
Axelrod Documentation

Release 0.0.1

Vincent Knight

Sep 19, 2017

Contents

1 Quick start	3
2 Table of Contents	5
2.1 Tutorials	5
2.1.1 New to Game Theory and/or Python	5
2.1.2 Research topics	13
2.1.3 Further capabilities in the library	25
2.1.4 Contributing	45
2.2 Reference	54
2.2.1 Background to Axelrod's Tournament	54
2.2.2 Play Contexts and Generic Prisoner's Dilemma	55
2.2.3 Tournaments	55
2.2.4 Strategies index	62
2.2.5 Bibliography	118
2.2.6 Glossary	118
2.3 Community	119
2.3.1 Part of the team	120
2.3.2 Communication	120
2.3.3 Code of Conduct	120
2.4 Citing the library	121
3 Indices and tables	123
Bibliography	125
Python Module Index	129

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

- Over 100 strategies from the literature and some 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

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}),
 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 *Strategies 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.

Tutorials

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

Contents:

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 3.4 or greater. It will not run on Python 2.

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

```
$ pip install axelrod
```

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), (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), (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, 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),
↪ (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=0.7,
↪Wins=0.0, Initial_C_rate=1.0, CC_rate=...),
 Player(Rank=2, Name='Grudger', Median_score=2.3..., Cooperation_rating=0.7, 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 [Accessing tournament results](#).

Accessing the interactions

This tutorial will show you briefly how to access the detailed interaction results corresponding to the tournament.

To access the detailed interaction results we create a tournament as usual (see [Creating and running a simple tournament](#)) but indicate that we want to keep track of the interactions:

```
>>> import axelrod as axl
>>> players = [
...     axl.Cooperator(), axl.Defector(),
...     axl.TitForTat(), axl.Grudger()]
>>> tournament = axl.Tournament(players, turns=3, repetitions=1)
>>> results = tournament.play(keep_interactions=True)
```

If the play method is called with `keep_interactions=True`, the result set object will have an `interactions` attribute which contains all the interactions between the players. These can be used to view the history of the interactions:

```

>>> for index_pair, interaction in sorted(results.interactions.items()):
...     player1 = tournament.players[index_pair[0]]
...     player2 = tournament.players[index_pair[1]]
...     print('%s vs %s: %s' % (player1, player2, interaction[0]))
Cooperator vs Cooperator: [(C, C), (C, C), (C, C)]
Cooperator vs Defector: [(C, D), (C, D), (C, D)]
Cooperator vs Tit For Tat: [(C, C), (C, C), (C, C)]
Cooperator vs Grudger: [(C, C), (C, C), (C, C)]
Defector vs Defector: [(D, D), (D, D), (D, D)]
Defector vs Tit For Tat: [(D, C), (D, D), (D, D)]
Defector vs Grudger: [(D, C), (D, D), (D, D)]
Tit For Tat vs Tit For Tat: [(C, C), (C, C), (C, C)]
Tit For Tat vs Grudger: [(C, C), (C, C), (C, C)]
Grudger vs Grudger: [(C, C), (C, C), (C, C)]

```

We can use these interactions to reconstruct `axelrod.Match` objects which have a variety of available methods for analysis (more information can be found in [Creating Matches](#)):

```

>>> matches = []
>>> for index_pair, interaction in sorted(results.interactions.items()):
...     player1 = tournament.players[index_pair[0]]
...     player2 = tournament.players[index_pair[1]]
...     match = axl.Match([player1, player2], turns=3)
...     match.result = interaction[0]
...     matches.append(match)
>>> len(matches)
10

```

As an example let us view all winners of each match (False indicates a tie):

```

>>> for match in matches:
...     print("{} v {}, winner: {}".format(match.players[0], match.players[1], match.
...     ↪winner()))
Cooperator v Cooperator, winner: False
Cooperator v Defector, winner: Defector
Cooperator v Tit For Tat, winner: False
Cooperator v Grudger, winner: False
Defector v Defector, winner: False
Defector v Tit For Tat, winner: Defector
Defector v Grudger, winner: Defector
Tit For Tat v Tit For Tat, winner: False
Tit For Tat v Grudger, winner: False
Grudger v Grudger, winner: False

```

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())
>>> 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 [Accessing tournament results](#), this becomes particularly useful when viewing the outputs of tournaments with a large number of strategies:

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

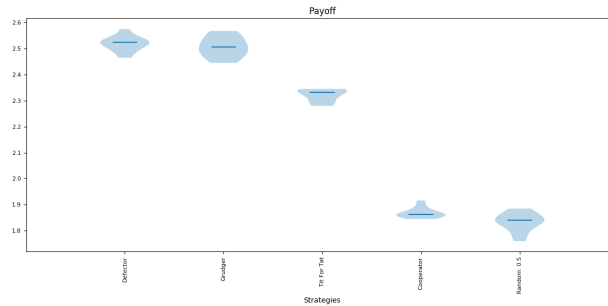
Saving all plots

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
>>> axl.seed(0) # for reproducible example
>>> players = [axl.Cooperator(), axl.Defector(),
...           axl.TitForTat(), axl.Grudger()]
>>> mp = axl.MoranProcess(players)
>>> populations = mp.play()
>>> mp.winning_strategy_name
'Defector'
```

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

```
>>> len(mp)
16
```

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

```
Counter({'Defector': 3, 'Cooperator': 1}),
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]
[6.0, 7.0, 7.0, 7.0]
[6.0, 11.0, 7.0, 6.0]
[3.0, 11.0, 11.0, 3.0]
[6.0, 15.0, 6.0, 6.0]
[6.0, 15.0, 6.0, 6.0]
[3.0, 11.0, 11.0, 3.0]
[7.0, 7.0, 7.0, 0.0]
[7.0, 7.0, 7.0, 0.0]
[7.0, 7.0, 7.0, 0.0]
[7.0, 7.0, 7.0, 0.0]
[7.0, 7.0, 7.0, 0.0]
[7.0, 7.0, 7.0, 0.0]
[7.0, 7.0, 7.0, 0.0]
[7.0, 7.0, 7.0, 0.0]
[7.0, 7.0, 7.0, 0.0]
```

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
>>> axl.seed(15) # for reproducible example
>>> 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)
>>> populations = mp.play()
>>> mp.winning_strategy_name
'Cooperator'
>>> ax = mp.populations_plot()
>>> plt.show()
```

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


Other types of implemented Moran processes:

- *Moran Process on Graphs*
- *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*

Research topics

This section contains descriptions of particular tools of interest to those doing game theoretic research.

Contents:

Noisy tournaments

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()
```

Probabilistic Ending Tournaments

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 *Accessing 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()
```

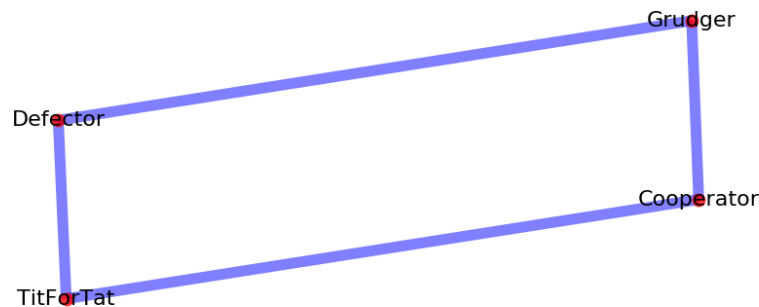
Spatial tournaments

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:

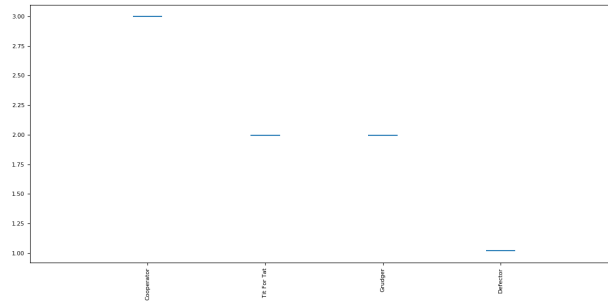
```
>>> import axelrod as axl
>>> players = [axl.Cooperator(), axl.Defector(),
...           axl.TitForTat(), axl.Grudger()]
>>> edges = [(0, 2), (0, 3), (1, 2), (1, 3)]
```

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

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

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 5 repetitions and obtain the interactions:

```
>>> spatial_tournament = axl.Tournament(players, turns=2, repetitions=2, edges=edges)
>>> results = spatial_tournament.play(keep_interactions=True)
>>> for index_pair, interaction in sorted(results.interactions.items()):
...     player1 = spatial_tournament.players[index_pair[0]]
...     player2 = spatial_tournament.players[index_pair[1]]
...     print('%s vs %s: %s' % (player1, player2, interaction))
Cooperator vs Tit For Tat: [[(C, C), (C, C)], [(C, C), (C, C)]]
Cooperator vs Grudger: [[(C, C), (C, C)], [(C, C), (C, C)]]
Defector vs Tit For Tat: [[(D, C), (D, D)], [(D, C), (D, D)]]
Defector vs Grudger: [[(D, C), (D, D)], [(D, C), (D, D)]]
```

As anticipated Cooperator does not interact with Defector neither TitForTat with Grudger.

