooperates no matter what the opponent does for two rounds.
>
> The *punishment_limit* is incremented whenever the opponent defects and the strategy is not in either calming or punishing state.

Note that a different version of Gradual appears in [CRISTAL-SMAC2018]. This was brought to the attention of the maintainers of the library by one of the authors of [Beaufils1997] and is documented here https://github.com/Axelrod-Python/Axelrod/issues/1294.

The strategy implemented in [CRISTAL-SMAC2018] and defined here as `Gradual` has the following description:

> Similar to OriginalGradual, this is a player that punishes defections with a growing number of defections but after punishing for *punishment_limit* number of times enters a calming state and cooperates no matter what the opponent does for two rounds.
>
> This version of Gradual is an update of *OriginalGradual* and the difference is that the *punishment_limit* is incremented whenever the opponent defects (regardless of the state of the player).

This highlights the importance of best practice and reproducible computational research. Both strategies implemented in this library are fully tested and documented clearly and precisely.

2.3.4 Play Contexts and Generic Prisoner's Dilemma

There are four possible round outcomes:

- Mutual cooperation: (C, C)
- Defection: (C, D) or (D, C)
- Mutual defection: (D, D)

Each of these corresponds to one particular set of payoffs in the following generic Prisoner's dilemma:

	Cooperate	Defect
Cooperate	(R,R)	(S,T)
Defect	(T,S)	(P,P)

For the above to constitute a Prisoner's dilemma, the following must hold: $T > R > P > S$.

These payoffs are commonly referred to as:

- R: the **Reward** payoff (default value in the library: 3)
- P: the **Punishment** payoff (default value in the library: 1)
- S: the **Sucker** payoff (default value in the library: 0)
- T: the **Temptation** payoff (default value in the library: 5)

A particular Prisoner's Dilemma is often described by the 4-tuple: (R, P, S, T):

```
>>> import axelrod
>>> axelrod.game.DefaultGame.RPST()
(3, 1, 0, 5)
```

2.3.5 Community

Contents:

Part of the team

If you're reading this you're probably interested in contributing to and/or using the Axelrod library! Firstly: **thank you** and **welcome**!

We are proud of the library and the environment that surrounds it. A primary goal of the project is to cultivate an open and welcoming community, considerate and respectful to newcomers to python and game theory.

The Axelrod library has been a first contribution to open source software for many, and this is in large part due to the fact that we all aim to help and encourage all levels of contribution. If you're a beginner, that's awesome! You're very welcome and don't hesitate to ask for help.

With regards to any contribution, please do not feel the need to wait until your contribution is perfectly polished and complete: we're happy to offer early feedback, help with git, and anything else that you need to have a positive experience.

If you are using the library for your own work and there's anything in the documentation that is unclear: we want to know so that we can fix it. We also want to help so please don't hesitate to get in touch.

Communication

There are various ways of communicating with the team:

- Gitter: a web based chat client, you can talk directly to the users and maintainers of the library.
- Irc: we have an irc channel. It's #axelrod-python on freenode.
- Email forum.
- Issues: you are also very welcome to open an issue on github
- Twitter. This account periodically tweets out random match and tournament results; you're welcome to get in touch through twitter as well.

Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting a member of the core team. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at http://contributor-covenant.org/version/1/4

2.4 Reference

This section is the reference guide for the various components of the library.

2.4.1 Bibliography

This is a collection of various bibliographic items referenced in the documentation.

2.4.2 Strategy index

Here are the docstrings of all the strategies in the library.

class axelrod.strategies.adaptive.**Adaptive**(*initial_plays: List[axelrod.action.Action] = None*)

Start with a specific sequence of C and D, then play the strategy that has worked best, recalculated each turn.

Names:

- Adaptive: [Li2011]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action

Actual strategy definition that determines player's action.

class axelrod.strategies.adaptor.**AbstractAdaptor**(*delta: Dict[Tuple[axelrod.action.Action, axelrod.action.Action], float], perr: float = 0.01*)

An adaptive strategy that updates an internal state based on the last round of play. Using this state the player Cooperates with a probability derived from the state.

s, float: the internal state, initially 0

perr, float: an error threshold for misinterpreted moves

delta, a dictionary of floats: additive update values for s depending on the last round's outcome

Names:

- Adaptor: [Hauert2002]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action

Actual strategy definition that determines player's action.

class axelrod.strategies.adaptor.**AdaptorBrief**

An Adaptor trained on short interactions.

Names:

- AdaptorBrief: [Hauert2002]

class axelrod.strategies.adaptor.**AdaptorLong**

An Adaptor trained on long interactions.

Names:

- AdaptorLong: [Hauert2002]

class axelrod.strategies.alternator.**Alternator**

A player who alternates between cooperating and defecting.

Names

- Alternator: [Axelrod1984]
- Periodic player CD: [Mittal2009]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.ann.**ANN**(*num_features: int*, *num_hidden: int*, *weights: List[float] = None*)
Artificial Neural Network based strategy.

A single layer neural network based strategy, with the following features: * Opponent's first move is C * Opponent's first move is D * Opponent's second move is C * Opponent's second move is D * Player's previous move is C * Player's previous move is D * Player's second previous move is C * Player's second previous move is D * Opponent's previous move is C * Opponent's previous move is D * Opponent's second previous move is C * Opponent's second previous move is D * Total opponent cooperations * Total opponent defections * Total player cooperations * Total player defections * Round number

Original Source: https://gist.github.com/mojones/550b32c46a8169bb3cd89d917b73111a#file-ann-strategy-test-L60

Names

- Artificial Neural Network based strategy: Original name by Martin Jones

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.ann.**EvolvableANN**(*num_features: int*, *num_hidden: int*, *weights: List[float] = None*, *mutation_probability: float = None*, *mutation_distance: int = 5*, *seed: int = None*)
Evolvable version of ANN.

crossover(*other*)
Optional method to allow Player to produce variants in combination with another player. Returns a new Player.

mutate()
Optional method to allow Player to produce a variant (not in place).

class axelrod.strategies.ann.**EvolvedANN**
A strategy based on a pre-trained neural network with 17 features and a hidden layer of size 10.

Trained using the *axelrod_dojo* version: 0.0.8 Training data is archived at doi.org/10.5281/zenodo.1306926

Names:

- Evolved ANN: Original name by Martin Jones.

class axelrod.strategies.ann.**EvolvedANN5**
A strategy based on a pre-trained neural network with 17 features and a hidden layer of size 5.

Trained using the *axelrod_dojo* version: 0.0.8 Training data is archived at doi.org/10.5281/zenodo.1306931

Names:

- Evolved ANN 5: Original name by Marc Harper.

class axelrod.strategies.ann.**EvolvedANNNoise05**
A strategy based on a pre-trained neural network with a hidden layer of size 5, trained with noise=0.05.

Trained using the *axelrod_dojo* version: 0.0.8 Training data i archived at doi.org/10.5281/zenodo.1314247.

Names:

- Evolved ANN Noise 5: Original name by Marc Harper.

axelrod.strategies.ann.**activate**(*bias: List[float], hidden: List[float], output: List[float], inputs: numpy.ndarray*) → float

Compute the output of the neural network: output = relu(inputs * hidden_weights + bias) * output_weights

axelrod.strategies.ann.**compute_features**(*player: axelrod.player.Player*, *opponent: axelrod.player.Player*) → numpy.ndarray

Compute history features for Neural Network: * Opponent's first move is C * Opponent's first move is D * Opponent's second move is C * Opponent's second move is D * Player's previous move is C * Player's previous move is D * Player's second previous move is C * Player's second previous move is D * Opponent's previous move is C * Opponent's previous move is D * Opponent's second previous move is C * Opponent's second previous move is D * Total opponent cooperations * Total opponent defections * Total player cooperations * Total player defections * Round number

axelrod.strategies.ann.**split_weights**(*weights: List[float], num_features: int, num_hidden: int*) → Tuple[List[List[float]], List[float], List[float]]

Splits the input vector into the the NN bias weights and layer parameters.

class axelrod.strategies.apavlov.**APavlov2006**

APavlov attempts to classify its opponent as one of five strategies: Cooperative, ALLD, STFT, PavlovD, or Random. APavlov then responds in a manner intended to achieve mutual cooperation or to defect against uncooperative opponents.

Names:

- Adaptive Pavlov 2006: [Li2007]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action

Actual strategy definition that determines player's action.

class axelrod.strategies.apavlov.**APavlov2011**

APavlov attempts to classify its opponent as one of four strategies: Cooperative, ALLD, STFT, or Random. APavlov then responds in a manner intended to achieve mutual cooperation or to defect against uncooperative opponents.

Names:

- Adaptive Pavlov 2011: [Li2011]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action

Actual strategy definition that determines player's action.

class axelrod.strategies.appeaser.**Appeaser**

A player who tries to guess what the opponent wants.

Switch the classifier every time the opponent plays D. Start with C, switch between C and D when opponent plays D.

Names:

- Appeaser: Original Name by Jochen Müller

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action

Actual strategy definition that determines player's action.

class axelrod.strategies.averagecopier.**AverageCopier**

The player will cooperate with probability p if the opponent's cooperation ratio is p. Starts with random decision.

Names:

- Average Copier: Original name by Geraint Palmer

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.averagecopier.**NiceAverageCopier**
Same as Average Copier, but always starts by cooperating.

Names:

- Average Copier: Original name by Owen Campbell

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

Strategies submitted to Axelrod's first tournament. All strategies in this module are prefixed by *FirstBy* to indicate that they were submitted in Axelrod's First tournament by the given author.

Note that these strategies are implemented from the descriptions presented in:

Axelrod, R. (1980). Effective Choice in the Prisoner's Dilemma. Journal of Conflict Resolution, 24(1), 3–25.

These descriptions are not always clear and/or precise and when assumptions have been made they are explained in the strategy docstrings.

class axelrod.strategies.axelrod_first.**FirstByAnonymous**
Submitted to Axelrod's first tournament by a graduate student whose name was withheld.

The description written in [Axelrod1980] is:

> "This rule has a probability of cooperating, P, which is initially 30% and > is updated every 10 moves. P is adjusted if the other player seems random, > very cooperative, or very uncooperative. P is also adjusted after move 130 > if the rule has a lower score than the other player. Unfortunately, the > complex process of adjustment frequently left the probability of cooperation > in the 30% to 70% range, and therefore the rule appeared random to many > other players."

Given the lack of detail this strategy is implemented based on the final sentence of the description which is to have a cooperation probability that is uniformly random in the 30 to 70% range.

Names:

- (Name withheld): [Axelrod1980]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_first.**FirstByDavis**(*rounds_to_cooperate: int = 10*)
Submitted to Axelrod's first tournament by Morton Davis.

The description written in [Axelrod1980] is:

> "A player starts by cooperating for 10 rounds then plays Grudger, > defecting if at any point the opponent has defected."

This strategy came 8th in Axelrod's original tournament.

Names:

- Davis: [Axelrod1980]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Begins by playing C, then plays D for the remaining rounds if the opponent ever plays D.

class axelrod.strategies.axelrod_first.**FirstByDowning**
Submitted to Axelrod's first tournament by Downing

The description written in [Axelrod1980] is:

> "This rule selects its choice to maximize its own longterm expected payoff on > the assumption that the other rule cooperates with a fixed probability which > depends only on whether the other player cooperated or defected on the previous > move. These two probabilities estimates are continuously updated as the game > progresses. Initially, they are both assumed to be .5, which amounts to the > pessimistic assumption that the other player is not responsive. This rule is > based on an outcome maximization interpretation of human performances proposed > by Downing (1975)."

The Downing (1975) paper is "The Prisoner's Dilemma Game as a Problem-Solving Phenomenon" [Downing1975] and this is used to implement the strategy.

There are a number of specific points in this paper, on page 371:

> "[...] In these strategies, O's [the opponent's] response on trial N is in some way dependent or contingent on S's [the subject's] response on trial N- 1. All varieties of these lag-one matching strategies can be defined by two parameters: the conditional probability that O will choose C following C by S, P(C_o | C_s) and the conditional probability that O will choose C following D by S, P(C_o, D_s)."

Throughout the paper the strategy (S) assumes that the opponent (O) is playing a reactive strategy defined by these two conditional probabilities.

The strategy aims to maximise the long run utility against such a strategy and the mechanism for this is described in Appendix A (more on this later).

One final point from the main text is, on page 372:

> "For the various lag-one matching strategies of O, the maximizing strategies of S will be 100% C, or 100% D, or for some strategies all S strategies will be functionally equivalent."

This implies that the strategy S will either always cooperate or always defect (or be indifferent) dependent on the opponent's defining probabilities.

To understand the particular mechanism that describes the strategy S, we refer to Appendix A of the paper on page 389.

The stated goal of the strategy is to maximize (using the notation of the paper):

EV_TOT = #CC(EV_CC) + #CD(EV_CD) + #DC(EV_DC) + #DD(EV_DD)

This differs from the more modern literature where #CC, #CD, #DC and #DD would imply that counts of both players playing C and C, or the first playing C and the second D etc... In this case the author uses an argument based on the sequence of plays by the player (S) so #CC denotes the number of times the player plays C twice in a row.

On the second page of the appendix, figure 4 (page 390) identifies an expression for EV_TOT. A specific term is made to disappear in the case of T - R = P - S (which is not the case for the standard (R, P, S, T) = (3, 1, 0, 5)):

> "Where (t - r) = (p - s), EV_TOT will be a function of alpha, beta, t, r, p, s and N are known and V which is unknown.

V is the total number of cooperations of the player S (this is noted earlier in the abstract) and as such the final expression (with only V as unknown) can be used to decide if V should indicate that S always cooperates or not.

This final expression is used to show that EV_TOT is linear in the number of cooperations by the player thus justifying the fact that the player will always cooperate or defect.

All of the above details are used to give the following interpretation of the strategy:

1. On any given turn, the strategy will estimate alpha = P(C_o | C_s) and beta = P(C_o | D_s). 2. The strategy will calculate the expected utility of always playing C OR always playing D against the estimated probabilities. This corresponds to:

a. In the case of the player always cooperating:

   P_CC = alpha and P_CD = 1 - alpha

b. In the case of the player always defecting:

P_DC = beta and P_DD = 1 - beta

Using this we have:

E_C = alpha R + (1 - alpha) S E_D = beta T + (1 - beta) P

Thus at every turn, the strategy will calculate those two values and cooperate if E_C > E_D and will defect if E_C < E_D.

In the case of E_C = E_D, the player will alternate from their previous move. This is based on specific sentence from Axelrod's original paper:

> "Under certain circumstances, DOWNING will even determine that the best > strategy is to alternate cooperation and defection."

One final important point is the early game behaviour of the strategy. It has been noted that this strategy was implemented in a way that assumed that alpha and beta were both 1/2:

> "Initially, they are both assumed to be .5, which amounts to the > pessimistic assumption that the other player is not responsive."

Note that if alpha = beta = 1 / 2 then:

E_C = alpha R + alpha S E_D = alpha T + alpha P

And from the defining properties of the Prisoner's Dilemma (T > R > P > S) this gives: E_D > E_C. Thus, the player opens with a defection in the first two rounds. Note that from the Axelrod publications alone there is nothing to indicate defections on the first two rounds, although a defection in the opening round is clear. However there is a presentation available at http://www.sci.brooklyn.cuny.edu/~sklar/teaching/f05/alife/notes/azhar-ipd-Oct19th.pdf That clearly states that Downing defected in the first two rounds, thus this is assumed to be the behaviour. Interestingly, in future tournaments this strategy was revised to not defect on the opening two rounds.

It is assumed that these first two rounds are used to create initial estimates of beta = P(C_o | D_s) and we will use the opening play of the player to estimate alpha = P(C_o | C_s). Thus we assume that the opponents first play is a response to a cooperation "before the match starts".

So for example, if the plays are:

[(D, C), (D, C)]

Then the opponent's first cooperation counts as a cooperation in response to the non existent cooperation of round 0. The total number of cooperations in response to a cooperation is 1. We need to take in to account that extra phantom cooperation to estimate the probability alpha=P(C_o | C_s) as 1 / 1 = 1.

This is an assumption with no clear indication from the literature.

– This strategy came 10th in Axelrod's original tournament.

Names:

- Downing: [Axelrod1980]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_first.**FirstByFeld**(*start_coop_prob: float = 1.0*, *end_coop_prob: float = 0.5*, *rounds_of_decay: int = 200*)

Submitted to Axelrod's first tournament by Scott Feld.

The description written in [Axelrod1980] is:

> "This rule starts with tit for tat and gradually lowers its probability of > cooperation following the other's cooperation to .5 by the two hundredth > move. It always defects after a defection by the other."

This strategy plays Tit For Tat, always defecting if the opponent defects but cooperating when the opponent cooperates with a gradually decreasing probability until it is only .5. Note that the description does not clearly indicate how the cooperation probability should drop. This implements a linear decreasing function.

This strategy came 11th in Axelrod's original tournament.

Names:

- Feld: [Axelrod1980]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_first.**FirstByGraaskamp**(*alpha: float = 0.05*)
Submitted to Axelrod's first tournament by James Graaskamp.

The description written in [Axelrod1980] is:

> "This rule plays tit for tat for 50 moves, defects on move 51, and then > plays 5 more moves of tit for tat. A check is then made to see if the player > seems to be RANDOM, in which case it defects from then on. A check is also > made to see if the other is TIT FOR TAT, ANALOGY (a program from the > preliminary tournament), and its own twin, in which case it plays tit for > tat. Otherwise it randomly defects every 5 to 15 moves, hoping that enough > trust has been built up so that the other player will not notice these > defections.:

This is implemented as:

1. Plays Tit For Tat for the first 50 rounds;
2. Defects on round 51;
3. Plays 5 further rounds of Tit For Tat;
4. A check is then made to see if the opponent is playing randomly in which case it defects for the rest of the game. This is implemented with a chi squared test.
5. The strategy also checks to see if the opponent is playing Tit For Tat or a clone of itself. If so it plays Tit For Tat. If not it cooperates and randomly defects every 5 to 15 moves.

Note that there is no information about 'Analogy' available thus Step 5 is a "best possible" interpretation of the description in the paper. Furthermore the test for the clone is implemented as checking that both players have played the same moves for the entire game. This is unlikely to be the original approach but no further details are available.