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)
>>> prob_end_results = prob_end_spatial_tournament.play(keep_interactions=True)
```

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

```
>>> axl.seed(0)
>>> lengths = []
>>> for interaction in prob_end_results.interactions.values():
...     lengths.append(len(interaction[0]))
>>> min(lengths) != max(lengths)
True
```

Moran Process on Graphs

The library also provides a graph-based Moran process [\[Shakarjian2013\]](#) 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
>>> axl.seed(40)
>>> 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)
>>> results = mp.play()
>>> mp.population_distribution()
Counter({'Cooperator': 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`.

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:

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

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:

```
>>> axl.seed(3)
>>> amp = axl.MoranProcess(players)
>>> results = amp.play()
```

```
>>> amp.population_distribution()
Counter({'Defector': 3})
```

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]
```

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()
```

Fingerprinting

Ashlock Fingerprints

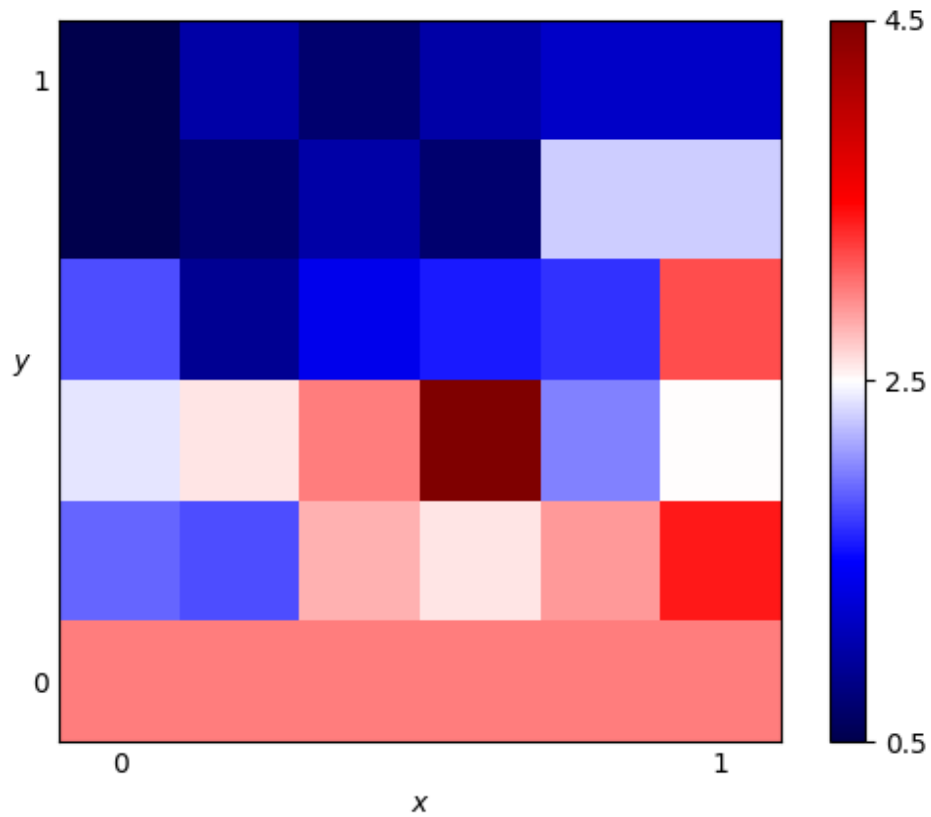
In [\[Ashlock2008\]](#), [\[Ashlock2009\]](#) a methodology for obtaining visual representation of a strategy's behaviour is described. The basic method is to play the strategy against a probe strategy with varying noise parameters. These noise parameters are implemented through the `JossAnnTransformer`. 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 $x + y \geq 1$ 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
>>> axl.seed(0) # Fingerprinting is a random process
>>> strategy = axl.WinStayLoseShift
>>> probe = axl.TitForTat
>>> af = axl.AshlockFingerprint(strategy, probe)
>>> data = af.fingerprint(turns=10, repetitions=2, step=0.2)
>>> 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()
```

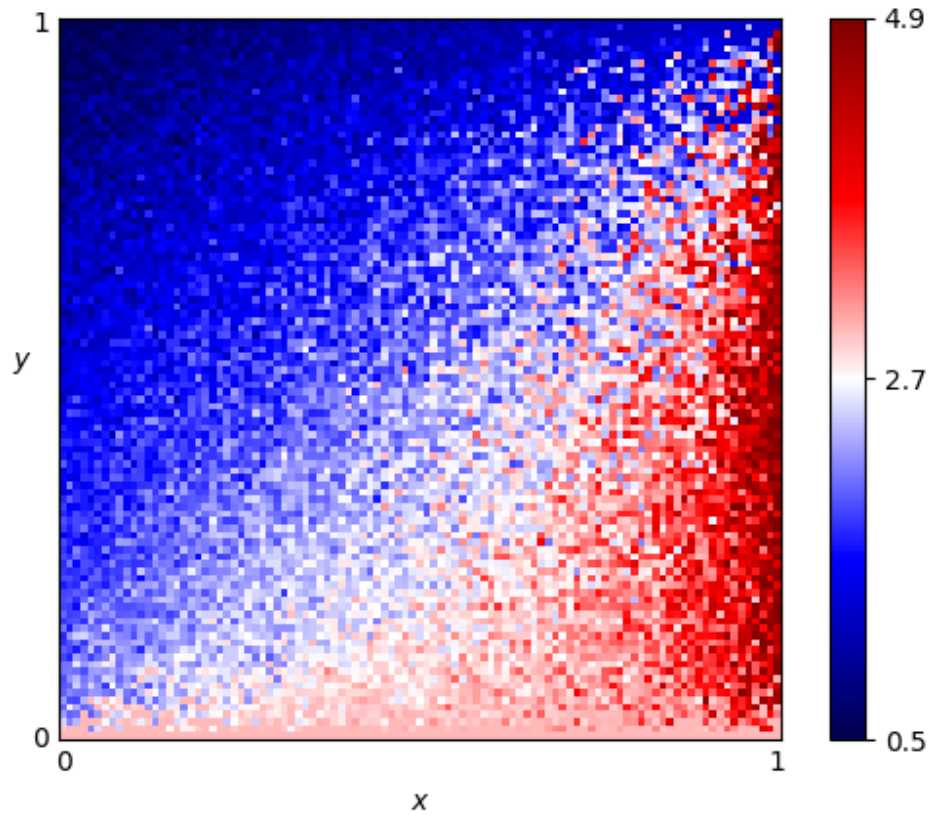


In reality we would need much more detail to make this plot useful.

Running the above with the following parameters:

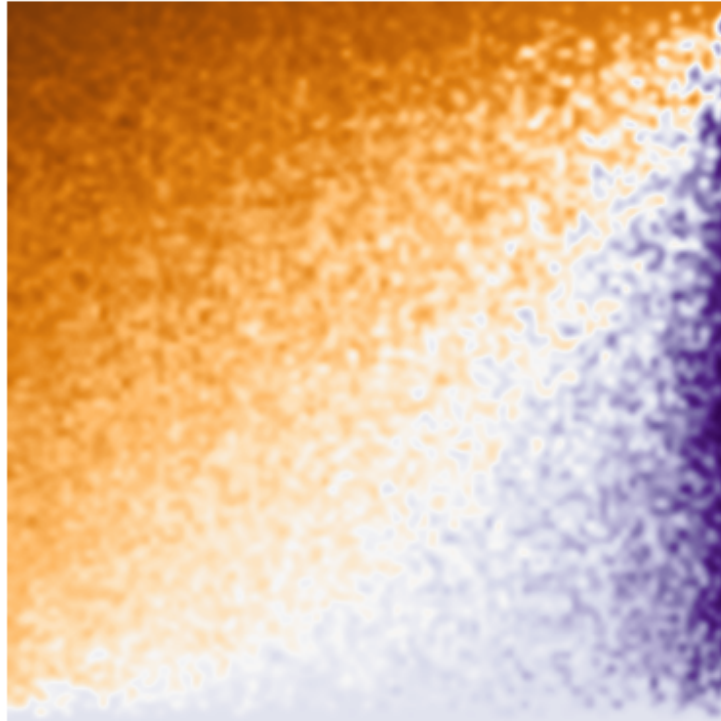
```
>>> af.fingerprint(turns=50, repetitions=2, step=0.01)
```

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:

```
>>> axl.seed(0)
>>> 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)
>>> data
{...
>>> data[(0, 0)]
4.4...
```

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:

```
>>> axl.seed(0)
>>> player = axl.TitForTat()
>>> tf = axl.TransitiveFingerprint(player)
```

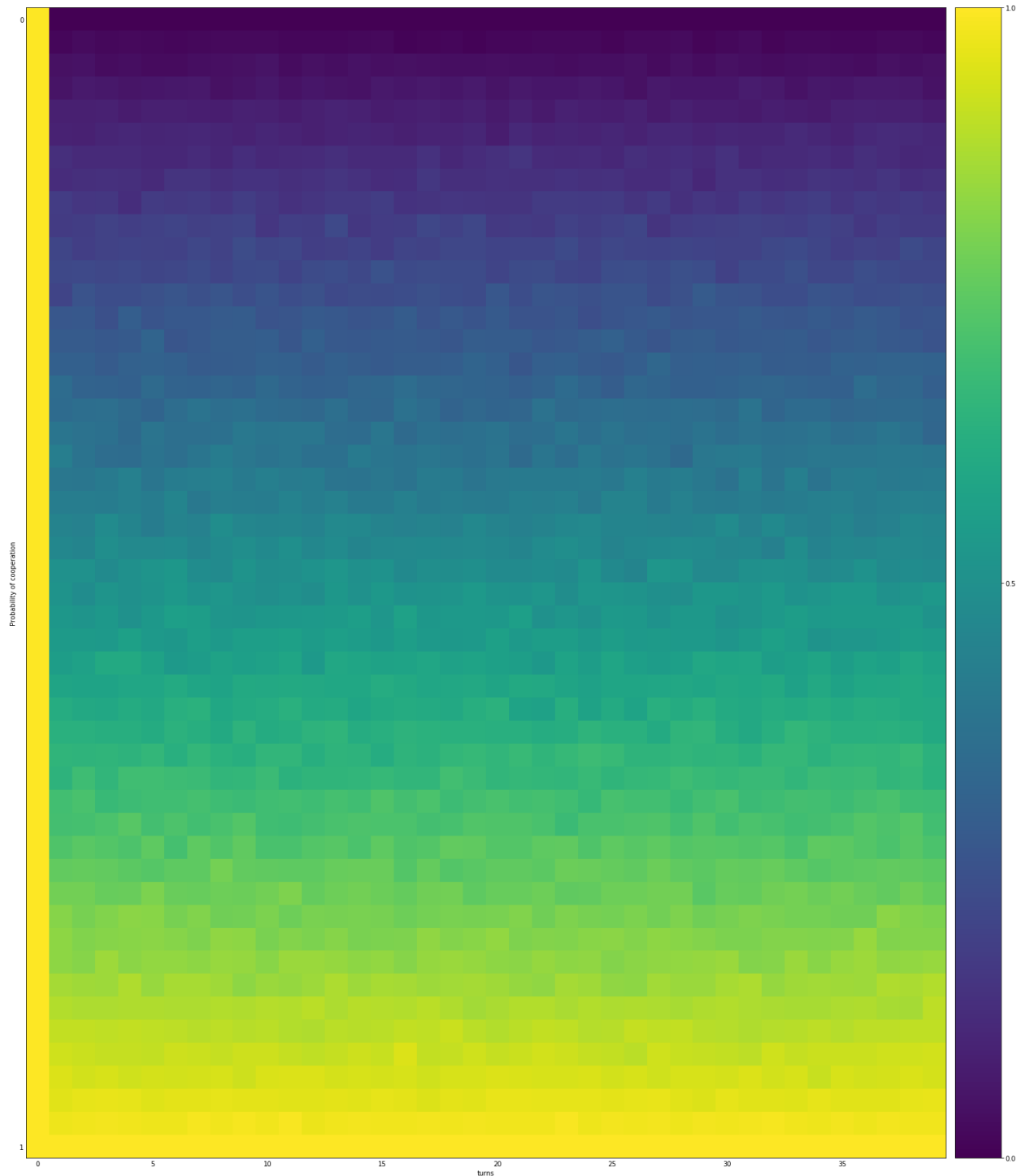
```
>>> data = tf.fingerprint(turns=40)
```

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()
```

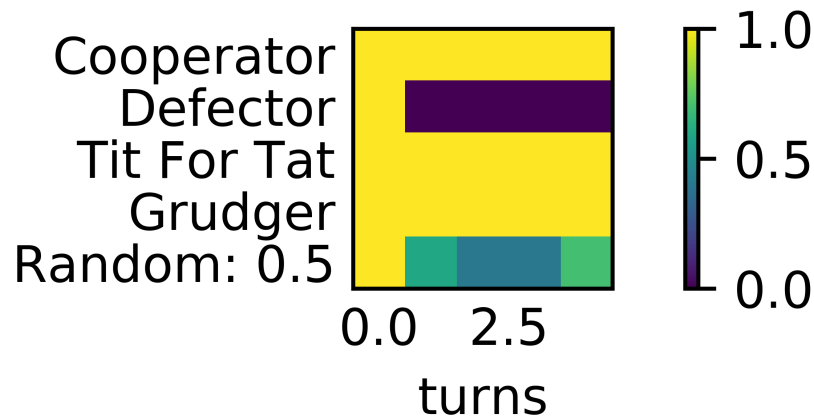


It is also possible to fingerprint against a given set of opponents:

```
>>> axl.seed(1)
>>> opponents = [s() for s in axl.demo_strategies]
>>> tf = axl.TransitiveFingerprint(player, opponents=opponents)
>>> data = tf.fingerprint(turns=5, repetitions=10)
```

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

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



Further capabilities in the library

This section shows some of the more intricate capabilities of the library.

Contents:

Accessing 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*.

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,
...     'makes_use_of': set([]),
...     'long_run_time': False,
...     '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
```

and that we can retrieve individual entries from that classifier dictionary:

```
>>> s = axl.TitForTat
>>> s.classifier['memory_depth']
1
>>> s = axl.Random
>>> s.classifier['stochastic']
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)
73
```

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)
31
```

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)
>>> len(strategies)
55
```

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)
30
```

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)
22
```

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 obey_axelrod function:

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

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()
>>> 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 consecutive 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]:

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

- `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:
        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.

Accessing 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 *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, 10, 10, 10], [10, 10, 10, 10], [10, 10, 10, 10], [10, 10, 10, 10]],
 [[10, 10, 10, 10], [10, 10, 10, 10], [10, 10, 10, 10], [10, 10, 10, 10]],
 [[10, 10, 10, 10], [10, 10, 10, 10], [10, 10, 10, 10], [10, 10, 10, 10]]]
```

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
[[0, 30, 30, 30], [0, 0, 0, 0], [30, 3, 0, 30], [30, 3, 30, 0]]
```

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({(C, C): 1.0}),
 Counter()]
```



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

```
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 *Morality Metrics*.