This strategy came 9th in Axelrod's original tournament.

Names:

- Graaskamp: [Axelrod1980]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is the actual strategy

class axelrod.strategies.axelrod_first.**FirstByGrofman**
Submitted to Axelrod's first tournament by Bernard Grofman.

The description written in [Axelrod1980] is:

> "If the players did different things on the previous move, this rule > cooperates with probability 2/7. Otherwise this rule always cooperates."

This strategy came 4th in Axelrod's original tournament.

Names:

- Grofman: [Axelrod1980]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_first.**FirstByJoss**(*p: float = 0.9*)
Submitted to Axelrod's first tournament by Johann Joss.

The description written in [Axelrod1980] is:

> "This rule cooperates 90% of the time after a cooperation by the other. It > always defects after a defection by the other."

This strategy came 12th in Axelrod's original tournament.

Names:

- Joss: [Axelrod1980]
- Hard Joss: [Stewart2012]

class axelrod.strategies.axelrod_first.**FirstByNydegger**
Submitted to Axelrod's first tournament by Rudy Nydegger.

The description written in [Axelrod1980] is:

> "The program begins with tit for tat for the first three moves, except > that if it was the only one to cooperate on the first move and the only one > to defect on the second move, it defects on the third move. After the third > move, its choice is determined from the 3 preceding outcomes in the > following manner. Let A be the sum formed by counting the other's defection > as 2 points and one's own as 1 point, and giving weights of 16, 4, and 1 to > the preceding three moves in chronological order. The choice can be > described as defecting only when A equals 1, 6, 7, 17, 22, 23, 26, 29, 30, > 31, 33, 38, 39, 45, 49, 54, 55, 58, or 61. Thus if all three preceding moves > are mutual defection, A = 63 and the rule cooperates. This rule was > designed for use in laboratory experiments as a stooge which had a memory > and appeared to be trustworthy, potentially cooperative, but not gullible > (Nydegger, 1978)."

The program begins with tit for tat for the first three moves, except that if it was the only one to cooperate on the first move and the only one to defect on the second move, it defects on the third move. After the third move, its choice is determined from the 3 preceding outcomes in the following manner.

$$A = 16a_1 + 4a_2 + a_3$$

Where a_i is dependent on the outcome of the previous i th round. If both strategies defect, $a_i = 3$, if the opponent only defects: $a_i = 2$ and finally if it is only this strategy that defects then $a_i = 1$.

Finally this strategy defects if and only if:

$$A \in \{1, 6, 7, 17, 22, 23, 26, 29, 30, 31, 33, 38, 39, 45, 49, 54, 55, 58, 61\}$$

Thus if all three preceding moves are mutual defection, A = 63 and the rule cooperates. This rule was designed for use in laboratory experiments as a stooge which had a memory and appeared to be trustworthy, potentially cooperative, but not gullible.

This strategy came 3rd in Axelrod's original tournament.

Names:

- Nydegger: [Axelrod1980]

static score_history(*my_history: List[axelrod.action.Action], opponent_history: List[axelrod.action.Action], score_map: Dict[Tuple[axelrod.action.Action, axelrod.action.Action], int]*) → int
Implements the Nydegger formula A = 16 a_1 + 4 a_2 + a_3

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_first.**FirstByShubik**
Submitted to Axelrod's first tournament by Martin Shubik.

The description written in [Axelrod1980] is:

> "This rule cooperates until the other defects, and then defects once. If > the other defects again after the rule's cooperation is resumed, the rule > defects twice. In general, the length of retaliation is increased by one for > each departure from mutual cooperation. This rule is described with its > strategic implications in Shubik (1970). Further treatment of its is given > in Taylor (1976).

There is some room for interpretation as to how the strategy reacts to a defection on the turn where it starts to cooperate once more. In Shubik (1970) the strategy is described as:

> "I will play my move 1 to begin with and will continue to do so, so long > as my information shows that the other player has chosen his move 1. If my > information tells me he has used move 2, then I will use move 2 for the > immediate k subsequent periods, after which I will resume using move 1. If > he uses his move 2 again after I have resumed using move 1, then I will > switch to move 2 for the k + 1 immediately subsequent periods ... and so > on, increasing my retaliation by an extra period for each departure from the > (1, 1) steady state."

This is interpreted as:

The player cooperates, if when it is cooperating, the opponent defects it defects for k rounds. After k rounds it starts cooperating again and increments the value of k if the opponent defects again.

This strategy came 5th in Axelrod's original tournament.

Names:

- Shubik: [Axelrod1980]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_first.**FirstBySteinAndRapoport**(*alpha: float = 0.05*)
Submitted to Axelrod's first tournament by William Stein and Amnon Rapoport.

The description written in [Axelrod1980] is:

> "This rule plays tit for tat except that it cooperates on the first four > moves, it defects on the last two moves, and every fifteen moves it checks > to see if the opponent seems to be playing randomly. This check uses a > chi-squared test of the other's transition probabilities and also checks for > alternating moves of CD and DC.

This is implemented as follows:

1. It cooperates for the first 4 moves.
2. It defects on the last 2 moves.
3. Every 15 moves it makes use of a chi-squared test to check if the opponent is playing randomly. If so it defects.

This strategy came 6th in Axelrod's original tournament.

Names:

- SteinAndRapoport: [Axelrod1980]

original_class
alias of *FirstBySteinAndRapoport*

class axelrod.strategies.axelrod_first.**FirstByTidemanAndChieruzzi**
Submitted to Axelrod's first tournament by Nicolas Tideman and Paula Chieruzzi.

The description written in [Axelrod1980] is:

> "This rule begins with cooperation and tit for tat. However, when the > other player finishes his second run of defec- tions, an extra punishment is > instituted, and the number of punishing defections is increased by one with > each run of the other's defections. The other player is given a fresh start > if he is 10 or more points behind, if he has not just started a run of > defections, if it has been at least 20 moves since a fresh start, if there > are at least 10 moves remaining, and if the number of defections differs > from a 50-50 random generator by at least 3.0 standard deviations. A fresh > start involves two cooperations and then play as if the game had just > started. The program defects automatically on the last two moves."

This is interpreted as:

1. Every run of defections played by the opponent increases the number of defections that this strategy retaliates with by 1.
2. **The opponent is given a 'fresh start' if:**
    - it is 10 points behind this strategy
    - and it has not just started a run of defections
    - and it has been at least 20 rounds since the last 'fresh start'
    - and there are more than 10 rounds remaining in the match
    - and the total number of defections differs from a 50-50 random sample by at least 3.0 standard deviations.

    A 'fresh start' is a sequence of two cooperations followed by an assumption that the game has just started (everything is forgotten).
3. The strategy defects on the last two moves.

This strategy came 2nd in Axelrod's original tournament.

Names:

- TidemanAndChieruzzi: [Axelrod1980]

original_class
alias of *FirstByTidemanAndChieruzzi*

class axelrod.strategies.axelrod_first.**FirstByTullock**
Submitted to Axelrod's first tournament by Gordon Tullock.

The description written in [Axelrod1980] is:

> "This rule cooperates on the first eleven moves. It then cooperates 10% > less than the other player has cooperated on the preceding ten moves. This > rule is based on an idea developed in Overcast and Tullock (1971). Professor > Tullock was invited to specify how the idea could be implemented, and he did > so out of scientific interest rather than an expectation that it would be a > likely winner."

This is interpreted as:

Cooperates for the first 11 rounds then randomly cooperates 10% less often than the opponent has in the previous 10 rounds.

This strategy came 13th in Axelrod's original tournament.

Names:

- Tullock: [Axelrod1980]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

Strategies from Axelrod's second tournament. All strategies in this module are prefixed by *SecondBy* to indicate that they were submitted in Axelrod's Second tournament by the given author.

class axelrod.strategies.axelrod_second.**SecondByAppold**
Strategy submitted to Axelrod's second tournament by Scott Appold (K88R) and came in 22nd in that tournament.

Cooperates for first four turns.

After four turns, will cooperate immediately following the first time the opponent cooperates (starting with the opponent's fourth move). Otherwise will cooperate with probability equal to:

- If this strategy defected two turns ago, the portion of the time (historically) that the opponent followed a defection with a cooperation.
- If this strategy cooperated two turns ago, the portion of the time (historically) that the opponent followed a cooperation with a cooperation. The opponent's first move is counted as a response to a cooperation.

Names:

- Appold: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is a placeholder strategy.

class axelrod.strategies.axelrod_second.**SecondByBlack**
Strategy submitted to Axelrod's second tournament by Paul E Black (K83R) and came in fifteenth in that tournament.

The strategy Cooperates for the first five turns. Then it calculates the number of opponent defects in the last five moves and Cooperates with probability *prob_coop'['number_defects*], where:

prob_coop[number_defects] = 1 - (number_defects^ 2 - 1) / 25

Names:

- Black: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is a placeholder strategy.

class axelrod.strategies.axelrod_second.**SecondByBorufsen**
Strategy submitted to Axelrod's second tournament by Otto Borufsen (K32R), and came in third in that tournament.

This player keeps track of the the opponent's responses to own behavior:

- *cd_count* counts: Opponent cooperates as response to player defecting.
- *cc_count* counts: Opponent cooperates as response to player cooperating.

The player has a defect mode and a normal mode. In defect mode, the player will always defect. In normal mode, the player obeys the following ranked rules:

1. If in the last three turns, both the player/opponent defected, then cooperate for a single turn.
2. If in the last three turns, the player/opponent acted differently from each other and they're alternating, then change next defect to cooperate. (Doesn't block third rule.)
3. Otherwise, do tit-for-tat.

Start in normal mode, but every 25 turns starting with the 27th turn, re-evaluate the mode. Enter defect mode if any of the following conditions hold:

- Detected random: Opponent cooperated 7-18 times since last mode evaluation (or start) AND less than 70% of opponent cooperation was in response to player's cooperation, i.e. cc_count / (cc_count+cd_count) < 0.7
- Detect defective: Opponent cooperated fewer than 3 times since last mode evaluation.

When switching to defect mode, defect immediately. The first two rules for normal mode require that last three turns were in normal mode. When starting normal mode from defect mode, defect on first move.

Names:

- Borufsen: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

try_return(*to_return*)
We put the logic here to check for the *flip_next_defect* bit here, and proceed like normal otherwise.

class axelrod.strategies.axelrod_second.**SecondByCave**
Strategy submitted to Axelrod's second tournament by Rob Cave (K49R), and came in fourth in that tournament.

First look for overly-defective or apparently random opponents, and defect if found. That is any opponent meeting one of:

- turn > 39 and percent defects > 0.39
- turn > 29 and percent defects > 0.65
- turn > 19 and percent defects > 0.79

Otherwise, respond to cooperation with cooperation. And respond to defections with either a defection (if opponent has defected at least 18 times) or with a random (50/50) choice. [Cooperate on first.]

Names:

- Cave: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_second.**SecondByChampion**
Strategy submitted to Axelrod's second tournament by Danny Champion.

This player cooperates on the first 10 moves and plays Tit for Tat for the next 15 more moves. After 25 moves, the program cooperates unless all the following are true: the other player defected on the previous move, the other player cooperated less than 60% and the random number between 0 and 1 is greater that the other player's cooperation rate.

Names:

- Champion: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_second.**SecondByColbert**
Strategy submitted to Axelrod's second tournament by William Colbert (K51R) and came in eighteenth in that tournament.

In the first eight turns, this strategy Coopearates on all but the sixth turn, in which it Defects. After that, the strategy responds to an opponent Cooperation with a single Cooperation, and responds to a Defection with a chain of responses: Defect, Defect, Cooperate, Cooperate. During this chain, the strategy ignores opponent's moves.

Names:

- Colbert: [Axelrod1980b]

class axelrod.strategies.axelrod_second.**SecondByEatherley**
Strategy submitted to Axelrod's second tournament by Graham Eatherley.

A player that keeps track of how many times in the game the other player defected. After the other player defects, it defects with a probability equal to the ratio of the other's total defections to the total moves to that point.

Names:

- Eatherley: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_second.**SecondByGetzler**
Strategy submitted to Axelrod's second tournament by Abraham Getzler (K35R) and came in eleventh in that tournament.

Strategy Defects with probability *flack*, where *flack* is calculated as the sum over opponent Defections of 0.5 ^ (turns ago Defection happened).

Names:

- Getzler: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_second.**SecondByGladstein**
Submitted to Axelrod's second tournament by David Gladstein.

This strategy is also known as Tester and is based on the reverse engineering of the Fortran strategies from Axelrod's second tournament.

This strategy is a TFT variant that defects on the first round in order to test the opponent's response. If the opponent ever defects, the strategy 'apologizes' by cooperating and then plays TFT for the rest of the game. Otherwise, it defects as much as possible subject to the constraint that the ratio of its defections to moves remains under 0.5, not counting the first defection.

Names:

- Gladstein: [Axelrod1980b]
- Tester: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_second.**SecondByGraaskampKatzen**
Strategy submitted to Axelrod's second tournament by Jim Graaskamp and Ken Katzen (K60R), and came in sixth in that tournament.

Play Tit-for-Tat at first, and track own score. At select checkpoints, check for a high score. Switch to Default Mode if:

- On move 11, score < 23
- On move 21, score < 53
- On move 31, score < 83
- On move 41, score < 113

- On move 51, score < 143
- On move 101, score < 293

Once in Defect Mode, defect forever.

Names:

- GraaskampKatzen: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_second.**SecondByGrofman**
Submitted to Axelrod's second tournament by Bernard Grofman.

This strategy has 3 phases:

1. First it cooperates on the first two rounds
2. For rounds 3-7 inclusive, it plays the same as the opponent's last move
3. Thereafter, it applies the following logic, looking at its memory of the last 8* rounds (ignoring the most recent round).

- If its own previous move was C and the opponent has defected less than 3 times in the last 8* rounds, cooperate
- If its own previous move was C and the opponent has defected 3 or more times in the last 8* rounds, defect
- If its own previous move was D and the opponent has defected only once or not at all in the last 8* rounds, cooperate
- If its own previous move was D and the opponent has defected more than once in the last 8* rounds, defect

The code looks at the first 7 of the last 8 rounds, ignoring the most recent round.

Names: - Grofman's strategy: [Axelrod1980b] - K86R: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_second.**SecondByHarrington**
Strategy submitted to Axelrod's second tournament by Paul Harrington (K75R) and came in eighth in that tournament.

This strategy has three modes: Normal, Fair-weather, and Defect. These mode names were not present in Harrington's submission.

In Normal and Fair-weather modes, the strategy begins by:

- Update history
- Try to detect random opponent if turn is multiple of 15 and >=30.
- Check if *burned* flag should be raised.
- Check for Fair-weather opponent if turn is 38.

Updating history means to increment the correct cell of the *move_history*. *move_history* is a matrix where the columns are the opponent's previous move and the rows are indexed by the combo of this player's and the opponent's moves two turns ago. [The upper-left cell must be all Cooperations, but otherwise order doesn't matter.] After we enter Defect mode, *move_history* won't be used again.

If the turn is a multiple of 15 and >=30, then attempt to detect random. If random is detected, enter Defect mode and defect immediately. If the player was previously in Defect mode, then do not re-enter. The random detection

logic is a modified Pearson's Chi Squared test, with some additional checks. [More details in *detect_random* docstrings.]

Some of this player's moves are marked as "generous." If this player made a generous move two turns ago and the opponent replied with a Defect, then raise the *burned* flag. This will stop certain generous moves later.

The player mostly plays Tit-for-Tat for the first 36 moves, then defects on the 37th move. If the opponent cooperates on the first 36 moves, and defects on the 37th move also, then enter Fair-weather mode and cooperate this turn. Entering Fair-weather mode is extremely rare, since this can only happen if the opponent cooperates for the first 36 then defects unprovoked on the 37th. (That is, this player's first 36 moves are also Cooperations, so there's nothing really to trigger an opponent Defection.)

Next in Normal Mode:

1. Check for defect and parity streaks.
2. Check if cooperations are scheduled.
3. Otherwise,

- If turn < 37, Tit-for-Tat.
- If turn = 37, defect, mark this move as generous, and schedule two more cooperations**.
- If turn > 37, then if *burned* flag is raised, then Tit-for-Tat. Otherwise, Tit-for-Tat with probability 1 - *prob*. And with probability *prob*, defect, schedule two cooperations, mark this move as generous, and increase *prob* by 5%.

** Scheduling two cooperations means to set *more_coop* flag to two. If in Normal mode and no streaks are detected, then the player will cooperate and lower this flag, until hitting zero. It's possible that the flag can be overwritten. Notable on the 37th turn defect, this is set to two, but the 38th turn Fair-weather check will set this.

If the opponent's last twenty moves were defections, then defect this turn. Then check for a parity streak, by flipping the parity bit (there are two streaks that get tracked which are something like odd and even turns, but this flip bit logic doesn't get run every turn), then incrementing the parity streak that we're pointing to. If the parity streak that we're pointing to is then greater than *parity_limit* then reset the streak and cooperate immediately. *parity_limit* is initially set to five, but after it has been hit eight times, it decreases to three. The parity streak that we're pointing to also gets incremented if in normal mode and we defect but not on turn 38, unless we are defecting as the result of a defect streak. Note that the parity streaks resets but the defect streak doesn't.

If *more_coop* >= 1, then we cooperate and lower that flag here, in Normal mode after checking streaks. Still lower this flag if cooperating as the result of a parity streak or in Fair-weather mode.

Then use the logic based on turn from above.

In Fair-Weather mode after running the code from above, check if opponent defected last turn. If so, exit Fair-Weather mode, and proceed THIS TURN with Normal mode. Otherwise cooperate.

In Defect mode, update the *exit_defect_meter* (originally zero) by incrementing if opponent defected last turn and decreasing by three otherwise. If *exit_defect_meter* is then 11, then set mode to Normal (for future turns), cooperate and schedule two more cooperations. [Note that this move is not marked generous.]

Names:

- Harrington: [Axelrod1980b]

calculate_chi_squared(*turn*)

Pearson's Chi Squared statistic = sum[(E_i-O_i)^2 / E_i], where O_i are the observed matrix values, and E_i is calculated as number (of defects) in the row times the number in the column over (total number in the matrix minus 1). Equivalently, we expect we expect (for an independent distribution) the total number of recorded turns times the portion in that row times the portion in that column.