Reading and writing 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:

```

0,0,Alternator,Alternator,CDCD,CDCD
0,0,Alternator,Alternator,CDCD,CDCD
0,1,Alternator,Anti Tit For Tat,CDCD,CDCD
0,1,Alternator,Anti Tit For Tat,CDCD,CDCD
0,2,Alternator,Bully,CDCD,DDCD
0,2,Alternator,Bully,CDCD,DDCD
0,3,Alternator,Cooperator,CDCD,CCCC
0,3,Alternator,Cooperator,CDCD,CCCC
0,4,Alternator,Defector,CDCD,DDDD
0,4,Alternator,Defector,CDCD,DDDD
0,5,Alternator,Suspicious Tit For Tat,CDCD,DCDC
0,5,Alternator,Suspicious Tit For Tat,CDCD,DCDC
0,6,Alternator,Tit For Tat,CDCD,CCDC
0,6,Alternator,Tit For Tat,CDCD,CCDC
0,7,Alternator,Win-Stay Lose-Shift,CDCD,CCDD
0,7,Alternator,Win-Stay Lose-Shift,CDCD,CCDD
1,1,Anti Tit For Tat,Anti Tit For Tat,CDCD,CDCD
1,1,Anti Tit For Tat,Anti Tit For Tat,CDCD,CDCD
1,2,Anti Tit For Tat,Bully,CCCC,DDDD
1,2,Anti Tit For Tat,Bully,CCCC,DDDD
1,3,Anti Tit For Tat,Cooperator,CDDD,CCCC
1,3,Anti Tit For Tat,Cooperator,CDDD,CCCC
1,4,Anti Tit For Tat,Defector,CCCC,DDDD
1,4,Anti Tit For Tat,Defector,CCCC,DDDD
1,5,Anti Tit For Tat,Suspicious Tit For Tat,CCDD,DCCD
1,5,Anti Tit For Tat,Suspicious Tit For Tat,CCDD,DCCD
1,6,Anti Tit For Tat,Tit For Tat,CDDC,CCDD
1,6,Anti Tit For Tat,Tit For Tat,CDDC,CCDD
1,7,Anti Tit For Tat,Win-Stay Lose-Shift,CDDC,CCDC
1,7,Anti Tit For Tat,Win-Stay Lose-Shift,CDDC,CCDC
2,2,Bully,Bully,DCDC,DCDC
2,2,Bully,Bully,DCDC,DCDC
2,3,Bully,Cooperator,DDDD,CCCC
2,3,Bully,Cooperator,DDDD,CCCC
2,4,Bully,Defector,DCCC,DDDD
2,4,Bully,Defector,DCCC,DDDD
2,5,Bully,Suspicious Tit For Tat,DCCD,DDCC
2,5,Bully,Suspicious Tit For Tat,DCCD,DDCC
2,6,Bully,Tit For Tat,DDCC,CDDC
2,6,Bully,Tit For Tat,DDCC,CDDC
2,7,Bully,Win-Stay Lose-Shift,DDCD,CDCC
2,7,Bully,Win-Stay Lose-Shift,DDCD,CDCC
3,3,Cooperator,Cooperator,CCCC,CCCC
3,3,Cooperator,Cooperator,CCCC,CCCC
3,4,Cooperator,Defector,CCCC,DDDD
3,4,Cooperator,Defector,CCCC,DDDD
3,5,Cooperator,Suspicious Tit For Tat,CCCC,DCCC
3,5,Cooperator,Suspicious Tit For Tat,CCCC,DCCC
3,6,Cooperator,Tit For Tat,CCCC,CCCC
3,6,Cooperator,Tit For Tat,CCCC,CCCC
3,7,Cooperator,Win-Stay Lose-Shift,CCCC,CCCC
3,7,Cooperator,Win-Stay Lose-Shift,CCCC,CCCC
4,4,Defector,Defector,DDDD,DDDD
4,4,Defector,Defector,DDDD,DDDD
4,5,Defector,Suspicious Tit For Tat,DDDD,DDDD
4,5,Defector,Suspicious Tit For Tat,DDDD,DDDD
4,6,Defector,Tit For Tat,DDDD,CDDD

```

```
4,6,Defector,Tit For Tat,DDDD,CDDD
4,7,Defector,Win-Stay Lose-Shift,DDDD,CDCD
4,7,Defector,Win-Stay Lose-Shift,DDDD,CDCD
5,5,Suspicious Tit For Tat,Suspicious Tit For Tat,DDDD,DDDD
5,5,Suspicious Tit For Tat,Suspicious Tit For Tat,DDDD,DDDD
5,6,Suspicious Tit For Tat,Tit For Tat,DCDC,CDCD
5,6,Suspicious Tit For Tat,Tit For Tat,DCDC,CDCD
5,7,Suspicious Tit For Tat,Win-Stay Lose-Shift,DCDD,CDDC
5,7,Suspicious Tit For Tat,Win-Stay Lose-Shift,DCDD,CDDC
6,6,Tit For Tat,Tit For Tat,CCCC,CCCC
6,6,Tit For Tat,Tit For Tat,CCCC,CCCC
6,7,Tit For Tat,Win-Stay Lose-Shift,CCCC,CCCC
6,7,Tit For Tat,Win-Stay Lose-Shift,CCCC,CCCC
7,7,Win-Stay Lose-Shift,Win-Stay Lose-Shift,CCCC,CCCC
7,7,Win-Stay Lose-Shift,Win-Stay Lose-Shift,CCCC,CCCC
```

The columns of this file are of the form:

1. Index of first player
2. Index of second player
3. Name of first player
4. Name of second player
5. History of play of the first player
6. History of play of the second player

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

Alternator versus TitForTat has the following interactions: CCDC, CDCD:

- First turn: C versus C (the first two letters)
- Second turn: D versus C (the second pair of letters)
- Third turn: C versus D (the third pair of letters)
- Fourth turn: D versus C (the fourth pair of letters)

This can be transformed in to the usual interactions by zipping:

```
>>> from axelrod.action import str_to_actions
>>> list(zip(str_to_actions("CCDC"), str_to_actions("CDCD")))
[(C, C), (C, D), (D, C), (C, 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 $O(\text{players}^2 * \text{turns} * \text{repetitions})$.

It is also possible to generate a standard result set from a datafile:

```
>>> results = axl.ResultSetFromFile(filename="basic_tournament.csv")
>>> results.ranked_names
['Defector',
 'Bully',
 'Suspicious Tit For Tat',
 'Alternator',
 'Tit For Tat',
 'Anti Tit For Tat',
```

```
'Win-Stay Lose-Shift',
'Cooperator']
```

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)
```

Using the 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():
...     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', 200): [(C, C), ..., (C, D)]}
>>> len(cache)
1
```

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
```

Setting a random seed

The library has a variety of strategies whose behaviour is stochastic. To ensure reproducible results a random seed should be set. As both Numpy and the standard library are used for random number generation, both seeds need to be set. To do this we can use the *seed* function:

```
>>> import axelrod as axl
>>> players = (axl.Random(), axl.MetaMixer()) # Two stochastic strategies
>>> axl.seed(0)
>>> results = axl.Match(players, turns=3).play()
```

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

```
>>> axl.seed(0)
>>> results == axl.Match(players, turns=3).play()
True
```

Note that this is equivalent to:

```
>>> import numpy
>>> import random
>>> players = (axl.Random(), axl.MetaMixer())
>>> random.seed(0)
>>> numpy.random.seed(0)
>>> results = axl.Match(players, turns=3).play()
>>> numpy.random.seed(0)
>>> random.seed(0)
>>> results == axl.Match(players, turns=3).play()
True
```

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 `SteinAndRapoport` and `Alternator`. The latter of these two always defects on the last 2 turns:

```
>>> import axelrod as axl
>>> players = (axl.Alternator(), axl.SteinAndRapoport())
>>> 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.

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 *Fingerprinting*.

Using and playing 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']
```

Contributing

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. 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](#)
4. Testing: the project uses the [unittest](#) library and has a nice testing suite that makes some things very easy to write tests for. Please try to increase the test coverage on pull requests.
5. 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.

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 [Strategies 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.

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`).

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.

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
makes_use_of
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.Joss()
>>> boring_joss = axelrod.Joss(p=1)
>>> joss.classifier['stochastic'], boring_joss.classifier['stochastic']
(True, False)
```

Dimensions that are not classified have value `None` in the dictionary.

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 player 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 use 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 test 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

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
```

Continuous integration

This project is being taken care of by [travis-ci](#), so all tests will be run automatically when opening a pull request. You can see the latest build status [here](#).

Reference

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

Contents:

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).

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)
```

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 2.1: Strategies in Axelrod’s first tournament

Name	Author	Axelrod Library Name
Tit For Tat	Anatol Rapoport	<i>TitForTat</i>
<i>Tideman and Chieruzzi</i>	T Nicolaus Tideman and Paula Chieruzz	<i>TidemanAndChieruzzi</i>
Nydegger	Rudy Nydegger	<i>Nydegger</i>
Grofman	Bernard Grofman	<i>Grofman</i>
Shubik	Martin Shubik	<i>Shubik</i>
Stein and Rapoport	Stein and Anatol Rapoport	<i>SteinAndRapoport</i>
Grudger	James W Friedman	<i>Grudger</i>
Davis	Morton Davis	<i>Davis</i>
<i>Graaskamp</i>	Jim Graaskamp	Not Implemented
Downing	Leslie Downing	<i>RevisedDowning</i>
Feld	Scott Feld	<i>Feld</i>
Joss	Johann Joss	<i>Joss</i>
Tullock	Gordon Tullock	<i>Tullock</i>
Unnamed Strategy	Unknown	<i>UnnamedStrategy</i>
Random	Unknownd	<i>Random</i>

Tideman and Chieruzzi

Not implemented yet

This strategy begins by playing Tit For Tat and then things get slightly complicated:

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 tournament
 - **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).

This strategy came 2nd in Axelrod’s original tournament.

Graaskamp

Not implemented yet

This strategy follows the following rules:

1. Play 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;

5. The strategy also checks to see if the opponent is playing Tit For Tat or another strategy from a preliminary tournament called ‘Analogy’. If so it plays Tit For Tat. If not it cooperates and randomly defects every 5 to 15 moves.

This strategy came 9th in Axelrod’s original tournament.

Axelrod’s second tournament

The code for Axelrod’s second tournament 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 2.2: Strategies in Axelrod’s second tournament

Original Code	Author	Axelrod Library Name
GRASR	Unknown	Not Implemented
K31R	Gail Grisell	Not Implemented
K32R	Charles Kluepfel	Not Implemented
K33R	Harold Rabbie	Not Implemented
K34R	James W Friedman	<i>Grudger</i>
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	Not Implemented
K42R	Otto Borufsen	Not Implemented
K43R	R D Anderson	Not Implemented
K44R	William Adams	Not Implemented
K45R	Michael F McGurrin	Not Implemented
K46R	Graham J Eatherley	Not Implemented
K47R	Richard Hufford	Not Implemented
K48R	George Hufford	Not Implemented
K49R	Rob Cave	Not Implemented
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	Not Implemented
K59R	Leslie Downing	Not Implemented
K60R	Jim Graaskamp and Ken Katzen	Not Implemented
K61R	Danny C Champion	<i>Champion</i>

Continued on next page

Table 2.2 – continued from previous page

Original Code	Author	Axelrod Library Name
K62R	Howard R Hollander	Not Implemented
K63R	George Duisman	Not Implemented
K64R	Brian Yamachi	Not Implemented
K65R	Mark F Batell	Not Implemented
K66R	Ray Mikkelson	Not Implemented
K67R	Craig Feathers	<i>Tranquilizer</i>
K68R	Fransois Leyvraz	Not Implemented
K69R	Johann Joss	Not Implemented
K70R	Robert Pebly	Not Implemented
K71R	James E Hall	Not Implemented
K72R	Edward C White Jr	Not Implemented
K73R	George Zimmerman	Not Implemented
K74R	Edward Friedland	Not Implemented
K74RXX	Edward Friedland	Not Implemented
K75R	Paul D Harrington	Not Implemented
K76R	David Gladstein	<i>Gladstein</i>
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	Not Implemented
K84R	T Nicolaus Tideman and Paula Chieruzz	Not Implemented
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	Not Implemented
K89R	Gene Snodgrass	Not Implemented
K90R	John Maynard Smith	Not Implemented
K91R	Jonathan Pinkley	Not Implemented
K92R	Anatol Rapoport	<i>TitForTat</i>
K93R	Unknown	Not Implemented
KPAVLOVC	Unknown	<i>WinStayLoseShift</i>
KRANDOMC	Unknown	<i>Random</i>
KTF2TC	Unknown	<i>TitFor2Tats</i>
KTITFORTATC	Unknown	<i>TitForTat</i>

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 2.3: Strategies in Stewart and Plotkin's tournament

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

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

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:

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

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:

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

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:

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

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:

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

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:

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

- $P(C | DD) = 0$

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:

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

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](<http://www.prisoners-dilemma.com/strategies.html>) 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.

Strategies index

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

```
class axelrod.strategies.adaptive.Adaptive (initial_plays: typing.List[axelrod.action.Action]
                                           = None) → None
    Start with a specific sequence of C and D, then play the strategy that has worked best, recalculated each turn.
```

Names:

- Adaptive: [\[Li2011\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': {'game'}, 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'Adaptive'
score_last_round (opponent: axelrod.player.Player)
strategy (opponent: axelrod.player.Player) → axelrod.action.Action
```

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

A player who alternates between cooperating and defecting.

Names

- Alternator: [\[Axelrod1984\]](#)
- Periodic player CD: [\[Mittal2009\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'Alternator'
strategy (opponent: axelrod.player.Player) → axelrod.action.Action
```

class axelrod.strategies.ann.**ANN** (weights: typing.List[float], num_features: int, num_hidden: int) → 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

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'ANN'
strategy (opponent: axelrod.player.Player) → axelrod.action.Action
```

class axelrod.strategies.ann.**EvolvedANN** → None

A strategy based on a pre-trained neural network with 17 features and a hidden layer of size 10.

Names:

- Evolved ANN: Original name by Martin Jones.

```
name = 'Evolved ANN'
```

class axelrod.strategies.ann.**EvolvedANN5** → None

A strategy based on a pre-trained neural network with 17 features and a hidden layer of size 5.

Names:

- Evolved ANN 5: Original name by Marc Harper.

```
name = 'Evolved ANN 5'
```

class `axelrod.strategies.ann.EvolvedANNNoise05` → None

A strategy based on a pre-trained neural network with a hidden layer of size 10, trained with noise=0.05.

Names:

- Evolved ANN Noise 05: Original name by Marc Harper.

name = 'Evolved ANN 5 Noise 05'

`axelrod.strategies.ann.activate` (*bias: typing.List[float], hidden: typing.List[float], output: typing.List[float], inputs: typing.List[int]*) → 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*) → typing.List[int]

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: typing.List[float], num_features: int, num_hidden: int*) → typing.Tuple[typing.List[typing.List[float]], typing.List[float], typing.List[float]]

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

class `axelrod.strategies.apavlov.APavlov2006` → None

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: [\[Li2007\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'Adaptive Pavlov 2006'

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

class `axelrod.strategies.apavlov.APavlov2011` → None

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: [\[Li2011\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'Adaptive Pavlov 2011'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Appeaser'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Average Copier'

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

class axelrod.strategies.averagecopier.**NiceAverageCopier**

Same as Average Copier, but always starts by cooperating.

Names:

- Average Copier: Original name by Owen Campbell

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Nice Average Copier'

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

Additional strategies from Axelrod's first tournament.

class axelrod.strategies.axelrod_first.**Davis** (*rounds_to_cooperate: int = 10*) → None

Submitted to Axelrod's first tournament by Morton Davis.

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: [\[Axelrod1980\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Davis'

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.**Feld** (*start_coop_prob: float = 1.0, end_coop_prob: float = 0.5, rounds_of_decay: int = 200*) → None

Submitted to Axelrod's first tournament by Scott Feld.

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.

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

Names:

•Feld: [\[Axelrod1980\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run'
name = 'Feld'
```

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

class axelrod.strategies.axelrod_first.**Grofman**

Submitted to Axelrod's first tournament by Bernard Grofman.

Cooperate on the first two rounds and returns the opponent's last action for the next 5. For the rest of the game Grofman cooperates if both players selected the same action in the previous round, and otherwise cooperates randomly with probability 2/7.

This strategy came 4th in Axelrod's original tournament.

Names:

•Grofman: [\[Axelrod1980\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run'
name = 'Grofman'
```

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

class axelrod.strategies.axelrod_first.**Joss** (*p: float = 0.9*) → None

Submitted to Axelrod's first tournament by Johann Joss.

Cooperates with probability 0.9 when the opponent cooperates, otherwise emulates Tit-For-Tat.