In this function, the statistic is non-standard in that it excludes summands where E_i <= 1.

detect_parity_streak(*last_move*)
Switch which *parity_streak* we're pointing to and incerement if the opponent's last move was a Defection. Otherwise reset the flag. Then return true if and only if the *parity_streak* is at least *parity_limit*.

This is similar to detect_streak with alternating streaks, except that these streaks get incremented elsewhere as well.

detect_random(*turn*)
We check if the top-left cell of the matrix (corresponding to all Cooperations) has over 80% of the turns. In which case, we label non-random.

Then we check if over 75% or under 25% of the opponent's turns are Defections. If so, then we label as non-random.

Otherwise we calculates a modified Pearson's Chi Squared statistic on self.history, and returns True (is random) if and only if the statistic is less than or equal to 3.

detect_streak(*last_move*)
Return true if and only if the opponent's last twenty moves are defects.

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

try_return(*to_return*, *lower_flags=True*, *inc_parity=False*)
This will return to_return, with some end-of-turn logic.

class axelrod.strategies.axelrod_second.**SecondByKluepfel**
Strategy submitted to Axelrod's second tournament by Charles Kluepfel (K32R).

This player keeps track of the the opponent's responses to own behavior:

- *cd_count* counts: Opponent cooperates as response to player defecting.
- *dd_count* counts: Opponent defects as response to player defecting.
- *cc_count* counts: Opponent cooperates as response to player cooperating.
- *dc_count* counts: Opponent defects as response to player cooperating.

After 26 turns, the player then tries to detect a random player. The player decides that the opponent is random if cd_counts >= (cd_counts+dd_counts)/2 - 0.75*sqrt(cd_counts+dd_counts) AND cc_counts >= (dc_counts+cc_counts)/2 - 0.75*sqrt(dc_counts+cc_counts). If the player decides that they are playing against a random player, then they will always defect.

Otherwise respond to recent history using the following set of rules:

- If opponent's last three choices are the same, then respond in kind.
- If opponent's last two choices are the same, then respond in kind with probability 90%.
- Otherwise if opponent's last action was to cooperate, then cooperate with probability 70%.
- Otherwise if opponent's last action was to defect, then defect with probability 60%.

Names:

- Kluepfel: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_second.**SecondByLeyvraz**
Strategy submitted to Axelrod's second tournament by Fransois Leyvraz (K68R) and came in twelfth in that tournament.

The strategy uses the opponent's last three moves to decide on an action based on the following ordered rules.

1. If opponent Defected last two turns, then Defect with prob 75%.
2. If opponent Defected three turns ago, then Cooperate.
3. If opponent Defected two turns ago, then Defect.
4. If opponent Defected last turn, then Defect with prob 50%.
5. Otherwise (all Cooperations), then Cooperate.

Names:

- Leyvraz: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is a placeholder strategy.

class axelrod.strategies.axelrod_second.**SecondByMikkelson**
Strategy submitted to Axelrod's second tournament by Ray Mikkelson (K66R) and came in twentieth in that tournament.

The strategy keeps track of a variable called *credit*, which determines if the strategy will Cooperate, in the sense that if *credit* is positive, then the strategy Cooperates. *credit* is initialized to 7. After the first turn, *credit* increments if the opponent Cooperated last turn, and decreases by two otherwise. *credit* is capped above by 8 and below by -7. [*credit* is assessed as postive or negative, after increasing based on opponent's last turn.]

If *credit* is non-positive within the first ten turns, then the strategy Defects and *credit* is set to 4. If *credit* is non-positive later, then the strategy Defects if and only if (total # opponent Defections) / (turn#) is at least 15%. [Turn # starts at 1.]

Names:

- Mikkelson: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_second.**SecondByRichardHufford**
Strategy submitted to Axelrod's second tournament by Richard Hufford (K47R) and came in sixteenth in that tournament.

The strategy tracks opponent "agreements", that is whenever the opponent's previous move is the some as this player's move two turns ago. If the opponent's first move is a Defection, this is counted as a disagreement, and otherwise an agreement. From the agreement counts, two measures are calculated:

- *proportion_agree*: This is the number of agreements (through opponent's last turn) + 2 divided by the current turn number.
- *last_four_num*: The number of agreements in the last four turns. If there have been fewer than four previous turns, then this is number of agreement + (4 - number of past turns).

We then use these measures to decide how to play, using these rules:

1. If *proportion_agree* > 0.9 and *last_four_num* >= 4, then Cooperate.
2. Otherwise if *proportion_agree* >= 0.625 and *last_four_num* >= 2, then Tit-for-Tat.
3. Otherwise, Defect.

However, if the opponent has Cooperated the last *streak_needed* turns, then the strategy deviates from the usual strategy, and instead Defects. (We call such deviation an "aberration".) In the turn immediately after an

aberration, the strategy doesn't override, even if there's a streak of Cooperations. Two turns after an aberration, the strategy: Restarts the Cooperation streak (never looking before this turn); Cooperates; and changes *streak_needed* to:

floor(20.0 * *num_abb_def* / *num_abb_coop*) + 1

Here *num_abb_def* is 2 + the number of times that the opponent Defected in the turn after an aberration, and *num_abb_coop* is 2 + the number of times that the opponent Cooperated in response to an aberration.

Names:

- RichardHufford: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is a placeholder strategy.

class axelrod.strategies.axelrod_second.**SecondByRowsam**
Strategy submitted to Axelrod's second tournament by Glen Rowsam (K58R) and came in 21st in that tournament.

The strategy starts in Normal mode, where it cooperates every turn. Every six turns it checks the score per turn. [Rather the score of all previous turns divided by the turn number, which will be one more than the number of turns scored.] If this measure is less than 2.5 (the strategy is doing badly) and it increases *distrust_points*. *distrust_points* is a variable that starts at 0; if it ever exceeds 6 points, the strategy will enter Defect mode and defect from then on. It will increase *distrust_points* depending on the precise score per turn according to:

- 5 points if score per turn is less than 1.0
- 3 points if score per turn is less than 1.5, but at least 1.0
- 2 points if score per turn is less than 2.0, but at least 1.5
- 1 points if score per turn is less than 2.5, but at least 2.0

If *distrust_points* are increased, then the strategy defects on that turn, then cooperates and defects on the next two turns. [Unless *distrust_points* exceeds 6 points, then it will enter Defect mode immediately.]

Every 18 turns in Normal mode, the strategy will decrement *distrust_score* if it's more than 3. This represents a wearing off effect of distrust.

Names:

- Rowsam: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is a placeholder strategy.

class axelrod.strategies.axelrod_second.**SecondByTester**
Submitted to Axelrod's second tournament by David Gladstein.

This strategy is a TFT variant that attempts to exploit certain strategies. It defects on the first move. If the opponent ever defects, TESTER 'apologies' by cooperating and then plays TFT for the rest of the game. Otherwise TESTER alternates cooperation and defection.

This strategy came 46th in Axelrod's second tournament.

Names:

- Tester: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_second.**SecondByTidemanAndChieruzzi**

Strategy submitted to Axelrod's second tournament by T. Nicolaus Tideman and Paula Chieruzzi (K84R) and came in ninth in that tournament.

This strategy Cooperates if this player's score exceeds the opponent's score by at least *score_to_beat*. *score_to_beat* starts at zero and increases by *score_to_beat_inc* every time the opponent's last two moves are a Cooperation and Defection in that order. *score_to_beat_inc* itself increase by 5 every time the opponent's last two moves are a Cooperation and Defection in that order.

Additionally, the strategy executes a "fresh start" if the following hold:

- The strategy would Defect by score (difference less than *score_to_beat*)
- The opponent did not Cooperate and Defect (in order) in the last two turns.
- It's been at least 10 turns since the last fresh start. Or since the match started if there hasn't been a fresh start yet.

A "fresh start" entails two Cooperations and resetting scores, *scores_to_beat* and *scores_to_beat_inc*.

Names:

- TidemanAndChieruzzi: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action

Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_second.**SecondByTranquilizer**

Submitted to Axelrod's second tournament by Craig Feathers

Description given in Axelrod's "More Effective Choice in the Prisoner's Dilemma" paper: The rule normally cooperates but is ready to defect if the other player defects too often. Thus the rule tends to cooperate for the first dozen or two moves if the other player is cooperating, but then it throws in a defection. If the other player continues to cooperate, then defections become more frequent. But as long as Tranquilizer is maintaining an average payoff of at least 2.25 points per move, it will never defect twice in succession and it will not defect more than one-quarter of the time.

This implementation is based on the reverse engineering of the Fortran strategy K67R from Axelrod's second tournament. Reversed engineered by: Owen Campbell, Will Guo and Mansour Hakem.

The strategy starts by cooperating and has 3 states.

At the start of the strategy it updates its states:

- It counts the number of consecutive defections by the opponent.
- If it was in state 2 it moves to state 0 and calculates the following quantities two_turns_after_good_defection_ratio and two_turns_after_good_defection_ratio_count.

  Formula for:

  two_turns_after_good_defection_ratio:

  self.two_turns_after_good_defection_ratio = (((self.two_turns_after_good_defection_ratio * self.two_turns_after_good_defection_ratio_count) + (3 - (3 * self.dict[opponent.history[-1]])) + (2 * self.dict[self.history[-1]]) - ((self.dict[opponent.history[-1]] * self.dict[self.history[-1]]))) / (self.two_turns_after_good_defection_ratio_count + 1))

  two_turns_after_good_defection_ratio_count = two_turns_after_good_defection_ratio + 1
- If it was in state 1 it moves to state 2 and calculates the following quantities one_turn_after_good_defection_ratio and one_turn_after_good_defection_ratio_count.

  Formula for:

one_turn_after_good_defection_ratio:

self.one_turn_after_good_defection_ratio = (((self.one_turn_after_good_defection_ratio * self.one_turn_after_good_defection_ratio_count) + (3 - (3 * self.dict[opponent.history[-1]])) + (2 * self.dict[self.history[-1]]) - (self.dict[opponent.history[-1]] * self.dict[self.history[-1]])) / (self.one_turn_after_good_defection_ratio_count + 1))

one_turn_after_good_defection_ratio_count:

one_turn_after_good_defection_ratio_count = one_turn_after_good_defection_ratio + 1

If after this it is in state 1 or 2 then it cooperates.

If it is in state 0 it will potentially perform 1 of the 2 following stochastic tests:

1. If average score per turn is greater than 2.25 then it calculates a value of probability:

probability = ((.95 - (((self.one_turn_after_good_defection_ratio) + (self.two_turns_after_good_defection_ratio) - 5) / 15)) + (1 / (((len(self.history))+1) ** 2)) - (self.dict[opponent.history[-1]] / 4))

and will cooperate if a random sampled number is less than that value of probability. If it does not cooperate then the strategy moves to state 1 and defects.

2. If average score per turn is greater than 1.75 but less than 2.25 then it calculates a value of probability:

probability = ((.25 + ((opponent.cooperations + 1) / ((len(self.history)) + 1))) - (self.opponent_consecutive_defections * .25) + ((current_score[0] - current_score[1]) / 100) + (4 / ((len(self.history)) + 1)))

and will cooperate if a random sampled number is less than that value of probability. If not, it defects.

If none of the above holds the player simply plays tit for tat.

Tranquilizer came in 27th place in Axelrod's second torunament.

Names:

- Tranquilizer: [Axelrod1980]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

update_state(*opponent*)
Calculates the ratio values for the one_turn_after_good_defection_ratio, two_turns_after_good_defection_ratio and the probability values, and sets the value of num_turns_after_good_defection.

class axelrod.strategies.axelrod_second.**SecondByWeiner**
Strategy submitted to Axelrod's second tournament by Herb Weiner (K41R), and came in seventh in that tournament.

Play Tit-for-Tat with a chance for forgiveness and a defective override.

The chance for forgiveness happens only if *forgive_flag* is raised (flag discussed below). If raised and *turn* is greater than *grudge*, then override Tit-for-Tat with Cooperation. *grudge* is a variable that starts at 0 and increments 20 with each forgiven Defect (a Defect that is overriden through the forgiveness logic). *forgive_flag* is lower whether logic is overriden or not.

The variable *defect_padding* increments with each opponent Defect, but resets to zero with each opponent Cooperate (or *forgive_flag* lowering) so that it roughly counts Defects between Cooperates. Whenever the opponent Cooperates, if *defect_padding* (before reseting) is odd, then we raise *forgive_flag* for next turn.

Finally a defective override is assessed after forgiveness. If five or more of the opponent's last twelve actions are Defects, then Defect. This will overrule a forgiveness, but doesn't undo the lowering of *forgiveness_flag*.

Note that "last twelve actions" doesn't count the most recent action. Actually the original code updates history after checking for defect override.

Names:

- Weiner: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

try_return(*to_return*)
We put the logic here to check for the defective override.

class axelrod.strategies.axelrod_second.**SecondByWhite**
Strategy submitted to Axelrod's second tournament by Edward C White (K72R) and came in thirteenth in that tournament.

- Cooperate in the first ten turns.
- If the opponent Cooperated last turn then Cooperate.
- **Otherwise Defect if and only if:** floor(log(turn)) * opponent Defections >= turn

Names:

- White: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is a placeholder strategy.

class axelrod.strategies.axelrod_second.**SecondByWmAdams**
Strategy submitted to Axelrod's second tournament by William Adams (K44R), and came in fifth in that tournament.

Count the number of opponent defections after their first move, call *c_defect*. Defect if c_defect equals 4, 7, or 9. If c_defect > 9, then defect immediately after opponent defects with probability = (0.5)^(c_defect-1). Otherwise cooperate.

Names:

- WmAdams: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.axelrod_second.**SecondByYamachi**
Strategy submitted to Axelrod's second tournament by Brian Yamachi (K64R) and came in seventeenth in that tournament.

The strategy keeps track of play history through a variable called *count_them_us_them*, which is a dict indexed by (X, Y, Z), where X is an opponent's move and Y and Z are the following moves by this player and the opponent, respectively. Each turn, we look at our opponent's move two turns ago, call X, and our move last turn, call Y. If (X, Y, C) has occurred more often (or as often) as (X, Y, D), then Cooperate. Otherwise Defect. [Note that this reflects likelihood of Cooperations or Defections in opponent's previous move; we don't update *count_them_us_them* with previous move until next turn.]

Starting with the 41st turn, there's a possibility to override this behavior. If *portion_defect* is between 45% and 55% (exclusive), then Defect, where *portion_defect* equals number of opponent defects plus 0.5 divided by the turn number (indexed by 1). When overriding this way, still record *count_them_us_them* as though the strategy didn't override.

Names:

- Yamachi: [Axelrod1980b]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is a placeholder strategy.

try_return(*to_return*, *opp_def*)
Return *to_return*, unless the turn is greater than 40 AND *portion_defect* is between 45% and 55%.

In this case, still record the history as *to_return* so that the modified behavior doesn't affect the calculation of *count_us_them_us*.

class axelrod.strategies.backstabber.**BackStabber**
Forgives the first 3 defections but on the fourth will defect forever. Defects on the last 2 rounds unconditionally.

Names:

- Backstabber: Original name by Thomas Campbell

original_class
alias of *BackStabber*

class axelrod.strategies.backstabber.**DoubleCrosser**
Forgives the first 3 defections but on the fourth will defect forever. Defects on the last 2 rounds unconditionally.

If 8 <= current round <= 180, if the opponent did not defect in the first 7 rounds, the player will only defect after the opponent has defected twice in-a-row.

Names:

- Double Crosser: Original name by Thomas Campbell

original_class
alias of *DoubleCrosser*

class axelrod.strategies.better_and_better.**BetterAndBetter**
Defects with probability of '(1000 - current turn) / 1000'. Therefore it is less and less likely to defect as the round goes on.

Names:

- Better and Better: [Prison1998]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.bush_mosteller.**BushMosteller**(*c_prob: float = 0.5*, *d_prob: float = 0.5*, *aspiration_level_divider: float = 3.0*, *learning_rate: float = 0.5*)
A player that is based on Bush Mosteller reinforced learning algorithm, it decides what it will play only depending on its own previous payoffs.

The probability of playing C or D will be updated using a stimulus which represents a win or a loss of value based on its previous play's payoff in the specified probability. The more a play will be rewarded through rounds, the more the player will be tempted to use it.

Names:

- Bush Mosteller: [Luis2008]

stimulus_update(*opponent: axelrod.player.Player*)
Updates the stimulus attribute based on the opponent's history. Used by the strategy.

Parameters

opponent [axelrod.Player] The current opponent

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.calculator.**Calculator**
Plays like (Hard) Joss for the first 20 rounds. If periodic behavior is detected, defect forever. Otherwise play TFT.

Names:

- Calculator: [Prison1998]

set_seed(*seed: int = None*)
Set a random seed for the player's random number generator.

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.cooperator.**Cooperator**
A player who only ever cooperates.

Names:

- Cooperator: [Axelrod1984]
- ALLC: [Press2012]
- Always cooperate: [Mittal2009]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.cooperator.**TrickyCooperator**
A cooperator that is trying to be tricky.

Names:

- Tricky Cooperator: Original name by Karol Langner

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Almost always cooperates, but will try to trick the opponent by defecting.

Defect once in a while in order to get a better payout. After 3 rounds, if opponent has not defected to a max history depth of 10, defect.

class axelrod.strategies.cycler.**AntiCycler**
A player that follows a sequence of plays that contains no cycles: CDD CD CCD CCCD CCCCD ...

Names:

- Anti Cycler: Original name by Marc Harper

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.cycler.**Cycler**(*cycle: str = 'CCD'*)
A player that repeats a given sequence indefinitely.

Names:

- Cycler: Original name by Marc Harper

set_cycle(*cycle: str*)
Set or change the cycle.