This strategy came 12th in Axelrod's original tournament.

Names:

•Joss: [\[Axelrod1980\]](#)

•Hard Joss: [\[Stewart2012\]](#)

```
name = 'Joss'
```

class axelrod.strategies.axelrod_first.**Nydegger** → None

Submitted to Axelrod's first tournament by Rudy Nydegger.

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: [\[Axelrod1980\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Nydegger'
```

```
static score_history (my_history: typing.List[axelrod.action.Action], opponent_history: typing.List[axelrod.action.Action], score_map: typing.Dict[typing.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
```

class axelrod.strategies.axelrod_first.**RevisedDowning** (*revised: bool = True*) → None

This strategy attempts to estimate the next move of the opponent by estimating the probability of cooperating given that they defected ($p(C|D)$) or cooperated on the previous round ($p(C|C)$). These probabilities are continuously updated during play and the strategy attempts to maximise the long term play. Note that the initial values are $p(C|C) = p(C|D) = .5$.

Downing is implemented as *RevisedDowning*. Apparently in the first tournament the strategy was implemented incorrectly and defected on the first two rounds. This can be controlled by setting *revised=True* to prevent the initial defections.

This strategy came 10th in Axelrod's original tournament but would have won if it had been implemented correctly.

Names:

•Revised Downing: [\[Axelrod1980\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Revised Downing'
```

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

class axelrod.strategies.axelrod_first.**Shubik** → None

Submitted to Axelrod's first tournament by Martin Shubik.

Plays like Tit-For-Tat with the following modification. After each retaliation, the number of rounds that Shubik retaliates increases by 1.

This strategy came 5th in Axelrod's original tournament.

Names:

•Shubik: [\[Axelrod1980\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Shubik'
```

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

class axelrod.strategies.axelrod_first.**SteinAndRapoport** (*alpha: float = 0.05*) → None

This strategy plays a modification of Tit For Tat.

- 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.

This strategy came 6th in Axelrod's original tournament.

Names:

•SteinAndRapoport: *[Axelrod1980]*

```

classifier = {'manipulates_state': False, 'makes_use_of': {'length'}, 'stochastic': False, 'inspects_source': False, 'long_run': False}
decorator = <axelrod.strategy_transformers.StrategyTransformerFactory.<locals>.Decorator object>
name = 'Stein and Rapoport'
original_class
    alias of SteinAndRapoport
strategy (opponent)
    
```

class axelrod.strategies.axelrod_first.**TidemanAndChieruzzi** → None

This strategy begins by playing Tit For Tat and then follows the following rules:

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).

This strategy came 2nd in Axelrod's original tournament.

Names:

•TidemanAndChieruzzi: *[Axelrod1980]*

```

classifier = {'manipulates_state': False, 'makes_use_of': {'length', 'game'}, 'stochastic': False, 'inspects_source': False, 'long_run': False}
name = 'Tideman and Chieruzzi'
strategy (opponent: axelrod.player.Player) → axelrod.action.Action
    
```

class axelrod.strategies.axelrod_first.**Tullock** (*rounds_to_cooperate: int = 11*) → None

Submitted to Axelrod's first tournament by Gordon Tullock.

Cooperates for the first 11 rounds then randomly cooperates 10% less often than the opponent has in previous rounds.

This strategy came 13th in Axelrod's original tournament.

Names:

•Tullock: *[Axelrod1980]*

```

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run': False}
name = 'Tullock'
strategy (opponent: axelrod.player.Player) → axelrod.action.Action
    
```

class axelrod.strategies.axelrod_first.**UnnamedStrategy**

Apparently written by a grad student in political science whose name was withheld, this strategy cooperates with a given probability P. This probability (which has initial value .3) is updated every 10 rounds based on whether

the opponent seems to be random, very cooperative or very uncooperative. Furthermore, if after round 130 the strategy is losing then P is also adjusted.

Fourteenth Place with 282.2 points is a 77-line program by a graduate student of political science whose dissertation is in game theory. 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.

Names:

- Unnamed Strategy: [\[Axelrod1980\]](#)

Warning: This strategy is not identical to the original strategy (source unavailable) and was written based on published descriptions.

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run'
name = 'Unnamed Strategy'
static strategy (opponent: axelrod.player.Player) → axelrod.action.Action
```

Additional strategies from Axelrod's second tournament.

class axelrod.strategies.axelrod_second.**Champion**
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 than the other player's cooperation rate.

Names:

- Champion: [\[Axelrod1980b\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': {'length'}, 'stochastic': True, 'inspects_source': False, 'long_run'
name = 'Champion'
strategy (opponent: axelrod.player.Player) → axelrod.action.Action
```

class axelrod.strategies.axelrod_second.**Eatherley**
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: [\[Axelrod1980b\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run'
name = 'Eatherley'
static strategy (opponent: axelrod.player.Player) → axelrod.action.Action
```

class axelrod.strategies.axelrod_second.**Gladstein** → None
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: [\[Axelrod1980b\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Gladstein'
```

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

class axelrod.strategies.axelrod_second.**MoreGrofman**

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: [\[Axelrod1980b\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'MoreGrofman'
```

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

class axelrod.strategies.axelrod_second.**Tester** → None

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: [\[Axelrod1980b\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Tester'
```

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

class axelrod.strategies.axelrod_second.**Tranquilizer**

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`:

$$\text{self.two_turns_after_good_defection_ratio} = \left(\left(\text{self.two_turns_after_good_defection_ratio} * \text{self.two_turns_after_good_defection_ratio_count} \right) + \left(3 - \left(3 * \text{self.dict[opponent.history[-1]]} \right) \right) + \left(2 * \text{self.dict[self.history[-1]]} \right) - \left(\text{self.dict[opponent.history[-1]]} * \text{self.dict[self.history[-1]]} \right) \right) / \left(\text{self.two_turns_after_good_defection_ratio_count} + 1 \right)$$

$$\text{two_turns_after_good_defection_ratio_count} = \text{two_turns_after_good_defection_ratio} + 1$$

- 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`:

$$\text{self.one_turn_after_good_defection_ratio} = \left(\left(\text{self.one_turn_after_good_defection_ratio} * \text{self.one_turn_after_good_defection_ratio_count} \right) + \left(3 - \left(3 * \text{self.dict[opponent.history[-1]]} \right) \right) + \left(2 * \text{self.dict[self.history[-1]]} \right) - \left(\text{self.dict[opponent.history[-1]]} * \text{self.dict[self.history[-1]]} \right) \right) / \left(\text{self.one_turn_after_good_defection_ratio_count} + 1 \right)$$

`one_turn_after_good_defection_ratio_count`:

$$\text{one_turn_after_good_defection_ratio_count} = \text{one_turn_after_good_defection_ratio} + 1$$

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:

$$\text{probability} = \left(\left(.95 - \left(\left(\left(\text{self.one_turn_after_good_defection_ratio} \right) + \left(\text{self.two_turns_after_good_defection_ratio} \right) \right) - 5 \right) / 15 \right) + \left(1 / \left(\left(\text{len}(\text{self.history}) + 1 \right) ** 2 \right) \right) - \left(\text{self.dict[opponent.history[-1]]} / 4 \right) \right)$$

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 tournament.

Names:

- Tranquilizer: *[Axelrod1980]*

```
classifier = {'manipulates_state': False, 'makes_use_of': {'game'}, 'stochastic': True, 'inspects_source': False, 'long'
```

```
name = 'Tranquilizer'
```

```
strategy (opponent: axelrod.player.Player) → axelrod.action.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.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

```
classifier = {'manipulates_state': False, 'makes_use_of': {'length'}, 'stochastic': False, 'inspects_source': False, 'long'
```

```
decorator = <axelrod.strategy_transformers.StrategyTransformerFactory.<locals>.Decorator object>
```

```
name = 'BackStabber'
```

```
original_class
```

alias of *BackStabber*

```
strategy (opponent)
```

```
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

```
classifier = {'manipulates_state': False, 'makes_use_of': {'length'}, 'stochastic': False, 'inspects_source': False, 'long'
```

```
decorator = <axelrod.strategy_transformers.StrategyTransformerFactory.<locals>.Decorator object>
```

```
name = 'DoubleCROSSER'
```

```
original_class
```

alias of *DoubleCROSSER*

```
strategy (opponent)
```

```
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: [\[Prison1998\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Better and Better'

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

class axelrod.strategies.calculator.**Calculator** → None

Plays like (Hard) Joss for the first 20 rounds. If periodic behavior is detected, defect forever. Otherwise play TFT.

Names:

- Calculator: [\[Prison1998\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

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

name = 'Calculator'

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

class axelrod.strategies.cooperator.**Cooperator**

A player who only ever cooperates.

Names:

- Cooperator: [\[Axelrod1984\]](#)
- ALLC: [\[Press2012\]](#)
- Always cooperate: [\[Mittal2009\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Cooperator'

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

class axelrod.strategies.cooperator.**TrickyCooperator**

A cooperator that is trying to be tricky.

Names:

- Tricky Cooperator: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Tricky Cooperator'

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** → None

A player that follows a sequence of plays that contains no cycles: CDD CD CCD CCCD CCCCC ...

Names:

- Anti Cycler: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'AntiCycler'

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

class *axelrod.strategies.cycler.Cycler* (*cycle*: *str* = 'CCD') → None
 A player that repeats a given sequence indefinitely.

Names:

- Cycler: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': False}

get_new_itertools_cycle ()

name = 'Cycler'

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

class *axelrod.strategies.cycler.CyclerCCCCCD* → None
 Cycles C, C, C, C, C, D

Names:

- Cycler CCCCD: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': False}

name = 'Cycler CCCCD'

class *axelrod.strategies.cycler.CyclerCCCD* → None
 Cycles C, C, C, D

Names:

- Cycler CCCD: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': False}

name = 'Cycler CCCD'

class *axelrod.strategies.cycler.CyclerCCCDCD* → None
 Cycles C, C, C, D, C, D

Names:

- Cycler CCCDCD: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': False}

name = 'Cycler CCCDCD'

class *axelrod.strategies.cycler.CyclerCCD* → None
 Cycles C, C, D

Names:

- Cycler CCD: Original name by Marc Harper

- Periodic player CCD: *[Mittal2009]*

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': False}

name = 'Cycler CCD'

class *axelrod.strategies.cycler.CyclerDC* → None
 Cycles D, C

Names:

- Cycler DC: Original name by Marc Harper

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Cycler DC'
```

class `axelrod.strategies.cycler.CyclerDDC` → None
Cycles D, D, C

Names:

- Cycler DDC: Original name by Marc Harper
- Periodic player DDC: [Mittal2009]

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Cycler DDC'
```

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` → None

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

```
classifier = {'manipulates_state': True, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': True, 'long_run'
```

```
static foil_strategy_inspection () → axelrod.action.Action
    Foils _strategy_utils.inspect_strategy and _strategy_utils.look_ahead
```

```
genome = [C]
```

```
mutate (outcome: tuple, trial: int) → None
    Select response according to outcome.
```

```
name = 'Darwin'
```

```
receive_match_attributes ()
```

```
reset ()
    Reset instance properties.
```

```
static reset_genome () → None
    For use in testing methods.
```

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

```
valid_callers = ['play']
```

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

- Desired Belief Strategy: *[Au2006]*

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

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. `alpha`: int, optional. Discount factor. Default is 1.

name = 'DBS'

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

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.

get_value ()

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.

get_siblings ()

is_stochastic ()

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.action_to_int` (*action*)

`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: [\[Mittal2009\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Defector'

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

class `axelrod.strategies.defector.TrickyDefector`

A defector that is trying to be tricky.

Names:

- Tricky Defector: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Tricky Defector'

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*]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'memory_depth': inf, 'inspects_source': False, 'long_

name = 'Doubler'

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

class `axelrod.strategies.finite_state_machines.EvolvedFSM16` → None

A 16 state FSM player trained with an evolutionary algorithm.

Names:

- Evolved FSM 16: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Evolved FSM 16'

class `axelrod.strategies.finite_state_machines.EvolvedFSM16Noise05` → None

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Evolved FSM 16 Noise 05'

class `axelrod.strategies.finite_state_machines.EvolvedFSM4` → None

A 4 state FSM player trained with an evolutionary algorithm.

Names:

- Evolved FSM 4: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Evolved FSM 4'

class `axelrod.strategies.finite_state_machines.FSMPlayer` (*transitions: tuple = ((1, C, 1, C), (1, D, 1, D))*,
initial_state: int = 1,
initial_action: axelrod.action.Action = C)
 → None

Abstract base class for finite state machine players.

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'FSM Player'

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

class `axelrod.strategies.finite_state_machines.Fortress3` → None

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: [\[Ashlock2006b\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Fortress3'
```

class `axelrod.strategies.finite_state_machines.Fortress4` → None
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: [\[Ashlock2006b\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Fortress4'
```

class `axelrod.strategies.finite_state_machines.Predator` → None
Finite state machine player specified in <http://DOI.org/10.1109/CEC.2006.1688322>.

Names:

•Predator: [\[Ashlock2006b\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Predator'
```

class `axelrod.strategies.finite_state_machines.Pun1` → None
FSM player described in [\[Ashlock2006\]](#).

Names:

•Pun1: [\[Ashlock2006\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Pun1'
```

class `axelrod.strategies.finite_state_machines.Raider` → None
FSM player described in <http://DOI.org/10.1109/FOCI.2014.7007818>.

Names

•Raider: [\[Ashlock2014\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Raider'
```

class `axelrod.strategies.finite_state_machines.Ripoff` → None
FSM player described in <http://DOI.org/10.1109/TEVC.2008.920675>.

Names

•Ripoff: [\[Ashlock2008\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Ripoff'
```

class `axelrod.strategies.finite_state_machines.SimpleFSM` (*transitions: tuple, initial_state: int*) → None
Simple implementation of a finite state machine that transitions between states based on the last round of play.
http://en.wikipedia.org/wiki/Finite-state_machine

move (*opponent_action*: *axelrod.action.Action*) → *axelrod.action.Action*
Computes the response move and changes state.

state

state_transitions

class *axelrod.strategies.finite_state_machines.SolutionB1* → None
FSM player described in <http://DOI.org/10.1109/TCIAIG.2014.2326012>.

Names

•Solution B1: *[Ashlock2015]*

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'SolutionB1'

class *axelrod.strategies.finite_state_machines.SolutionB5* → None
FSM player described in <http://DOI.org/10.1109/TCIAIG.2014.2326012>.

Names

•Solution B5: *[Ashlock2015]*

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'SolutionB5'

class *axelrod.strategies.finite_state_machines.TF1* → None
A FSM player trained to maximize Moran fixation probabilities.

Names:

•TF1: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'TF1'

class *axelrod.strategies.finite_state_machines.TF2* → None
A FSM player trained to maximize Moran fixation probabilities.

Names:

•TF2: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'TF2'

class *axelrod.strategies.finite_state_machines.TF3* → None
A FSM player trained to maximize Moran fixation probabilities.

Names:

•TF3: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'TF3'

class *axelrod.strategies.finite_state_machines.Thumper* → None
FSM player described in <http://DOI.org/10.1109/TEVC.2008.920675>.

Names

•Thumper: *[Ashlock2008]*

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Thumper'
```

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

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Forgiver'
```

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

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Forgiving Tit For Tat'
```

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.Gambler` (*lookup_dict: dict = None, initial_actions: tuple = None, pattern: typing.Any = None, parameters: axelrod.strategies.lookerup.Plays = None*) → `None`

A stochastic version of LookerUp which will select randomly an action in some cases.

Names:

- Gambler: Original name by Georgios Koutsovoulos

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run'
name = 'Gambler'
```

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

class `axelrod.strategies.gambler.PSOGambler1_1_1` → `None`

A 1x1x1 PSOGambler trained with pyswarm.

Names:

- PSO Gambler 1_1_1: Original name by Marc Harper

```
name = 'PSO Gambler 1_1_1'
```

class `axelrod.strategies.gambler.PSOGambler2_2_2` → `None`

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

name = 'PSO Gambler 2_2_2'

class axelrod.strategies.gambler.**PSOGambler2_2_2_Noise05** → None
 A 2x2x2 PSOGambler trained with pyswarm with noise=0.05.