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.cycler.**CyclerCCCCCD**
Cycles C, C, C, C, C, D

Names:

- Cycler CCCD: Original name by Marc Harper

class axelrod.strategies.cycler.**CyclerCCCD**
Cycles C, C, C, D

Names:

- Cycler CCCD: Original name by Marc Harper

class axelrod.strategies.cycler.**CyclerCCCDCD**
Cycles C, C, C, D, C, D

Names:

- Cycler CCCDCD: Original name by Marc Harper

class axelrod.strategies.cycler.**CyclerCCD**
Cycles C, C, D

Names:

- Cycler CCD: Original name by Marc Harper
- Periodic player CCD: [Mittal2009]

class axelrod.strategies.cycler.**CyclerDC**
Cycles D, C

Names:

- Cycler DC: Original name by Marc Harper

class axelrod.strategies.cycler.**CyclerDDC**
Cycles D, D, C

Names:

- Cycler DDC: Original name by Marc Harper
- Periodic player DDC: [Mittal2009]

class axelrod.strategies.cycler.**EvolvableCycler**(*cycle: str = None*, *cycle_length: int = None*, *mutation_probability: float = 0.2*, *mutation_potency: int = 1*, *seed: int = None*)
Evolvable version of Cycler.

crossover(*other*) → axelrod.evolvable_player.EvolvablePlayer
Creates and returns a new Player instance with a single crossover point.

mutate() → axelrod.evolvable_player.EvolvablePlayer
Basic mutation which may change any random actions in the sequence.

The player class in this module does not obey standard rules of the IPD (as indicated by their classifier). We do not recommend putting a lot of time in to optimising it.

class axelrod.strategies.darwin.**Darwin**
A strategy which accumulates a record (the 'genome') of what the most favourable response in the previous round should have been, and naively assumes that this will remain the correct response at the same round of future trials.

This 'genome' is preserved between opponents, rounds and repetitions of the tournament. It becomes a characteristic of the type and so a single version of this is shared by all instances for each loading of the class.

As this results in information being preserved between tournaments, this is classified as a cheating strategy!

If no record yet exists, the opponent's response from the previous round is returned.

Names:

- Darwin: Original name by Paul Slavin

mutate(*outcome: tuple*, *trial: int*) → None
Select response according to outcome.

reset()
Reset instance properties.

static reset_genome() → None
For use in testing methods.

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.dbs.**DBS**(*discount_factor=0.75*, *promotion_threshold=3*, *violation_threshold=4*, *reject_threshold=3*, *tree_depth=5*)
A strategy that learns the opponent's strategy and uses symbolic noise detection for detecting whether anomalies in player's behavior are deliberate or accidental. From the learned opponent's strategy, a tree search is used to choose the best move.

Default values for the parameters are the suggested values in the article. When noise increases you can try to diminish violation_threshold and rejection_threshold.

Names

- Derived Belief Strategy: [Au2006]

compute_prob_rule(*outcome*, *alpha=1*)
Uses the game history to compute the probability of the opponent playing C, in the outcome situation (example: outcome = (C, C)). When alpha = 1, the results is approximately equal to the frequency of the occurrence of outcome C. alpha is a discount factor that gives more weight to recent events than earlier ones.

Parameters

outcome: tuple of two actions.Action alpha: int, optional. Discount factor. Default is 1.

should_demote(*r_minus*, *violation_threshold=4*)
Checks if the number of successive violations of a deterministic rule (in the opponent's behavior) exceeds the user-defined violation_threshold.

should_promote(*r_plus*, *promotion_threshold=3*)
This function determines if the move r_plus is a deterministic behavior of the opponent, and then returns True, or if r_plus is due to a random behavior (or noise) which would require a probabilistic rule, in which case it returns False.

To do so it looks into the game history: if the k last times when the opponent was in the same situation than in r_plus it played the same thing then then r_plus is considered as a deterministic rule (where K is the user-defined promotion_threshold).

Parameters

r_plus: tuple of (tuple of actions.Action, actions.Action) example: ((C, C), D) r_plus represents one outcome of the history, and the following move played by the opponent.

promotion_threshold: int, optional Number of successive observations needed to promote an opponent behavior as a deterministic rule. Default is 3.

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

update_history_by_cond(*opponent_history*)
Updates self.history_by_cond between each turns of the game.

class axelrod.strategies.dbs.**DeterministicNode**(*action1*, *action2*, *depth*)
Nodes (C, C), (C, D), (D, C), or (D, D) with deterministic choice for siblings.

get_siblings(*policy*)
Returns the siblings node of the current DeterministicNode. Builds 2 siblings (C, X) and (D, X) that are StochasticNodes. Those siblings are of the same depth as the current node. Their probabilities pC are defined by the policy argument.

is_stochastic()
Returns True if self is a StochasticNode.

class axelrod.strategies.dbs.**Node**
Nodes used to build a tree for the tree-search procedure. The tree has Deterministic and Stochastic nodes, as the opponent's strategy is learned as a probability distribution.

class axelrod.strategies.dbs.**StochasticNode**(*own_action*, *pC*, *depth*)
Node that have a probability pC to get to each sibling. A StochasticNode can be written (C, X) or (D, X), with X = C with a probability pC, else X = D.

get_siblings()
Returns the siblings node of the current StochasticNode. There are two siblings which are DeterministicNodes, their depth is equal to current node depth's + 1.

is_stochastic()
Returns True if self is a StochasticNode.

axelrod.strategies.dbs.**create_policy**(*pCC*, *pCD*, *pDC*, *pDD*)
Creates a dict that represents a Policy. As defined in the reference, a Policy is a set of (prev_move, p) where p is the probability to cooperate after prev_move, where prev_move can be (C, C), (C, D), (D, C) or (D, D).

Parameters

pCC, pCD, pDC, pDD [float] Must be between 0 and 1.

axelrod.strategies.dbs.**minimax_tree_search**(*begin_node*, *policy*, *max_depth*)
Tree search function (minimax search procedure) for the tree (built by recursion) corresponding to the opponent's policy, and solves it. Returns a tuple of two floats that are the utility of playing C, and the utility of playing D.

axelrod.strategies.dbs.**move_gen**(*outcome*, *policy*, *depth_search_tree=5*)
Returns the best move considering opponent's policy and last move, using tree-search procedure.

class axelrod.strategies.defector.**Defector**
A player who only ever defects.

Names:

- Defector: [Axelrod1984]
- ALLD: [Press2012]
- Always defect: [Mittal2009]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.defector.**TrickyDefector**
A defector that is trying to be tricky.

Names:

- Tricky Defector: Original name by Karol Langner

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Almost always defects, but will try to trick the opponent into cooperating.

Defect if opponent has cooperated at least once in the past and has defected for the last 3 turns in a row.

class axelrod.strategies.doubler.**Doubler**
Cooperates except when the opponent has defected and the opponent's cooperation count is less than twice their defection count.

Names:

- Doubler: [Prison1998]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.finite_state_machines.**EvolvableFSMPlayer**(*transitions: tuple = None, initial_state: int = None, initial_action: axelrod.action.Action = None, num_states: int = None, mutation_probability: float = 0.1, seed: int = None*)
Abstract base class for evolvable finite state machine players.

create_vector_bounds()
Creates the bounds for the decision variables.

crossover(*other*)
Optional method to allow Player to produce variants in combination with another player. Returns a new Player.

mutate()
Optional method to allow Player to produce a variant (not in place).

classmethod normalize_transitions(*transitions: Sequence[Sequence[T_co]]*) → Tuple[Tuple[Any, ...], ...]
Translate a list of lists to a tuple of tuples.

receive_vector(*vector*)
Read a serialized vector into the set of FSM parameters (less initial state). Then assign those FSM parameters to this class instance.

The vector has three parts. The first is used to define the next state (for each of the player's states - for each opponents action).

The second part is the player's next moves (for each state - for each opponent's actions).

Finally, a probability to determine the player's first move.

class axelrod.strategies.finite_state_machines.**EvolvedFSM16**
A 16 state FSM player trained with an evolutionary algorithm.

Names:

- Evolved FSM 16: Original name by Marc Harper

class axelrod.strategies.finite_state_machines.**EvolvedFSM16Noise05**
A 16 state FSM player trained with an evolutionary algorithm with noisy matches (noise=0.05).

Names:

- Evolved FSM 16 Noise 05: Original name by Marc Harper

class axelrod.strategies.finite_state_machines.**EvolvedFSM4**
A 4 state FSM player trained with an evolutionary algorithm.

Names:

- Evolved FSM 4: Original name by Marc Harper

class axelrod.strategies.finite_state_machines.**EvolvedFSM6**
An 6 state FSM player trained with an evolutionary algorithm.

Evolved using axelrod-dojo version 0.0.8 and axelrod version 4.10.0, trained to maximize score against the short_run_time_strategies with 10 machine states for 500 generations, population size of 40, mutation rate at 0.1, bottleneck at 10, 200 turns, and 0 noise. The resulting strategy had only 6 states in its accessible component.

Names:

- Evolved FSM 6: Original name by Frederick Vincent & Dashiell Fryer

class axelrod.strategies.finite_state_machines.**FSMPlayer**(*transitions: Tuple[Tuple[int, axelrod.action.Action, int, axelrod.action.Action], ...] = ((1, C, 1, C), (1, D, 1, D))*, *initial_state: int = 1*, *initial_action: axelrod.action.Action = C*)
Abstract base class for finite state machine players.

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.finite_state_machines.**Fortress3**
Finite state machine player specified in http://DOI.org/10.1109/CEC.2006.1688322.

Note that the description in http://www.graham-kendall.com/papers/lhk2011.pdf is not correct.

Names:

- Fortress 3: [Ashlock2006b]

class axelrod.strategies.finite_state_machines.**Fortress4**
Finite state machine player specified in http://DOI.org/10.1109/CEC.2006.1688322.

Note that the description in http://www.graham-kendall.com/papers/lhk2011.pdf is not correct.

Names:

- Fortress 4: [Ashlock2006b]

class axelrod.strategies.finite_state_machines.**Predator**
Finite state machine player specified in http://DOI.org/10.1109/CEC.2006.1688322.

Names:

- Predator: [Ashlock2006b]

class axelrod.strategies.finite_state_machines.**Pun1**
FSM player described in [Ashlock2006].

Names:

- Pun1: [Ashlock2006]

class axelrod.strategies.finite_state_machines.**Raider**
FSM player described in http://DOI.org/10.1109/FOCI.2014.7007818.

Names

- Raider: [Ashlock2014]

class axelrod.strategies.finite_state_machines.**Ripoff**
FSM player described in http://DOI.org/10.1109/TEVC.2008.920675.

Names

- Ripoff: [Ashlock2008]

class axelrod.strategies.finite_state_machines.**SimpleFSM**(*transitions: tuple*, *initial_state: int*)
Simple implementation of a finite state machine that transitions between states based on the last round of play.

https://en.wikipedia.org/wiki/Finite-state_machine

move(*opponent_action: axelrod.action.Action*) → axelrod.action.Action
Computes the response move and changes state.

num_states()
Return the number of states of the machine.

class axelrod.strategies.finite_state_machines.**SolutionB1**
FSM player described in http://DOI.org/10.1109/TCIAIG.2014.2326012.

Names

- Solution B1: [Ashlock2015]

class axelrod.strategies.finite_state_machines.**SolutionB5**
FSM player described in http://DOI.org/10.1109/TCIAIG.2014.2326012.

Names

- Solution B5: [Ashlock2015]

class axelrod.strategies.finite_state_machines.**TF1**
A FSM player trained to maximize Moran fixation probabilities.

Names:

- TF1: Original name by Marc Harper

class axelrod.strategies.finite_state_machines.**TF2**
A FSM player trained to maximize Moran fixation probabilities.

Names:

- TF2: Original name by Marc Harper

class axelrod.strategies.finite_state_machines.**TF3**
A FSM player trained to maximize Moran fixation probabilities.

Names:

- TF3: Original name by Marc Harper

class axelrod.strategies.finite_state_machines.**Thumper**
FSM player described in http://DOI.org/10.1109/TEVC.2008.920675.

Names

- Thumper: [Ashlock2008]

class axelrod.strategies.finite_state_machines.**UsuallyCooperates**
This strategy cooperates except after a C following a D.

Names:

- Usually Cooperates (UC): [Ashlock2009]

class axelrod.strategies.finite_state_machines.**UsuallyDefects**
This strategy defects except after a D following a C.

Names:

- Usually Defects (UD): [Ashlock2009]

class axelrod.strategies.forgiver.**Forgiver**
A player starts by cooperating however will defect if at any point the opponent has defected more than 10 percent of the time

Names:

- Forgiver: Original name by Thomas Campbell

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Begins by playing C, then plays D if the opponent has defected more than 10 percent of the time.

class axelrod.strategies.forgiver.**ForgivingTitForTat**
A player starts by cooperating however will defect if at any point, the opponent has defected more than 10 percent of the time, and their most recent decision was defect.

Names:

- Forgiving Tit For Tat: Original name by Thomas Campbell

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Begins by playing C, then plays D if the opponent has defected more than 10 percent of the time and their most recent decision was defect.

Stochastic variants of Lookup table based-strategies, trained with particle swarm algorithms.

For the original see: https://gist.github.com/GDKO/60c3d0fd423598f3c4e4

class axelrod.strategies.gambler.**EvolvableGambler**(*lookup_dict: dict = None*, *initial_actions: tuple = None*, *pattern: Any = None*, *parameters: axelrod.strategies.lookerup.Plays = None*, *mutation_probability: float = None*, *seed: int = None*)

create_vector_bounds()
Creates the bounds for the decision variables. Ignores extra parameters.

receive_vector(*vector*)
Receives a vector and updates the player's pattern. Ignores extra parameters.

class axelrod.strategies.gambler.**Gambler**(*lookup_dict: dict = None*, *initial_actions: tuple = None*, *pattern: Any = None*, *parameters: axelrod.strategies.lookerup.Plays = None*)
A stochastic version of LookerUp which will select randomly an action in some cases.

Names:

- Gambler: Original name by Georgios Koutsovoulos

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.gambler.**PSOGambler1_1_1**
A 1x1x1 PSOGambler trained with pyswarm.

Names:

- PSO Gambler 1_1_1: Original name by Marc Harper

class axelrod.strategies.gambler.**PSOGambler2_2_2**
A 2x2x2 PSOGambler trained with a particle swarm algorithm (implemented in pyswarm). Original version by Georgios Koutsovoulos.

Names:

- PSO Gambler 2_2_2: Original name by Marc Harper

class axelrod.strategies.gambler.**PSOGambler2_2_2_Noise05**
A 2x2x2 PSOGambler trained with pyswarm with noise=0.05.

Names:

- PSO Gambler 2_2_2 Noise 05: Original name by Marc Harper

class axelrod.strategies.gambler.**PSOGamblerMem1**
A 1x1x0 PSOGambler trained with pyswarm. This is the 'optimal' memory one strategy trained against the set of short run time strategies in the Axelrod library.

Names:

- PSO Gambler Mem1: Original name by Marc Harper

class axelrod.strategies.gambler.**ZDMem2**
A memory two generalization of a zero determinant player.

Names:

- ZDMem2: Original name by Marc Harper
- Unnamed [LiS2014]

class axelrod.strategies.gobymajority.**GoByMajority**(*memory_depth: Union[int, float] = inf*, *soft: bool = True*)
Submitted to Axelrod's second tournament by Gail Grisell. It came 23rd and was written in 10 lines of BASIC.

A player examines the history of the opponent: if the opponent has more defections than cooperations then the player defects.

In case of equal number of defections and cooperations this player will Cooperate. Passing the *soft=False* keyword argument when initialising will create a HardGoByMajority which Defects in case of equality.

An optional memory attribute will limit the number of turns remembered (by default this is 0)

Names:

- Go By Majority: [Axelrod1984]
- Grisell: [Axelrod1980b]
- Soft Majority: [Mittal2009]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is affected by the history of the opponent.

As long as the opponent cooperates at least as often as they defect then the player will cooperate. If at any point the opponent has more defections than cooperations in memory the player defects.

class axelrod.strategies.gobymajority.**GoByMajority10**
GoByMajority player with a memory of 10.

Names:

- Go By Majority 10: Original name by Karol Langner

class axelrod.strategies.gobymajority.**GoByMajority20**
GoByMajority player with a memory of 20.

Names:

- Go By Majority 20: Original name by Karol Langner

class axelrod.strategies.gobymajority.**GoByMajority40**
GoByMajority player with a memory of 40.

Names:

- Go By Majority 40: Original name by Karol Langner

class axelrod.strategies.gobymajority.**GoByMajority5**
GoByMajority player with a memory of 5.

Names:

- Go By Majority 5: Original name by Karol Langner

class axelrod.strategies.gobymajority.**HardGoByMajority**(*memory_depth: Union[int, float] = inf*)
A player examines the history of the opponent: if the opponent has more defections than cooperations then the player defects. In case of equal number of defections and cooperations this player will Defect.

An optional memory attribute will limit the number of turns remembered (by default this is 0)

Names:

- Hard Majority: [Mittal2009]

class axelrod.strategies.gobymajority.**HardGoByMajority10**
HardGoByMajority player with a memory of 10.

Names:

- Hard Go By Majority 10: Original name by Karol Langner

class axelrod.strategies.gobymajority.**HardGoByMajority20**
HardGoByMajority player with a memory of 20.

Names:

- Hard Go By Majority 20: Original name by Karol Langner

class axelrod.strategies.gobymajority.**HardGoByMajority40**
HardGoByMajority player with a memory of 40.

Names:

- Hard Go By Majority 40: Original name by Karol Langner

class axelrod.strategies.gobymajority.**HardGoByMajority5**
HardGoByMajority player with a memory of 5.

Names:

- Hard Go By Majority 5: Original name by Karol Langner

class axelrod.strategies.gradualkiller.**GradualKiller**
It begins by defecting in the first five moves, then cooperates two times. It then defects all the time if the opponent has defected in move 6 and 7, else cooperates all the time. Initially designed to stop Gradual from defeating TitForTat in a 3 Player tournament.

Names

- Gradual Killer: [Prison1998]

original_class
alias of *GradualKiller*

class axelrod.strategies.grudger.**Aggravater**
Grudger, except that it defects on the first 3 turns

Names

- Aggravater: Original name by Thomas Campbell

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.grudger.**Capri**
CAPRI is a memory-3 strategy proposed in [Murase2020]. Its behavior is defined by the following five rules applied to the last 3 moves of the player and the opponent:

- C: Cooperate at mutual cooperation. This rule prescribes c at (ccc, ccc).
- A: Accept punishment when you mistakenly defected from mutual cooperation. This rule prescribes c at (ccd, ccc), (cdc, ccd), (dcc, cdc), and (ccc, dcc).
- P: Punish your co-player by defecting once when he defected from mutual cooperation. This rule prescribes d at (ccc, ccd), and then c at (ccd, cdc), (cdc, dcc), and (dcc, ccc).
- R: Recover cooperation when you or your co-player cooperated at mutual defection. This rule prescribes c at (ddd, ddc), (ddc, dcc), (dcc, ccc), (ddc, ddd), (dcc, ddc), (ccc, dcc), (ddc, ddc), and (dcc, dcc).
- I: In all the other cases, defect.

The original implementation used in [Murase2020] is available at https://github.com/yohm/sim_exhaustive_m3_PDgame

Names:

- CAPRI: Original Name by Y. Murase et al. [Murase2020]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is a placeholder strategy.

class axelrod.strategies.grudger.**EasyGo**
A player starts by defecting however will cooperate if at any point the opponent has defected.

Names:

- Easy Go: [Prison1998]
- Reverse Grudger (RGRIM): [Li2011]
- Fool Me Forever: [Harper2017]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Begins by playing D, then plays C for the remaining rounds if the opponent ever plays D.

class axelrod.strategies.grudger.**ForgetfulGrudger**
A player starts by cooperating however will defect if at any point the opponent has defected, but forgets after mem_length matches.

Names:

- Forgetful Grudger: Original name by Geraint Palmer

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Begins by playing C, then plays D for mem_length rounds if the opponent ever plays D.

class axelrod.strategies.grudger.**GeneralSoftGrudger**(*n: int = 1*, *d: int = 4*, *c: int = 2*)
A generalization of the SoftGrudger strategy. SoftGrudger punishes by playing: D, D, D, D, C, C. after a defection by the opponent. GeneralSoftGrudger only punishes after its opponent defects a specified amount of times consecutively. The punishment is in the form of a series of defections followed by a 'penance' of a series of consecutive cooperations.

Names:

- General Soft Grudger: Original Name by J. Taylor Smith

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Punishes after its opponent defects 'n' times consecutively. The punishment is in the form of 'd' defections followed by a penance of 'c' consecutive cooperations.

class axelrod.strategies.grudger.**Grudger**
A player starts by cooperating however will defect if at any point the opponent has defected.

This strategy came 7th in Axelrod's original tournament.

Names:

- Friedman's strategy: [Axelrod1980]
- Grudger: [Li2011]
- Grim: [Berg2015]
- Grim Trigger: [Banks1990]
- Spite: [Beaufils1997]
- Spiteful: [Mathieu2015]
- Vengeful: [Ashlock2009]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Begins by playing C, then plays D for the remaining rounds if the opponent ever plays D.

class axelrod.strategies.grudger.**GrudgerAlternator**
A player starts by cooperating until the first opponents defection, then alternates D-C.

Names:

- c_then_per_dc: [Prison1998]

- Grudger Alternator: Original name by Geraint Palmer

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Begins by playing C, then plays Alternator for the remaining rounds if the opponent ever plays D.

class axelrod.strategies.grudger.**OppositeGrudger**
A player starts by defecting however will cooperate if at any point the opponent has cooperated.

Names:

- Opposite Grudger: Original name by Geraint Palmer

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Begins by playing D, then plays C for the remaining rounds if the opponent ever plays C.

class axelrod.strategies.grudger.**SoftGrudger**
A modification of the Grudger strategy. Instead of punishing by always defecting: punishes by playing: D, D, D, D, C, C. (Will continue to cooperate afterwards).

- Soft Grudger (SGRIM): [Li2011]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Begins by playing C, then plays D, D, D, D, C, C against a defection

class axelrod.strategies.grudger.**SpitefulCC**
Behaves like Grudger after cooperating for 2 turns

Names:

- spiteful_cc: [Mathieu2015]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Cooperates until the opponent defects. Then defects forever. Always cooperates twice at the start.

class axelrod.strategies.grumpy.**Grumpy**(*starting_state: str = 'Nice'*, *grumpy_threshold: int = 10*, *nice_threshold: int = -10*)
A player that defects after a certain level of grumpiness. Grumpiness increases when the opponent defects and decreases when the opponent co-operates.

Names:

- Grumpy: Original name by Jason Young

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
A player that gets grumpier the more the opposition defects, and nicer the more they cooperate.

Starts off Nice, but becomes grumpy once the grumpiness threshold is hit. Won't become nice once that grumpy threshold is hit, but must reach a much lower threshold before it becomes nice again.

class axelrod.strategies.handshake.**Handshake**(*initial_plays: List[axelrod.action.Action] = None*)
Starts with C, D. If the opponent plays the same way, cooperate forever, else defect forever.

Names:

- Handshake: [Robson1990]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.hmm.**EvolvableHMMPlayer**(*transitions_C=None*, *transitions_D=None*, *emission_probabilities=None*, *initial_state=0*, *initial_action=C*, *num_states=None*, *mutation_probability=None*, *seed: int = None*)

Evolvable version of HMMPlayer.

create_vector_bounds()
Creates the bounds for the decision variables.

crossover(*other*)
Optional method to allow Player to produce variants in combination with another player. Returns a new Player.

mutate()
Optional method to allow Player to produce a variant (not in place).

receive_vector(*vector*)
Read a serialized vector into the set of HMM parameters (less initial state). Then assign those HMM parameters to this class instance.

Assert that the vector has the right number of elements for an HMMParams class with self.num_states.

Assume the first num_states^2 entries are the transitions_C matrix. The next num_states^2 entries are the transitions_D matrix. Then the next num_states entries are the emission_probabilities vector. Finally the last entry is the initial_action.

class axelrod.strategies.hmm.**EvolvedHMM5**
An HMM-based player with five hidden states trained with an evolutionary algorithm.

Names:

- Evolved HMM 5: Original name by Marc Harper

class axelrod.strategies.hmm.**HMMPlayer**(*transitions_C=None*, *transitions_D=None*, *emission_probabilities=None*, *initial_state=0*, *initial_action=C*)

Abstract base class for Hidden Markov Model players.

Names

- HMM Player: Original name by Marc Harper

is_stochastic() → bool
Determines if the player is stochastic.

set_seed(*seed=None*)
Set a random seed for the player's random number generator.

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.hmm.**SimpleHMM**(*transitions_C*, *transitions_D*, *emission_probabilities*, *initial_state*)

Implementation of a basic Hidden Markov Model. We assume that the transition matrix is conditioned on the opponent's last action, so there are two transition matrices. Emission distributions are stored as Bernoulli probabilities for each state. This is essentially a stochastic FSM.

https://en.wikipedia.org/wiki/Hidden_Markov_model

is_well_formed() → bool

Determines if the HMM parameters are well-formed:

- Both matrices are stochastic
- Emissions probabilities are in [0, 1]
- The initial state is valid.

move(*opponent_action: axelrod.action.Action*) → axelrod.action.Action
Changes state and computes the response action.

Parameters

opponent_action: Axelrod.Action The opponent's last action.

axelrod.strategies.hmm.**is_stochastic_matrix**(*m*, *ep=1e-08*) → bool
Checks that the matrix m (a list of lists) is a stochastic matrix.

axelrod.strategies.hmm.**mutate_row**(*row*, *mutation_probability*, *rng*)
, crossover_lists_of_lists Given a row of probabilities, randomly change each entry with probability *mutation_probability* (a value between 0 and 1). If changing, then change by a value randomly (uniformly) chosen from [-0.25, 0.25] bounded by 0 and 100%.

class axelrod.strategies.hunter.**AlternatorHunter**
A player who hunts for alternators.

Names:

- Alternator Hunter: Original name by Karol Langner

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.hunter.**CooperatorHunter**
A player who hunts for cooperators.

Names:

- Cooperator Hunter: Original name by Karol Langner

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.hunter.**CycleHunter**
Hunts strategies that play cyclically, like any of the Cyclers, Alternator, etc.

Names:

- Cycle Hunter: Original name by Marc Harper

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.hunter.**DefectorHunter**
A player who hunts for defectors.

Names:

- Defector Hunter: Original name by Karol Langner

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.hunter.**EventualCycleHunter**
Hunts strategies that eventually play cyclically.

Names:

- Eventual Cycle Hunter: Original name by Marc Harper

strategy(*opponent: axelrod.player.Player*) → None
Actual strategy definition that determines player's action.

class axelrod.strategies.hunter.**MathConstantHunter**
A player who hunts for mathematical constant players.

Names:

Math Constant Hunter: Original name by Karol Langner

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Check whether the number of cooperations in the first and second halves of the history are close. The variance of the uniform distribution (1/4) is a reasonable delta but use something lower for certainty and avoiding false positives. This approach will also detect a lot of random players.

class axelrod.strategies.hunter.**RandomHunter**
A player who hunts for random players.

Names:

- Random Hunter: Original name by Karol Langner

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
A random player is unpredictable, which means the conditional frequency of cooperation after cooperation, and defection after defections, should be close to 50%... although how close is debatable.

class axelrod.strategies.inverse.**Inverse**
A player who defects with a probability that diminishes relative to how long ago the opponent defected.

Names:

- Inverse: Original Name by Karol Langner

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Looks at opponent history to see if they have defected.

If so, player defection is inversely proportional to when this occurred.

class axelrod.strategies.lookerup.**EvolvableLookerUp**(*lookup_dict: dict = None*, *initial_actions: tuple = None*, *pattern: Any = None*, *parameters: axelrod.strategies.lookerup.Plays = None*, *mutation_probability: float = None*, *seed: int = None*)

crossover(*other*)
Optional method to allow Player to produce variants in combination with another player. Returns a new Player.

mutate()
Optional method to allow Player to produce a variant (not in place).

class axelrod.strategies.lookerup.**EvolvedLookerUp1_1_1**
A 1 1 1 Lookerup trained with an evolutionary algorithm.

Names:

- Evolved Lookerup 1 1 1: Original name by Marc Harper

class axelrod.strategies.lookerup.**EvolvedLookerUp2_2_2**
A 2 2 2 Lookerup trained with an evolutionary algorithm.

Names:

- Evolved Lookerup 2 2 2: Original name by Marc Harper

class axelrod.strategies.lookerup.**LookerUp**(*lookup_dict: dict = None*, *initial_actions: tuple = None*, *pattern: Any = None*, *parameters: axelrod.strategies.lookerup.Plays = None*)

This strategy uses a LookupTable to decide its next action. If there is not enough history to use the table, it calls from a list of self.initial_actions.

if self_depth=2, op_depth=3, op_openings_depth=5, LookerUp finds the last 2 plays of self, the last 3 plays of opponent and the opening 5 plays of opponent. It then looks those up on the LookupTable and returns the appropriate action. If 5 rounds have not been played (the minimum required for op_openings_depth), it calls from self.initial_actions.

LookerUp can be instantiated with a dictionary. The dictionary uses tuple(tuple, tuple, tuple) or Plays as keys. for example.

- self_plays: depth=2
- op_plays: depth=1
- op_openings: depth=0:

```
{Plays((C, C), (C), ()): C,
 Plays((C, C), (D), ()): D,
 Plays((C, D), (C), ()): D,  <- example below
 Plays((C, D), (D), ()): D,
 Plays((D, C), (C), ()): C,
 Plays((D, C), (D), ()): D,
 Plays((D, D), (C), ()): C,
 Plays((D, D), (D), ()): D}
```

From the above table, if the player last played C, D and the opponent last played C (here the initial opponent play is ignored) then this round, the player would play D.

The dictionary must contain all possible permutations of C's and D's.

LookerUp can also be instantiated with *pattern=str/tuple* of actions, and:

```
parameters=Plays(
    self_plays=player_depth: int,
    op_plays=op_depth: int,
    op_openings=op_openings_depth: int)
```

It will create keys of len=2 ** (sum(parameters)) and map the pattern to the keys.

initial_actions is a tuple such as (C, C, D). A table needs initial actions equal to max(self_plays depth, opponent_plays depth, opponent_initial_plays depth). If provided initial_actions is too long, the extra will be ignored. If provided initial_actions is too short, the shortfall will be made up with C's.

Some well-known strategies can be expressed as special cases; for example Cooperator is given by the dict (All history is ignored and always play C):

```
{Plays((), (), ()) : C}
```

Tit-For-Tat is given by (The only history that is important is the opponent's last play.):

```
{Plays((), (D,), ()): D,
 Plays((), (C,), ()): C}
```

LookerUp's LookupTable defaults to Tit-For-Tat. The initial_actions defaults to playing C.

Names:

- Lookerup: Original name by Martin Jones

lookup_table_display(*sort_by: tuple = ('op_openings', 'self_plays', 'op_plays')*) → str
Returns a string for printing lookup_table info in specified order.

Parameters **sort_by** – only_elements='self_plays', 'op_plays', 'op_openings'

strategy(*opponent: axelrod.player.Player*) → Reaction
This is a placeholder strategy.

class axelrod.strategies.lookerup.**LookupTable**(*lookup_dict: dict*)
LookerUp and its children use this object to determine their next actions.

It is an object that creates a table of all possible plays to a specified depth and the action to be returned for each combination of plays. The "get" method returns the appropriate response. For the table containing:

```
....
Plays(self_plays=(C, C), op_plays=(C, D), op_openings=(D, C): D
Plays(self_plays=(C, C), op_plays=(C, D), op_openings=(D, D): C
...
```

with: player.history[-2:]=[C, C] and opponent.history[-2:]=[C, D] and opponent.history[:2]=[D, D], calling LookupTable.get(plays=(C, C), op_plays=(C, D), op_openings=(D, D)) will return C.

Instantiate the table with a lookup_dict. This is {(self_plays_tuple, op_plays_tuple, op_openings_tuple): action, ... }. It must contain every possible permutation with C's and D's of the above tuple. so:

```
good_dict = {((C,), (C,), ()): C,
             ((C,), (D,), ()): C,
             ((D,), (C,), ()): D,
             ((D,), (D,), ()): C}

bad_dict = {((C,), (C,), ()): C,
            ((C,), (D,), ()): C,
            ((D,), (C,), ()): D}
```

LookupTable.from_pattern() creates an ordered list of keys for you and maps the pattern to the keys.:

```
LookupTable.from_pattern(pattern=(C, D, D, C),
    player_depth=0, op_depth=1, op_openings_depth=1
)
```

creates the dictionary:

```
{Plays(self_plays=(), op_plays=(C), op_openings=(C)): C,
 Plays(self_plays=(), op_plays=(C), op_openings=(D)): D,
 Plays(self_plays=(), op_plays=(D), op_openings=(C)): D,
 Plays(self_plays=(), op_plays=(D), op_openings=(D)): C,}
```

and then returns a LookupTable with that dictionary.

display(*sort_by: tuple = ('op_openings', 'self_plays', 'op_plays')*) → str
Returns a string for printing lookup_table info in specified order.

Parameters **sort_by** – only_elements='self_plays', 'op_plays', 'op_openings'

class axelrod.strategies.lookerup.**Plays**(*self_plays*, *op_plays*, *op_openings*)

op_openings
Alias for field number 2

op_plays
Alias for field number 1

self_plays
Alias for field number 0

class axelrod.strategies.lookerup.**Winner12**
A lookup table based strategy.

Names:

- Winner12: [Mathieu2015]

class axelrod.strategies.lookerup.**Winner21**
A lookup table based strategy.

Names:

- Winner21: [Mathieu2015]

axelrod.strategies.lookerup.**create_lookup_table_keys**(*player_depth: int*, *op_depth: int*, *op_openings_depth: int*) → list
Returns a list of Plays that has all possible permutations of C's and D's for each specified depth. the list is in order, C < D sorted by ((player_tuple), (op_tuple), (op_openings_tuple)). create_lookup_keys(2, 1, 0) returns:

```
[Plays(self_plays=(C, C), op_plays=(C,), op_openings=()),
 Plays(self_plays=(C, C), op_plays=(D,), op_openings=()),
 Plays(self_plays=(C, D), op_plays=(C,), op_openings=()),
 Plays(self_plays=(C, D), op_plays=(D,), op_openings=()),
 Plays(self_plays=(D, C), op_plays=(C,), op_openings=()),
 Plays(self_plays=(D, C), op_plays=(D,), op_openings=()),
 Plays(self_plays=(D, D), op_plays=(C,), op_openings=()),
 Plays(self_plays=(D, D), op_plays=(D,), op_openings=())]
```

axelrod.strategies.lookerup.**get_last_n_plays**(*player: axelrod.player.Player*, *depth: int*) → tuple
Returns the last N plays of player as a tuple.

axelrod.strategies.lookerup.**make_keys_into_plays**(*lookup_table: dict*) → dict
Returns a dict where all keys are Plays.

class axelrod.strategies.mathematicalconstants.**CotoDeRatio**
The player will always aim to bring the ratio of co-operations to defections closer to the ratio as given in a sub class

Names:

- Co to Do Ratio: Original Name by Timothy Standen

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.mathematicalconstants.**Golden**
The player will always aim to bring the ratio of co-operations to defections closer to the golden mean

Names:

- Golden: Original Name by Timothy Standen

class axelrod.strategies.mathematicalconstants.**Pi**
The player will always aim to bring the ratio of co-operations to defections closer to the pi

Names:

- Pi: Original Name by Timothy Standen

class axelrod.strategies.mathematicalconstants.**e**
The player will always aim to bring the ratio of co-operations to defections closer to the e

Names:

- e: Original Name by Timothy Standen

Memory Two strategies.

class axelrod.strategies.memorytwo.**AON2**
AON2 a memory two strategy introduced in [Hilbe2017]. It belongs to the AONk (all-or-none) family of strategies. These strategies were designed to satisfy the three following properties:

1. Mutually Cooperative. A strategy is mutually cooperative if there are histories for which the strategy prescribes to cooperate, and if it continues to cooperate after rounds with mutual cooperation (provided the last k actions of the focal player were actually consistent).

2. Error correcting. A strategy is error correcting after at most k rounds if, after any history, it generally takes a group of players at most k + 1 rounds to re-establish mutual cooperation.

3. Retaliating. A strategy is retaliating for at least k rounds if, after rounds in which the focal player cooperated while the coplayer defected, the strategy responds by defecting the following k rounds.

In [Hilbe2017] the following vectors are reported as "equivalent" to AON2 with their respective self-cooperation rate (note that these are not the same):

1. [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1], self-cooperation rate: 0.952 2. [1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1], self-cooperation rate: 0.951 3. [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1], self-cooperation rate: 0.951 4. [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1], self-cooperation rate: 0.952

AON2 is implemented using vector 1 due its self-cooperation rate.

In essence it is a strategy that starts off by cooperating and will cooperate again only after the states (CC, CC), (CD, CD), (DC, DC), (DD, DD).

Names:

- AON2: [Hilbe2017]

class axelrod.strategies.memorytwo.**DelayedAON1**
Delayed AON1 a memory two strategy also introduced in [Hilbe2017] and belongs to the AONk family. Note that AON1 is equivalent to Win Stay Lose Shift.

In [Hilbe2017] the following vectors are reported as "equivalent" to Delayed AON1 with their respective self-cooperation rate (note that these are not the same):

1. [1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1], self-cooperation rate: 0.952 2. [1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1], self-cooperation rate: 0.970 3. [1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1], self-cooperation rate: 0.971

Delayed AON1 is implemented using vector 3 due its self-cooperation rate.

In essence it is a strategy that starts off by cooperating and will cooperate again only after the states (CC, CC), (CD, CD), (CD, DD), (DD, CD), (DC, DC) and (DD, DD).

Names:

- Delayed AON1: [Hilbe2017]

class axelrod.strategies.memorytwo.**MEM2**
A memory-two player that switches between TFT, TFTT, and ALLD.

Note that the reference claims that this is a memory two strategy but in fact it is infinite memory. This is because the player plays as ALLD if ALLD has ever been selected twice, which can only be known if the entire history of play is accessible.

Names:

- MEM2: [Li2014]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.memorytwo.**MemoryTwoPlayer**(*sixteen_vector: Optional[Tuple[float, ...]] = None*, *initial: Optional[Tuple[axelrod.action.Action, axelrod.action.Action]] = None*)
Uses a sixteen-vector for strategies based on the 16 conditional probabilities P(X | I,J,K,L) where X, I, J, K, L in [C, D] and I, J are the players last two moves and K, L are the opponents last two moves. These conditional probabilities are the following: 1. P(C|CC, CC) 2. P(C|CC, CD) 3. P(C|CC, DC) 4. P(C|CC, DD) 5. P(C|CD, CC) 6. P(C|CD, CD) 7. P(C|CD, DC) 8. P(C|CD, DD) 9. P(C|DC, CC) 10. P(C|DC, CD) 11. P(C|DC, DC) 12. P(C|DC, DD) 13. P(C|DD, CC) 14. P(C|DD, CD) 15. P(C|DD, DC) 16. P(C|DD, DD) Cooperator is set as the default player if sixteen_vector is not given.

Names

- Memory Two: [Hilbe2017]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

Memory One strategies. Note that there are Memory One strategies in other files, including titfortat.py and zero_determinant.py

class axelrod.strategies.memoryone.**ALLCorALLD**
This strategy is at the parameter extreme of the ZD strategies (phi = 0). It simply repeats its last move, and so mimics ALLC or ALLD after round one. If the tournament is noisy, there will be long runs of C and D.

For now starting choice is random of 0.6, but that was an arbitrary choice at implementation time.

Names:

- ALLC or ALLD: Original name by Marc Harper
- Repeat: [Akin2015]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is a placeholder strategy.

class axelrod.strategies.memoryone.**FirmButFair**
A strategy that cooperates on the first move, and cooperates except after receiving a sucker payoff.

Names:

- Firm But Fair: [Frean1994]

class axelrod.strategies.memoryone.**GTFT**(*p: float = None*)
Generous Tit For Tat Strategy.

Names:

- Generous Tit For Tat: [Nowak1993]

- Naive peace maker: [Gaudesi2016]
- Soft Joss: [Gaudesi2016]

class axelrod.strategies.memoryone.**MemoryOnePlayer**(*four_vector: Tuple[float, float, float, float] = None*, *initial: axelrod.action.Action = C*)

Uses a four-vector for strategies based on the last round of play, (P(C|CC), P(C|CD), P(C|DC), P(C|DD)). Win-Stay Lose-Shift is set as the default player if four_vector is not given. Intended to be used as an abstract base class or to at least be supplied with a initializing four_vector.

Names

- Memory One: [Nowak1990]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action

This is a placeholder strategy.

class axelrod.strategies.memoryone.**ReactivePlayer**(*probabilities: Tuple[float, float]*)

A generic reactive player. Defined by 2 probabilities conditional on the opponent's last move: P(C|C), P(C|D).

Names:

- Reactive: [Nowak1989]

class axelrod.strategies.memoryone.**SoftJoss**(*q: float = 0.9*)

Defects with probability 0.9 when the opponent defects, otherwise emulates Tit-For-Tat.

Names:

- Soft Joss: [Prison1998]

class axelrod.strategies.memoryone.**StochasticCooperator**

Stochastic Cooperator.

Names:

- Stochastic Cooperator: [Adami2013]

class axelrod.strategies.memoryone.**StochasticWSLS**(*ep: float = 0.05*)

Stochastic WSLS, similar to Generous TFT. Note that this is not the same as Stochastic WSLS described in [Amaral2016], that strategy is a modification of WSLS that learns from the performance of other strategies.

Names:

- Stochastic WSLS: Original name by Marc Harper

class axelrod.strategies.memoryone.**WinShiftLoseStay**(*initial: axelrod.action.Action = D*)

Win-Shift Lose-Stay, also called Reverse Pavlov.

Names:

- WSLS: [Li2011]

class axelrod.strategies.memoryone.**WinStayLoseShift**

Win-Stay Lose-Shift, also called Pavlov.

Names:

- Win Stay Lose Shift: [Nowak1993]
- WSLS: [Stewart2012]
- Pavlov: [Kraines1989]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.meta.**MemoryDecay**(*p_memory_delete: float = 0.1*, *p_memory_alter: float = 0.03*, *loss_value: float = -2*, *gain_value: float = 1*, *memory: list = None*, *start_strategy: axelrod.player.Player = <class 'axelrod.strategies.titfortat.TitForTat'>*, *start_strategy_duration: int = 15*)
A player utilizes the (default) Tit for Tat strategy for the first (default) 15 turns, at the same time memorizing the opponent's decisions. After the 15 turns have passed, the player calculates a 'net cooperation score' (NCS) for their opponent, weighing decisions to Cooperate as (default) 1, and to Defect as (default) -2. If the opponent's NCS is below 0, the player defects; otherwise, they cooperate.

The player's memories of the opponent's decisions have a random chance to be altered (i.e., a C decision becomes D or vice versa; default probability is 0.03) or deleted (default probability is 0.1).

It is possible to pass a different axelrod player class to change the initial player behavior.

Name: Memory Decay

gain_loss_translate()
Translates the actions (D and C) to numeric values (loss_value and gain_value).

memory_alter()
Alters memory entry, i.e. puts C if there's a D and vice versa.

memory_delete()
Deletes memory entry.

meta_strategy(*results*, *opponent*)
Determine the meta result based on results of all players. Override this function in child classes.

class axelrod.strategies.meta.**MetaHunter**
A player who uses a selection of hunters.

Names

- Meta Hunter: Original name by Karol Langner

static meta_strategy(*results*, *opponent*)
Determine the meta result based on results of all players. Override this function in child classes.

class axelrod.strategies.meta.**MetaHunterAggressive**(*team=None*)
A player who uses a selection of hunters.

Names

- Meta Hunter Aggressive: Original name by Marc Harper

static meta_strategy(*results*, *opponent*)
Determine the meta result based on results of all players. Override this function in child classes.

class axelrod.strategies.meta.**MetaMajority**(*team=None*)
A player who goes by the majority vote of all other non-meta players.

Names:

- Meta Majority: Original name by Karol Langner

static meta_strategy(*results*, *opponent*)
Determine the meta result based on results of all players. Override this function in child classes.

class axelrod.strategies.meta.**MetaMajorityFiniteMemory**
MetaMajority with the team of Finite Memory Players

Names

- Meta Majority Finite Memory: Original name by Marc Harper

class axelrod.strategies.meta.**MetaMajorityLongMemory**
MetaMajority with the team of Long (infinite) Memory Players

Names

- Meta Majority Long Memory: Original name by Marc Harper

class axelrod.strategies.meta.**MetaMajorityMemoryOne**
MetaMajority with the team of Memory One players

Names

- Meta Majority Memory One: Original name by Marc Harper

class axelrod.strategies.meta.**MetaMinority**(*team=None*)
A player who goes by the minority vote of all other non-meta players.

Names:

- Meta Minority: Original name by Karol Langner

static meta_strategy(*results*, *opponent*)
Determine the meta result based on results of all players. Override this function in child classes.

class axelrod.strategies.meta.**MetaMixer**(*team=None*, *distribution=None*)
A player who randomly switches between a team of players. If no distribution is passed then the player will uniformly choose between sub players.

In essence this is creating a Mixed strategy.

Parameters

team [list of strategy classes, optional] Team of strategies that are to be randomly played If none is passed will select the ordinary strategies.

distribution [list representing a probability distribution, optional] This gives the distribution from which to select the players. If none is passed will select uniformly.

Names

- Meta Mixer: Original name by Vince Knight

index_strategy(*opponent*)
When the team effectively has a single player, only use that strategy.

meta_strategy(*results*, *opponent*)
Using the _random.choice function to sample with weights.

class axelrod.strategies.meta.**MetaPlayer**(*team=None*)
A generic player that has its own team of players.

Names:

- Meta Player: Original name by Karol Langner

meta_strategy(*results*, *opponent*)
Determine the meta result based on results of all players. Override this function in child classes.

set_seed(*seed=None*)
Set a random seed for the player's random number generator.

strategy(*opponent*)
Actual strategy definition that determines player's action.

class axelrod.strategies.meta.**MetaWinner**(*team=None*)
A player who goes by the strategy of the current winner.

Names:

- Meta Winner: Original name by Karol Langner

meta_strategy(*results*, *opponent*)
Determine the meta result based on results of all players. Override this function in child classes.

class axelrod.strategies.meta.**MetaWinnerDeterministic**
Meta Winner with the team of Deterministic Players.

Names

- Meta Winner Deterministic: Original name by Marc Harper

class axelrod.strategies.meta.**MetaWinnerEnsemble**(*team=None*)
A variant of MetaWinner that chooses one of the top scoring strategies at random against each opponent. Note this strategy is always stochastic regardless of the team, if team larger than 1, and the players are distinct.

Names:

- Meta Winner Ensemble: Original name by Marc Harper

meta_strategy(*results*, *opponent*)
Determine the meta result based on results of all players. Override this function in child classes.

class axelrod.strategies.meta.**MetaWinnerFiniteMemory**
MetaWinner with the team of Finite Memory Players

Names

- Meta Winner Finite Memory: Original name by Marc Harper

class axelrod.strategies.meta.**MetaWinnerLongMemory**
MetaWinner with the team of Long (infinite) Memory Players

Names

- Meta Winner Long Memory: Original name by Marc Harper

class axelrod.strategies.meta.**MetaWinnerMemoryOne**
MetaWinner with the team of Memory One players

Names

- Meta Winner Memory Memory One: Original name by Marc Harper

class axelrod.strategies.meta.**MetaWinnerStochastic**
Meta Winner with the team of Stochastic Players.

Names

- Meta Winner Stochastic: Original name by Marc Harper

class axelrod.strategies.meta.**NMWEDeterministic**
Nice Meta Winner Ensemble with the team of Deterministic Players.

Names

- Nice Meta Winner Ensemble Deterministic: Original name by Marc Harper

class axelrod.strategies.meta.**NMWEFiniteMemory**
Nice Meta Winner Ensemble with the team of Finite Memory Players.

Names

- Nice Meta Winner Ensemble Finite Memory: Original name by Marc Harper

class axelrod.strategies.meta.**NMWELongMemory**
Nice Meta Winner Ensemble with the team of Long Memory Players.

Names

- Nice Meta Winner Ensemble Long Memory: Original name by Marc Harper

class axelrod.strategies.meta.**NMWEMemoryOne**
Nice Meta Winner Ensemble with the team of Memory One Players.

Names

- Nice Meta Winner Ensemble Memory One: Original name by Marc Harper

class axelrod.strategies.meta.**NMWEStochastic**
Nice Meta Winner Ensemble with the team of Stochastic Players.

Names

- Nice Meta Winner Ensemble Stochastic: Original name by Marc Harper

class axelrod.strategies.meta.**NiceMetaWinner**(*team=None*)
A player who goes by the strategy of the current winner.

Names:

- Meta Winner: Original name by Karol Langner

original_class
alias of *MetaWinner*

class axelrod.strategies.meta.**NiceMetaWinnerEnsemble**(*team=None*)
A variant of MetaWinner that chooses one of the top scoring strategies at random against each opponent. Note this strategy is always stochastic regardless of the team, if team larger than 1, and the players are distinct.

Names:

- Meta Winner Ensemble: Original name by Marc Harper

original_class
alias of *MetaWinnerEnsemble*

class axelrod.strategies.mutual.**Desperate**
A player that only cooperates after mutual defection.

Names:

- Desperate: [Berg2015]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.mutual.**Hopeless**
A player that only defects after mutual cooperation.

Names:

- Hopeless: [Berg2015]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.mutual.**Willing**
A player that only defects after mutual defection.

Names:

- Willing: [Berg2015]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.negation.**Negation**
A player starts by cooperating or defecting randomly if it's their first move, then simply doing the opposite of the opponents last move thereafter.

Names:

- Negation: [PD2017]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.oncebitten.**FoolMeOnce**
Forgives one D then retaliates forever on a second D.

Names:

- Fool me once: Original name by Marc Harper

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.oncebitten.**ForgetfulFoolMeOnce**(*forget_probability: float = 0.05*)
Forgives one D then retaliates forever on a second D. Sometimes randomly forgets the defection count, and so keeps a secondary count separate from the standard count in Player.

Names:

- Forgetful Fool Me Once: Original name by Marc Harper

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.oncebitten.**OnceBitten**
Cooperates once when the opponent defects, but if they defect twice in a row defaults to forgetful grudger for 10 turns defecting.

Names:

- Once Bitten: Original name by Holly Marissa

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Begins by playing C, then plays D for mem_length rounds if the opponent ever plays D twice in a row.

class axelrod.strategies.prober.**CollectiveStrategy**
Defined in [Li2009]. 'It always cooperates in the first move and defects in the second move. If the opponent also cooperates in the first move and defects in the second move, CS will cooperate until the opponent defects. Otherwise, CS will always defect.'

Names:

- Collective Strategy: [Li2009]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.prober.**Detective**(*initial_actions: List[axelrod.action.Action] = None*)
Starts with C, D, C, C, or with the given sequence of actions. If the opponent defects at least once in the first fixed rounds, play as TFT forever, else defect forever.

Names:

- Detective: [NC2019]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.prober.**HardProber**
Plays D, D, C, C initially. Defects forever if opponent cooperated in moves 2 and 3. Otherwise plays TFT.

Names:

- Hard Prober: [Prison1998]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.prober.**NaiveProber**(*p: float = 0.1*)
Like tit-for-tat, but it occasionally defects with a small probability.

Names:

- Naive Prober: [Li2011]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.prober.**Prober**
Plays D, C, C initially. Defects forever if opponent cooperated in moves 2 and 3. Otherwise plays TFT.

Names:

- Prober: [Li2011]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.prober.**Prober2**
Plays D, C, C initially. Cooperates forever if opponent played D then C in moves 2 and 3. Otherwise plays TFT.

Names:

- Prober 2: [Prison1998]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.prober.**Prober3**
Plays D, C initially. Defects forever if opponent played C in moves 2. Otherwise plays TFT.

Names:

- Prober 3: [Prison1998]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.prober.**Prober4**
Plays C, C, D, C, D, D, D, C, C, D, C, D, C, C, D, C, D, D, C, D initially. Counts retaliating and provocative defections of the opponent. If the absolute difference between the counts is smaller or equal to 2, defects forever. Otherwise plays C for the next 5 turns and TFT for the rest of the game.

Names:

- Prober 4: [Prison1998]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.prober.**RemorsefulProber**(*p: float = 0.1*)
Like Naive Prober, but it remembers if the opponent responds to a random defection with a defection by being remorseful and cooperating.

For reference see: [Li2011]. A more complete description is given in "The Selfish Gene" (https://books.google.co.uk/books?id=ekonDAAAQBAJ):

"Remorseful Prober remembers whether it has just spontaneously defected, and whether the result was prompt retaliation. If so, it 'remorsefully' allows its opponent 'one free hit' without retaliating."

Names:

- Remorseful Prober: [Li2011]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.punisher.**InversePunisher**
An inverted version of Punisher. The player starts by cooperating however will defect if at any point the opponent has defected, and forgets after mem_length matches, with 1 <= mem_length <= 20. This time mem_length is proportional to the amount of time the opponent has played C.

Names:

- Inverse Punisher: Original name by Geraint Palmer

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Begins by playing C, then plays D for an amount of rounds proportional to the opponents historical '%' of playing C if the opponent ever plays D.

class axelrod.strategies.punisher.**LevelPunisher**
A player starts by cooperating however, after 10 rounds will defect if at any point the number of defections by an opponent is greater than 20%.

Names:

- Level Punisher: [Eckhart2015]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.punisher.**Punisher**
A player starts by cooperating however will defect if at any point the opponent has defected, but forgets after meme_length matches, with 1<=mem_length<=20 proportional to the amount of time the opponent has played D, punishing that player for playing D too often.

Names:

- Punisher: Original name by Geraint Palmer

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Begins by playing C, then plays D for an amount of rounds proportional to the opponents historical '%' of playing D if the opponent ever plays D

class axelrod.strategies.punisher.**TrickyLevelPunisher**
A player starts by cooperating however, after 10, 50 and 100 rounds will defect if at any point the percentage of defections by an opponent is greater than 20%, 10% and 5% respectively.

Names:

- Tricky Level Punisher: [Eckhart2015]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.qlearner.**ArrogantQLearner**
A player who learns the best strategies through the q-learning algorithm.

This Q learner jumps to quick conclusions and cares about the future.

Names:

- Arrogant Q Learner: Original name by Geraint Palmer

class axelrod.strategies.qlearner.**CautiousQLearner**
A player who learns the best strategies through the q-learning algorithm.

This Q learner is slower to come to conclusions and wants to look ahead more.

Names:

- Cautious Q Learner: Original name by Geraint Palmer

class axelrod.strategies.qlearner.**HesitantQLearner**
A player who learns the best strategies through the q-learning algorithm.

This Q learner is slower to come to conclusions and does not look ahead much.

Names:

- Hesitant Q Learner: Original name by Geraint Palmer

class axelrod.strategies.qlearner.**RiskyQLearner**
A player who learns the best strategies through the q-learning algorithm.

This Q learner is quick to come to conclusions and doesn't care about the future.

Names:

- Risky Q Learner: Original name by Geraint Palmer

find_reward(*opponent: axelrod.player.Player*) → Dict[axelrod.action.Action, Dict[axelrod.action.Action, Union[int, float]]]
Finds the reward gained on the last iteration

find_state(*opponent: axelrod.player.Player*) → str
Finds the my_state (the opponents last n moves + its previous proportion of playing C) as a hashable state

perform_q_learning(*prev_state: str*, *state: str*, *action: axelrod.action.Action*, *reward*)
Performs the qlearning algorithm

select_action(*state: str*) → axelrod.action.Action
Selects the action based on the epsilon-soft policy

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Runs a qlearn algorithm while the tournament is running.

class axelrod.strategies.rand.**Random**(*p: float = 0.5*)
A player who randomly chooses between cooperating and defecting.

This strategy came 15th in Axelrod's original tournament.

Names:

- Random: [Axelrod1980]
- Lunatic: [Tzafestas2000]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.resurrection.**DoubleResurrection**
A player starts by cooperating and defects if the number of rounds played by the player is greater than five and the last five rounds are cooperations.

If the last five rounds were defections, the player cooperates.

Names:

- DoubleResurrection: [Eckhart2015]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.resurrection.**Resurrection**
A player starts by cooperating and defects if the number of rounds played by the player is greater than five and the last five rounds are defections.

Otherwise, the strategy plays like Tit-for-tat.

Names:

- Resurrection: [Eckhart2015]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.retaliate.**LimitedRetaliate**(*retaliation_threshold: float = 0.1*, *retaliation_limit: int = 20*)
A player that co-operates unless the opponent defects and wins. It will then retaliate by defecting. It stops when either, it has beaten the opponent 10 times more often that it has lost or it reaches the retaliation limit (20 defections).

Names:

- Limited Retaliate: Original name by Owen Campbell

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
If the opponent has played D to my C more often than x% of the time that I've done the same to him, retaliate by playing D but stop doing so once I've hit the retaliation limit.

class axelrod.strategies.retaliate.**LimitedRetaliate2**(*retaliation_threshold: float = 0.08*, *retaliation_limit: int = 15*)
LimitedRetaliate player with a threshold of 8 percent and a retaliation limit of 15.

Names:

- Limited Retaliate 2: Original name by Owen Campbell

class axelrod.strategies.retaliate.**LimitedRetaliate3**(*retaliation_threshold: float = 0.05*, *retaliation_limit: int = 20*)

LimitedRetaliate player with a threshold of 5 percent and a retaliation limit of 20.

Names:

- Limited Retaliate 3: Original name by Owen Campbell

class axelrod.strategies.retaliate.**Retaliate**(*retaliation_threshold: float = 0.1*)

A player starts by cooperating but will retaliate once the opponent has won more than 10 percent times the number of defections the player has.

Names:

- Retaliate: Original name by Owen Campbell

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action

If the opponent has played D to my C more often than x% of the time that I've done the same to him, play D. Otherwise, play C.

class axelrod.strategies.retaliate.**Retaliate2**(*retaliation_threshold: float = 0.08*)

Retaliate player with a threshold of 8 percent.

Names:

- Retaliate 2: Original name by Owen Campbell

class axelrod.strategies.retaliate.**Retaliate3**(*retaliation_threshold: float = 0.05*)

Retaliate player with a threshold of 5 percent.

Names:

- Retaliate 3: Original name by Owen Campbell

Revised Downing implemented from the Fortran source code for the second of Axelrod's tournaments.

class axelrod.strategies.revised_downing.**RevisedDowning**

Strategy submitted to Axelrod's second tournament by Leslie Downing. (K59R).

Revised Downing attempts to determine if players are cooperative or not. If so, it cooperates with them.

This strategy is a revision of the strategy submitted by Downing to Axelrod's first tournament.

Names: - Revised Downing: [Axelrod1980]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action

This is a placeholder strategy.

class axelrod.strategies.sequence_player.**SequencePlayer**(*generator_function: function*, *generator_args: Tuple = ()*)

Abstract base class for players that use a generated sequence to determine their plays.

Names:

- Sequence Player: Original name by Marc Harper

static meta_strategy(*value: int*) → axelrod.action.Action

Determines how to map the sequence value to cooperate or defect. By default, treat values like python truth values. Override in child classes for alternate behaviors.

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action

Iterate through the sequence and apply the meta strategy.

class axelrod.strategies.sequence_player.**ThueMorse**
A player who cooperates or defects according to the Thue-Morse sequence. The first few terms of the Thue-Morse sequence are: 0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0 . . .

Thue-Morse sequence: http://mathworld.wolfram.com/Thue-MorseSequence.html

Names:

- Thue Morse: Original name by Geraint Palmer

class axelrod.strategies.sequence_player.**ThueMorseInverse**
A player who plays the inverse of the Thue-Morse sequence.

Names:

- Inverse Thue Morse: Original name by Geraint Palmer

static meta_strategy(*value: int*) → axelrod.action.Action
Determines how to map the sequence value to cooperate or defect. By default, treat values like python truth values. Override in child classes for alternate behaviors.

class axelrod.strategies.shortmem.**ShortMem**
A player starts by always cooperating for the first 10 moves.

From the tenth round on, the player analyzes the last ten actions, and compare the number of defects and cooperates of the opponent, based in percentage. If cooperation occurs 30% more than defection, it will cooperate. If defection occurs 30% more than cooperation, the program will defect. Otherwise, the program follows the TitForTat algorithm.

Names:

- ShortMem: [Andre2013]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.selfsteem.**SelfSteem**
This strategy is based on the feeling with the same name. It is modeled on the sine curve(f = sin(2* pi * n / 10)), which varies with the current iteration.

If f > 0.95, 'ego' of the algorithm is inflated; always defects. If 0.95 > abs(f) > 0.3, rational behavior; follows TitForTat algortithm. If 0.3 > f > -0.3; random behavior. If f < -0.95, algorithm is at rock bottom; always cooperates.

Futhermore, the algorithm implements a retaliation policy, if the opponent defects; the sin curve is shifted. But due to lack of further information, this implementation does not include a sin phase change. Names:

- SelfSteem: [Andre2013]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.stalker.**Stalker**
This is a strategy which is only influenced by the score. Its behavior is based on three values: the very_bad_score (all rounds in defection) very_good_score (all rounds in cooperation) wish_score (average between bad and very_good score)

It starts with cooperation.

- If current_average_score > very_good_score, it defects
- If current_average_score lies in (wish_score, very_good_score) it cooperates
- If current_average_score > 2, it cooperates

- If current_average_score lies in (1, 2)
- The remaining case, current_average_score < 1, it behaves randomly.
- It defects in the last round

Names:

- Stalker: [Andre2013]

original_class
alias of *Stalker*

class axelrod.strategies.titfortat.**AdaptiveTitForTat**(*rate: float = 0.5*)
ATFT - Adaptive Tit For Tat (Basic Model)

Algorithm

if (opponent played C in the last cycle) then world = world + r*(1-world) else world = world + r*(0-world) If (world >= 0.5) play C, else play D

Attributes

world [float [0.0, 1.0], set to 0.5] continuous variable representing the world's image 1.0 - total cooperation 0.0 - total defection other values - something in between of the above updated every round, starting value shouldn't matter as long as it's >= 0.5

Parameters

rate [float [0.0, 1.0], default=0.5] adaptation rate - r in Algorithm above smaller value means more gradual and robust to perturbations behaviour

Names:

- Adaptive Tit For Tat: [Tzafestas2000]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**Alexei**
Plays similar to Tit-for-Tat, but always defect on last turn.

Names:

- Alexei: [LessWrong2011]

original_class
alias of *Alexei*

class axelrod.strategies.titfortat.**AntiTitForTat**
A strategy that plays the opposite of the opponents previous move. This is similar to Bully, except that the first move is cooperation.

Names:

- Anti Tit For Tat: [Hilbe2013]
- Psycho (PSYC): [Ashlock2009]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**Bully**
A player that behaves opposite to Tit For Tat, including first move.

Starts by defecting and then does the opposite of opponent's previous move. This is the complete opposite of Tit For Tat, also called Bully in the literature.

Names:

- Reverse Tit For Tat: [Nachbar1992]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**BurnBothEnds**
Plays like TitForTat except it cooperates after opponent cooperation with probability 9/10.

Names:

- BurnBothEnds (BBE): Marinoff [Marinoff1992]

strategy(*opponent*)
This is a placeholder strategy.

class axelrod.strategies.titfortat.**ContriteTitForTat**
A player that corresponds to Tit For Tat if there is no noise. In the case of a noisy match: if the opponent defects as a result of a noisy defection then ContriteTitForTat will become 'contrite' until it successfully cooperates.

Names:

- Contrite Tit For Tat: [Axelrod1995]

original_class
alias of *ContriteTitForTat*

class axelrod.strategies.titfortat.**DynamicTwoTitsForTat**
A player starts by cooperating and then punishes its opponent's defections with defections, but with a dynamic bias towards cooperating based on the opponent's ratio of cooperations to total moves (so their current probability of cooperating regardless of the opponent's move (aka: forgiveness)).

Names:

- Dynamic Two Tits For Tat: Original name by Grant Garrett-Grossman.

strategy(*opponent*)
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**EugineNier**
Plays similar to Tit-for-Tat, but with two conditions: 1) Always Defect on Last Move 2) If other player defects five times, switch to all defects.

Names:

- Eugine Nier: [LessWrong2011]

original_class
alias of *EugineNier*

class axelrod.strategies.titfortat.**Gradual**
Similar to OriginalGradual, this is a player that punishes defections with a growing number of defections but after punishing for *punishment_limit* number of times enters a calming state and cooperates no matter what the opponent does for two rounds.

This version of Gradual is an update of *OriginalGradual* and the difference is that the *punishment_limit* is incremented whenever the opponent defects (regardless of the state of the player).

Note that this version of *Gradual* appears in [CRISTAL-SMAC2018] however this version of *Gradual* does not give the results reported in [Beaufils1997] which is the paper that first introduced the strategy. For a longer discussion of this see: https://github.com/Axelrod-Python/Axelrod/issues/1294.

This version is based on https://github.com/cristal-smac/ipd/blob/master/src/strategies.py#L224

Names:

- Gradual: [CRISTAL-SMAC2018]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**HardTitFor2Tats**
A variant of Tit For Two Tats that uses a longer history for retaliation.

Names:

- Hard Tit For Two Tats: [Stewart2012]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**HardTitForTat**
A variant of Tit For Tat that uses a longer history for retaliation.

Names:

- Hard Tit For Tat: [PD2017]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**Michaelos**
Plays similar to Tit-for-Tat with two exceptions: 1) Defect on last turn. 2) After own defection and opponent's cooperation, 50 percent of the time, cooperate. The other 50 percent of the time, always defect for the rest of the game.

Names:

- Michaelos: [LessWrong2011]

original_class
alias of *Michaelos*

class axelrod.strategies.titfortat.**NTitsForMTats**(*N: int = 3*, *M: int = 2*)
A parameterizable Tit-for-Tat, The arguments are: 1) M: the number of defection before retaliation 2) N: the number of retaliations

Names:

- N Tit(s) For M Tat(s): Original name by Marc Harper

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**OmegaTFT**(*deadlock_threshold: int = 3*, *randomness_threshold: int = 8*)
OmegaTFT modifies Tit For Tat in two ways: - checks for deadlock loops of alternating rounds of (C, D) and (D, C), and attempting to break them - uses a more sophisticated retaliation mechanism that is noise tolerant

Names:

- OmegaTFT: [Slany2007]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**OriginalGradual**
A player that punishes defections with a growing number of defections but after punishing for *punishment_limit* number of times enters a calming state and cooperates no matter what the opponent does for two rounds.

The *punishment_limit* is incremented whenever the opponent defects and the strategy is not in either calming or punishing state.

Note that *Gradual* appears in [CRISTAL-SMAC2018] however that version of *Gradual* does not give the results reported in [Beaufils1997] which is the paper that first introduced the strategy. For a longer discussion of this see: https://github.com/Axelrod-Python/Axelrod/issues/1294. This is why this strategy has been renamed to *OriginalGradual*.

Names:

- Gradual: [Beaufils1997]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**RandomTitForTat**(*p: float = 0.5*)
A player starts by cooperating and then follows by copying its opponent (tit for tat style). From then on the player will switch between copying its opponent and randomly responding every other iteration.

Name:

- Random TitForTat: Original name by Zachary M. Taylor

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is the actual strategy

class axelrod.strategies.titfortat.**SlowTitForTwoTats2**
A player plays C twice, then if the opponent plays the same move twice, plays that move, otherwise plays previous move.

Names:

- Slow Tit For Tat: [Prison1998]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**SneakyTitForTat**
Tries defecting once and repents if punished.

Names:

- Sneaky Tit For Tat: Original name by Karol Langner

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**SpitefulTitForTat**
A player starts by cooperating and then mimics the previous action of the opponent until opponent defects twice in a row, at which point player always defects

Names:

- Spiteful Tit For Tat: [Prison1998]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**SuspiciousTitForTat**
A variant of Tit For Tat that starts off with a defection.

Names:

- Suspicious Tit For Tat: [Hilbe2013]
- Mistrust: [Beaufils1997]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**TitFor2Tats**
A player starts by cooperating and then defects only after two defects by opponent.

Submitted to Axelrod's second tournament by John Maynard Smith; it came in 24th in that tournament.

Names:

- Tit for two Tats: [Axelrod1984]
- Slow tit for two tats: Original name by Ranjini Das
- JMaynardSmith: [Axelrod1980b]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.titfortat.**TitForTat**
A player starts by cooperating and then mimics the previous action of the opponent.

This strategy was referred to as the *'simplest'* strategy submitted to Axelrod's first tournament. It came first.

Note that the code for this strategy is written in a fairly verbose way. This is done so that it can serve as an example strategy for those who might be new to Python.

Names:

- Rapoport's strategy: [Axelrod1980]
- TitForTat: [Axelrod1980]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
This is the actual strategy

class axelrod.strategies.titfortat.**TwoTitsForTat**
A player starts by cooperating and replies to each defect by two defections.

Names:

- Two Tits for Tats: [Axelrod1984]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.verybad.**VeryBad**
It cooperates in the first three rounds, and uses probability (it implements a memory, which stores the opponent's moves) to decide for cooperating or defecting. Due to a lack of information as to what that probability refers to in this context, probability(P(X)) refers to (Count(X)/Total_Moves) in this implementation P(C) = Cooperations / Total_Moves P(D) = Defections / Total_Moves = 1 - P(C)

Names:

- VeryBad: [Andre2013]

static strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.worse_and_worse.**KnowledgeableWorseAndWorse**
This strategy is based on 'Worse And Worse' but will defect with probability of 'current turn / total no. of turns'.

Names:

- Knowledgeable Worse and Worse: Original name by Adam Pohl

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.worse_and_worse.**WorseAndWorse**
Defects with probability of 'current turn / 1000'. Therefore it is more and more likely to defect as the round goes on.

Source code available at the download tab of [Prison1998]

Names:

- Worse and Worse: [Prison1998]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.worse_and_worse.**WorseAndWorse2**
Plays as tit for tat during the first 20 moves. Then defects with probability (current turn - 20) / current turn. Therefore it is more and more likely to defect as the round goes on.

Names:

- Worse and Worse 2: [Prison1998]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.worse_and_worse.**WorseAndWorse3**
Cooperates in the first turn. Then defects with probability no. of opponent defects / (current turn - 1). Therefore it is more likely to defect when the opponent defects for a larger proportion of the turns.

Names:

- Worse and Worse 3: [Prison1998]

strategy(*opponent: axelrod.player.Player*) → axelrod.action.Action
Actual strategy definition that determines player's action.

class axelrod.strategies.zero_determinant.**LRPlayer**(*phi: float = 0.2*, *s: float = 0.1*, *l: float = 1*)
Abstraction for Linear Relation players. These players enforce a linear difference in stationary payoffs $s(S_{xy} - l) = S_{yx} - l$.

The parameter s is called the slope and the parameter l the baseline payoff. For extortionate strategies, the extortion factor χ is the inverse of the slope s.

For the standard prisoner's dilemma where $T > R > P > S$ and $R > (T + S)/2 > P$, a pair (l, s) is enforceable iff

$$P <= l <= R \tag{2.1}$$

$$s_{min} = -\min\left(\frac{T-l}{l-S}, \frac{l-S}{T-l}\right) <= s <= 1 \tag{2.2}$$

And also that there exists ϕ such that

$$p_1 = P(C|CC) = 1 - \phi(1-s)(R-l) \tag{2.3}$$

$$p_2 = P(C|CD) = 1 - \phi(s(l-S) + (T-l)) \tag{2.4}$$

$$p_3 = P(C|DC) = \phi((l-S) + s(T-l)) \tag{2.5}$$

$$p_4 = P(C|DD) = \phi(1-s)(l-P) \tag{2.6}$$

These conditions also force $\phi >= 0$. For a given pair (l, s) there may be multiple such ϕ.

This parameterization is Equation 14 in [Hilbe2013]. See Figure 2 of the article for a more in-depth explanation. Other game parameters can alter the relations and bounds above.

Names:

- Linear Relation player: [Hilbe2013]

receive_match_attributes()
Parameters

phi, s, l: floats Parameter used to compute the four-vector according to the parameterization of the strategies below.

class axelrod.strategies.zero_determinant.**ZDExtort2**(*phi: float = 0.1111111111111111*, *s: float = 0.5*)

An Extortionate Zero Determinant Strategy with l=P.

Names:

- Extort-2: [Stewart2012]

receive_match_attributes()
Parameters

phi, s, l: floats Parameter used to compute the four-vector according to the parameterization of the strategies below.

class axelrod.strategies.zero_determinant.**ZDExtort2v2**(*phi: float = 0.125*, *s: float = 0.5*, *l: float = 1*)

An Extortionate Zero Determinant Strategy with l=1.

Names:

- EXTORT2: [Kuhn2017]

class axelrod.strategies.zero_determinant.**ZDExtort3**(*phi: float = 0.11538461538461539*, *s: float = 0.3333333333333333*, *l: float = 1*)

An extortionate strategy from Press and Dyson's paper witn an extortion factor of 3.

Names:

- ZDExtort3: Original name by Marc Harper
- Unnamed: [Press2012]

class axelrod.strategies.zero_determinant.**ZDExtort4**(*phi: float = 0.23529411764705882*, *s: float = 0.25*, *l: float = 1*)

An Extortionate Zero Determinant Strategy with l=1, s=1/4. TFT is the other extreme (with l=3, s=1)

Names:

- Extort 4: Original name by Marc Harper

class axelrod.strategies.zero_determinant.**ZDExtortion**(*phi: float = 0.2*, *s: float = 0.1*, *l: float = 1*)

An example ZD Extortion player.

Names:

- ZDExtortion: [Roemheld2013]

class axelrod.strategies.zero_determinant.**ZDGTFT2**(*phi: float = 0.25*, *s: float = 0.5*)
A Generous Zero Determinant Strategy with l=R.

Names:

- ZDGTFT-2: [Stewart2012]

receive_match_attributes()
Parameters

phi, s, l: floats Parameter used to compute the four-vector according to the parameterization of the strategies below.

class axelrod.strategies.zero_determinant.**ZDGen2**(*phi: float = 0.125*, *s: float = 0.5*, *l: float = 3*)
A Generous Zero Determinant Strategy with l=3.

Names:

- GEN2: [Kuhn2017]

class axelrod.strategies.zero_determinant.**ZDMischief**(*phi: float = 0.1*, *s: float = 0.0*, *l: float = 1*)
An example ZD Mischief player.

Names:

- ZDMischief: [Roemheld2013]

class axelrod.strategies.zero_determinant.**ZDSet2**(*phi: float = 0.25*, *s: float = 0.0*, *l: float = 2*)
A Generous Zero Determinant Strategy with l=2.

Names:

- SET2: [Kuhn2017]

2.4.3 Glossary

There are a variety of terms used in the documentation and throughout the library. Here is an overview:

An action

An **action** is either `C` or `D`. You can access these actions as follows but should not really have a reason to:

```
>>> import axelrod as axl
>>> axl.Action.C
C
>>> axl.Action.D
D
```

A play

A **play** is a single player choosing an **action**. In terms of code this is equivalent to:

```
>>> p1, p2 = axl.Cooperator(), axl.Defector()
>>> C = p1.strategy(p2)  # This constitutes two 'plays' (p1 plays and p2 plays).
>>> D = p2.strategy(p1)  # This constitutes two 'plays' (p1 plays and p2 plays).
```

A turn

A **turn** is a 1 shot interaction between two players. It is in effect a composition of two **plays**.

Each turn has four possible outcomes of a play: `(C, C)`, `(C, D)`, `(D, C)`, or `(D, D)`.

A match

A **match** is a consecutive number of **turns**. The default number of turns used in the tournament is 200. Here is a single match between two players over 3 turns:

```
>>> p1, p2 = axl.Cooperator(), axl.Defector()
>>> match = axl.Match((p1, p2), turns=3)
>>> result = match.play()
>>> result
[(C, D), (C, D), (C, D)]
>>> p1.history, p2.history
([C, C, C], [D, D, D])
```

A win

A **win** is attributed to the player who has the higher total score at the end of a match. For the example above, `Defector` would win that match.

A strategy

A **strategy** is a set of instructions that dictate how to play given one's own strategy and the strategy of an opponent. In the library these correspond to the strategy classes: *TitForTat*, *Grudger*, *Cooperator* etc. . .

When appropriate to do so this will be used interchangeable with *A player*.

A player

A **player** is a single agent using a given strategy. Players are the participants of tournament, usually they each represent one strategy but of course you can have multiple players choosing the same strategy. In the library these correspond to __instances__ of classes:

```
>>> p1, p2 = axl.Cooperator(), axl.Defector()
>>> p1
Cooperator
>>> p2
Defector
```

When appropriate to do so this will be used interchangeable with *A strategy*.

A round robin

A **round robin** is the set of all potential (order invariant) matches between a given collection of players.

A tournament

A **tournament** is a repetition of round robins so as to smooth out stochastic effects.

Noise

A match or tournament can be played with **noise**: this is the probability that indicates the chance of an action dictated by a strategy being swapped.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[Adami2013] Adami C and Hintze A. (2013) Evolutionary instability of zero-determinant strategies demonstrates that winning is not everything. Nature communications. https://www.nature.com/articles/ncomms3193

[Akin2015] Akin, Ethan. "What you gotta know to play good in the Iterated Prisoner's Dilemma." Games 6.3 (2015): 175-190.

[Amaral2016] Amaral, M. A., Wardil, L., Perc, M., & Da Silva, J. K. L. (2016). Stochastic win-stay-lose-shift strategy with dynamic aspirations in evolutionary social dilemmas. Physical Review E - Statistical, Nonlinear, and Soft Matter Physics, 94(3), 1–9. https://doi.org/10.1103/PhysRevE.94.032317

[Andre2013] Andre L. C., Honovan P., Felipe T. and Frederico G. (2013). Iterated Prisoner's Dilemma - An extended analysis, http://abricom.org.br/wp-content/uploads/2016/03/bricsccicbic2013_submission_202.pdf

[Ashlock2006] Ashlock, D., & Kim E. Y, & Leahy, N. (2006). Understanding Representational Sensitivity in the Iterated Prisoner's Dilemma with Fingerprints. IEEE Transactions On Systems, Man, And Cybernetics, Part C: Applications And Reviews, 36 (4)

[Ashlock2006b] Ashlock, W. & Ashlock, D. (2006). Changes in Prisoner's Dilemma Strategies Over Evolutionary Time With Different Population Sizes 2006 IEEE International Conference on Evolutionary Computation. http://DOI.org/10.1109/CEC.2006.1688322

[Ashlock2008] Ashlock, D., & Kim, E. Y. (2008). Fingerprinting: Visualization and automatic analysis of prisoner's dilemma strategies. IEEE Transactions on Evolutionary Computation, 12(5), 647–659. http://doi.org/10.1109/TEVC.2008.920675

[Ashlock2009] Ashlock, D., Kim, E. Y., & Ashlock, W. (2009) Fingerprint analysis of the noisy prisoner's dilemma using a finite-state representation. IEEE Transactions on Computational Intelligence and AI in Games. 1(2), 154-167 http://doi.org/10.1109/TCIAIG.2009.2018704

[Ashlock2014] Ashlock, W., Tsang, J. & Ashlock, D. (2014) The evolution of exploitation. 2014 IEEE Symposium on Foundations of Computational Intelligence (FOCI) http://DOI.org/10.1109/FOCI.2014.7007818

[Ashlock2015] Ashlock, D., Brown, J.A., & Hingston P. (2015). Multiple Opponent Optimization of Prisoner's Dilemma Playing Agents. Multiple Opponent Optimization of Prisoner's Dilemma Playing Agents http://DOI.org/10.1109/TCIAIG.2014.2326012

[Au2006] Au, T.-C. and Nau, D. S. (2006) Accident or intention: That is the question (in the iterated prisoner's dilemma). In Proc. Int. Conf. Auton. Agents and Multiagent Syst. (AAMAS), pp. 561–568. http://www.cs.umd.edu/~nau/papers/au2006accident.pdf

[Axelrod1980] Axelrod, R. (1980). Effective Choice in the Prisoner's Dilemma. Journal of Conflict Resolution, 24(1), 3–25.

[Axelrod1980b] Axelrod, R. (1980). More Effective Choice in the Prisoner's Dilemma. Journal of Conflict Resolution, 24(3), 379-403.

[Axelrod1984] The Evolution of Cooperation. Basic Books. ISBN 0-465-02121-2.

[Axelrod1995] Wu, J. and Axelrod, R. (1995). How to cope with noise in the Iterated prisoner's dilemma, Journal of Conflict Resolution, 39(1), pp. 183–189. doi: 10.1177/0022002795039001008.

[Banks1990] Banks, J. S., & Sundaram, R. K. (1990). Repeated games, finite automata, and complexity. Games and Economic Behavior, 2(2), 97–117. http://doi.org/10.1016/0899-8256(90)90024-O

[Bendor1993] Bendor, Jonathan. "Uncertainty and the Evolution of Cooperation." The Journal of Conflict Resolution, 37(4), 709–734.

[Beaufils1997] Beaufils, B. & Delahaye, J. & Mathieu, P. (1997). Our Meeting With Gradual: A Good Strategy For The Iterated Prisoner's Dilemma. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.4041

[Berg2015] Berg, P. Van Den, & Weissing, F. J. (2015). The importance of mechanisms for the evolution of cooperation. Proceedings of the Royal Society B-Biological Sciences, 282.

[CRISTAL-SMAC2018] CRISTAL Lab, SMAC Team, Lille University (2018). IPD : the Iterated Prisoner's Dilemma. https://github.com/cristal-smac/ipd

[Downing1975] Downing, Leslie L. "The Prisoner's Dilemma game as a problem-solving phenomenon: An outcome maximization interpretation." Simulation & Games 6.4 (1975): 366-391.

[Eckhart2015] Eckhart Arnold (2016) CoopSim v0.9.9 beta 6. https://github.com/jecki/CoopSim/

[Frean1994] Frean, Marcus R. "The Prisoner's Dilemma without Synchrony." Proceedings: Biological Sciences, vol. 257, no. 1348, 1994, pp. 75–79. www.jstor.org/stable/50253.

[Harper2017] Harper, M., Knight, V., Jones, M., Koutsovoulos, G., Glynatsi, N. E., & Campbell, O. (2017) Reinforcement learning produces dominant strategies for the Iterated Prisoner's Dilemma. PloS one. https://doi.org/10.1371/journal.pone.0188046

[Hauert2002] Hauert, Christoph, and Olaf Stenull. "Simple adaptive strategy wins the prisoner's dilemma." Journal of Theoretical Biology 218.3 (2002): 261-272.

[Hilbe2013] Hilbe, C., Nowak, M.A. and Traulsen, A. (2013). Adaptive dynamics of extortion and compliance, PLoS ONE, 8(11), p. e77886. doi: 10.1371/journal.pone.0077886.

[Hilbe2017] Hilbe, C., Martinez-Vaquero, L. A., Chatterjee K., Nowak M. A. (2017). Memory-n strategies of direct reciprocity, Proceedings of the National Academy of Sciences May 2017, 114 (18) 4715-4720; doi: 10.1073/pnas.1621239114.

[Kuhn2017] Kuhn, Steven, "Prisoner's Dilemma", The Stanford Encyclopedia of Philosophy (Spring 2017 Edition), Edward N. Zalta (ed.), https://plato.stanford.edu/archives/spr2017/entries/prisoner-dilemma/

[Kraines1989] Kraines, David, and Vivian Kraines. "Pavlov and the prisoner's dilemma." Theory and decision 26.1 (1989): 47-79. doi:10.1007/BF00134056

[Krapohl2020] Krapohl, S., Ocelík, V. & Walentek, D.M. The instability of globalization: applying evolutionary game theory to global trade cooperation. Public Choice 188, 31–51 (2021). https://doi.org/10.1007/s11127-020-00799-1

[LessWrong2011] Zoo of Strategies (2011) LessWrong. Available at: http://lesswrong.com/lw/7f2/prisoners_dilemma_tournament_results/

[Li2007] Li, J, How to Design a Strategy to Win an IPD Tournament, in Kendall G., Yao X. and Chong S. (eds.) The iterated prisoner's dilemma: 20 years on. World Scientific, chapter 4, pp. 29-40, 2007.

[Li2009] Li, J. & Kendall, G. (2009). A Strategy with Novel Evolutionary Features for the Iterated Prisoner's Dilemma. Evolutionary Computation 17(2): 257–274.

[Li2011] Li, J., Hingston, P., Member, S., & Kendall, G. (2011). Engineering Design of Strategies for Winning Iterated Prisoner ' s Dilemma Competitions, 3(4), 348–360.

[Li2014] Li, J. and Kendall, G. (2014). The Effect of Memory Size on the Evolutionary Stability of Strategies in Iterated Prisoner's Dilemma. IEEE Transactions on Evolutionary Computation, 18(6) 819-826

[LiS2014] Li, Siwei. (2014). Strategies in the Stochastic Iterated Prisoner's Dilemma. Available at: http://math.uchicago.edu/~may/REU2014/REUPapers/Li,Siwei.pdf

[Luis2008] Luis R. Izquierdo and Segismundo S. Izquierdo (2008). Dynamics of the Bush-Mosteller Learning Algorithm in 2x2 Games, Reinforcement Learning, Cornelius Weber, Mark Elshaw and Norbert Michael Mayer (Ed.), InTech, DOI: 10.5772/5282. Available from: https://www.intechopen.com/books/reinforcement_learning/dynamics_of_the_bush-mosteller_learning_algorithm_in_2x2_games

[Marinoff1992] Marinoff, Louis. (1992). Maximizing expected utilities in the prisoner's dilemma. Journal of Conflict Resolution 36.1: 183-216.

[Mathieu2015] Mathieu, P. and Delahaye, J. (2015). New Winning Strategies for the Iterated Prisoner's Dilemma. Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems.

[Mittal2009] Mittal, S., & Deb, K. (2009). Optimal strategies of the iterated prisoner's dilemma problem for multiple conflicting objectives. IEEE Transactions on Evolutionary Computation, 13(3), 554–565. https://doi.org/10.1109/TEVC.2008.2009459

[Murase2020] Murase, Y., & Baek, S.K. (2020). Five Rules for Friendly Rivalry in Direct Reciprocity. Scientific Reports 10:16904 https://doi.org/10.1038/s41598-020-73855-x

[Nachbar1992] Nachbar J., Evolution in the finitely repeated prisoner's dilemma, Journal of Economic Behavior & Organization, 19(3): 307-326, 1992.

[NC2019] https://github.com/ncase/trust (Accessed: 30 October 2019)

[Nowak1989] Nowak, Martin, and Karl Sigmund. "Game-dynamical aspects of the prisoner's dilemma." Applied Mathematics and Computation 30.3 (1989): 191-213.

[Nowak1990] Nowak, M., & Sigmund, K. (1990). The evolution of stochastic strategies in the Prisoner's Dilemma. Acta Applicandae Mathematica. https://link.springer.com/article/10.1007/BF00049570

[Nowak1992] Nowak, M.., & May, R. M. (1992). Evolutionary games and spatial chaos. Nature. http://doi.org/10.1038/359826a0

[Nowak1993] Nowak, M., & Sigmund, K. (1993). A strategy of win-stay, lose-shift that outperforms tit-for-tat in the Prisoner's Dilemma game. Nature, 364(6432), 56–58. http://doi.org/10.1038/364056a0

[Ohtsuki2006] Ohtsuki, Hisashi, et al. "A simple rule for the evolution of cooperation on graphs and social networks." Nature 441.7092 (2006): 502.

[PD2017] http://www.prisoners-dilemma.com/competition.html (Accessed: 6 June 2017). Archived at https://web.archive.org/web/20171227021632/http://www.prisoners-dilemma.com/competition.html

[Press2012] Press, W. H., & Dyson, F. J. (2012). Iterated Prisoner's Dilemma contains strategies that dominate any evolutionary opponent. Proceedings of the National Academy of Sciences, 109(26), 10409–10413. http://doi.org/10.1073/pnas.1206569109

[Prison1998] LIFL (1998) PRISON. Available at: http://www.lifl.fr/IPD/ipd.frame.html (Accessed: 19 September 2016).

[Robson1990] Robson, Arthur J. "Efficiency in evolutionary games: Darwin, Nash and the secret handshake." Journal of theoretical Biology 144.3 (1990): 379-396.

[Roemheld2013] Roemheld, Lars. "Evolutionary Extortion and Mischief: Zero Determinant strategies in iterated 2x2 games". Available at: https://arxiv.org/abs/1308.2576

[Singer-Clark2014] Singer-Clark, T. (2014). Morality Metrics On Iterated Prisoner's Dilemma Players.

[Shakarian2013] Shakarian, P., Roos, P. & Moores, G. A Novel Analytical Method for Evolutionary Graph Theory Problems.

[Slany2007] Slany W. and Kienreich W., On some winning strategies for the iterated prisoner's dilemma, in Kendall G., Yao X. and Chong S. (eds.) The iterated prisoner's dilemma: 20 years on. World Scientific, chapter 8, pp. 171-204, 2007.

[Stewart2012] Stewart, a. J., & Plotkin, J. B. (2012). Extortion and cooperation in the Prisoner's Dilemma. Proceedings of the National Academy of Sciences, 109(26), 10134–10135. http://doi.org/10.1073/pnas.1208087109

[Szabo2007] Szabó, G., & Fáth, G. (2007). Evolutionary games on graphs. Physics Reports, 446(4-6), 97–216. http://doi.org/10.1016/j.physrep.2007.04.004

[Gaudesi2016] Gaudesi, Marco, et al. "Exploiting evolutionary modeling to prevail in iterated prisoner's dilemma tournaments." IEEE Transactions on Computational Intelligence and AI in Games 8.3 (2016): 288-300.

[Tzafestas2000] Tzafestas, E. (2000). Toward adaptive cooperative behavior. From Animals to Animals: Proceedings of the 6th International Conference on the Simulation of Adaptive Behavior {(SAB-2000)}, 2, 334–340.

Python Module Index

a

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

S

U

V

W

Z