Names:

- PSO Gambler 2_2_2 Noise 05: Original name by Marc Harper

name = 'PSO Gambler 2_2_2 Noise 05'

class axelrod.strategies.gambler.**PSOGamblerMem1** → None
 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

name = 'PSO Gambler Mem1'

class axelrod.strategies.gambler.**ZDMem2** → None
 A memory two generalization of a zero determinant player.

Names:

- ZDMem2: Original name by Marc Harper

- Unnamed [*LiS2014*]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'ZD-Mem2'

The player classes in this module do not obey standard rules of the IPD (as indicated by their classifier). We do not recommend putting a lot of time in to optimising them.

class axelrod.strategies.geller.**Geller**
 Observes what the player will do in the next round and adjust.

If unable to do this: will play randomly.

This code is inspired by Matthew Williams' talk "Cheating at rock-paper-scissors — meta-programming in Python" given at Django Weekend Cardiff in February 2014.

His code is here: https://github.com/mattjw/rps_metaprogramming and there's some more info here: <http://www.mattjw.net/2014/02/rps-metaprogramming/>

This code is **way** simpler than Matt's, as in this exercise we already have access to the opponent instance, so don't need to go hunting for it in the stack. Instead we can just call it to see what it's going to play, and return a result based on that

This is almost certainly cheating, and more than likely against the spirit of the 'competition' :-)

Names:

- Geller: Original name by Martin Chorley (@martinjc)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': True, 'long_run

static foil_strategy_inspection () → axelrod.action.Action

Foils_strategy_utils.inspect_strategy and _strategy_utils.look_ahead

name = 'Geller'

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

Look at what the opponent will play in the next round and choose a strategy that gives the least jail time, which is equivalent to playing the same strategy as that which the opponent will play.

class axelrod.strategies.geller.**GellerCooperator**

Observes what the player will do (like Geller) but if unable to will cooperate.

Names:

- Geller Cooperator: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': True, 'long_run

static foil_strategy_inspection () → axelrod.action.Action

Foils_strategy_utils.inspect_strategy and _strategy_utils.look_ahead

name = 'Geller Cooperator'

class axelrod.strategies.geller.**GellerDefector**

Observes what the player will do (like Geller) but if unable to will defect.

Names:

- Geller Defector: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': True, 'long_run

static foil_strategy_inspection () → axelrod.action.Action

Foils_strategy_utils.inspect_strategy and _strategy_utils.look_ahead

name = 'Geller Defector'

class axelrod.strategies.gobymajority.**GoByMajority** (*memory_depth: typing.Union[int, float] = inf, soft: bool = True*) → None

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*]

- Soft Majority: [*Mittal2009*]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Go By Majority'

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** → None

GoByMajority player with a memory of 10.

Names:

- Go By Majority 10: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Go By Majority 10'

class `axelrod.strategies.gobymajority.GoByMajority20` → None
GoByMajority player with a memory of 20.

Names:

- Go By Majority 20: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Go By Majority 20'

class `axelrod.strategies.gobymajority.GoByMajority40` → None
GoByMajority player with a memory of 40.

Names:

- Go By Majority 40: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Go By Majority 40'

class `axelrod.strategies.gobymajority.GoByMajority5` → None
GoByMajority player with a memory of 5.

Names:

- Go By Majority 5: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Go By Majority 5'

class `axelrod.strategies.gobymajority.HardGoByMajority` (*memory_depth: typing.Union[int, float] = inf*) → None

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]*

name = 'Hard Go By Majority'

class `axelrod.strategies.gobymajority.HardGoByMajority10` → None
HardGoByMajority player with a memory of 10.

Names:

- Hard Go By Majority 10: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Hard Go By Majority 10'

class `axelrod.strategies.gobymajority.HardGoByMajority20` → None
HardGoByMajority player with a memory of 20.

Names:

- Hard Go By Majority 20: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Hard Go By Majority 20'

class `axelrod.strategies.gobymajority.HardGoByMajority40` → None
HardGoByMajority player with a memory of 40.

Names:

- Hard Go By Majority 40: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'

name = 'Hard Go By Majority 40'

class `axelrod.strategies.gobymajority.HardGoByMajority5` → None
HardGoByMajority player with a memory of 5.

Names:

- Hard Go By Majority 5: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'

name = 'Hard Go By Majority 5'

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]*

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'

decorator = <axelrod.strategy_transformers.StrategyTransformerFactory.<locals>.Decorator object>

name = 'Gradual Killer'

original_class

alias of *GradualKiller*

strategy (*opponent*)

class `axelrod.strategies.grudger.Aggravater`

Grudger, except that it defects on the first 3 turns

Names

- Aggravater: Original name by Thomas Campbell

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'

name = 'Aggravater'

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

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]*

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'

name = 'EasyGo'

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** → None
 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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'Forgetful Grudger'

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*) → None

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'General Soft Grudger'

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]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'Grudger'

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

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'GrudgerAlternator'
```

```
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

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Opposite Grudger'
```

```
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 → None
```

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]

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Soft Grudger'
```

```
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.grumpy.Grumpy (starting_state: str = 'Nice', grumpy_threshold: int = 10, nice_threshold: int = -10) → None
```

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

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Grumpy'
```

```
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: typing.List[axelrod.action.Action] = None) → None
```

Starts with C, D. If the opponent plays the same way, cooperate forever, else defect forever.

Names:

- Handshake: [Robson1990]

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Handshake'
```

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

class axelrod.strategies.hmm.**EvolvedHMM5** → None

An HMM-based player with five hidden states trained with an evolutionary algorithm.

Names:

- Evolved HMM 5: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Evolved HMM 5'

class axelrod.strategies.hmm.**HMMPlayer** (*transitions_C=None, transitions_D=None, emission_probabilities=None, initial_state=0, initial_action=C*) → None

Abstract base class for Hidden Markov Model players.

Names

- HMM Player: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

is_stochastic () → bool

Determines if the player is stochastic.

name = 'HMM Player'

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

class axelrod.strategies.hmm.**SimpleHMM** (*transitions_C, transitions_D, emission_probabilities, initial_state*) → None

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.

class axelrod.strategies.hunter.**AlternatorHunter** → None

A player who hunts for alternators.

Names:

- Alternator Hunter: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Alternator Hunter'

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

class axelrod.strategies.hunter.CooperatorHunter

A player who hunts for cooperators.

Names:

- Cooperator Hunter: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'Cooperator Hunter'

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

class axelrod.strategies.hunter.CycleHunter → None

Hunts strategies that play cyclically, like any of the Cyclers, Alternator, etc.

Names:

- Cycle Hunter: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'Cycle Hunter'

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

class axelrod.strategies.hunter.DefectorHunter

A player who hunts for defectors.

Names:

- Defector Hunter: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'Defector Hunter'

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

class axelrod.strategies.hunter.EventualCycleHunter → None

Hunts strategies that eventually play cyclically.

Names:

- Eventual Cycle Hunter: Original name by Marc Harper

name = 'Eventual Cycle Hunter'

strategy (*opponent: axelrod.player.Player*) → None

class axelrod.strategies.hunter.MathConstantHunter

A player who hunts for mathematical constant players.

Names:

Math Constant Hunter: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'Math Constant Hunter'

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` → None

A player who hunts for random players.

Names:

- Random Hunter: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Random Hunter'

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.

`axelrod.strategies.hunter.is_alternator` (*history: typing.List[axelrod.action.Action]*) → bool

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Inverse'

static 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.EvolvedLookerUp1_1_1` → None

A 1 1 1 Lookerup trained with an evolutionary algorithm.

Names:

- Evolved Lookerup 1 1 1: Original name by Marc Harper

name = 'EvolvedLookerUp1_1_1'

class `axelrod.strategies.lookerup.EvolvedLookerUp2_2_2` → None

A 2 2 2 Lookerup trained with an evolutionary algorithm.

Names:

- Evolved Lookerup 2 2 2: Original name by Marc Harper

name = 'EvolvedLookerUp2_2_2'

class `axelrod.strategies.lookerup.LookerUp` (*lookup_dict: dict = None, initial_actions: tuple = None, pattern: typing.Any = None, parameters: axelrod.strategies.lookerup.Plays = None*) → 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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

default_tft_lookup_table = {Plays(self_plays=(), op_plays=(D,), op_openings=()): D, Plays(self_plays=(), op_plays=(C,), op_openings=()): C}

lookup_dict

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'

name = 'LookerUp'

strategy (opponent: axelrod.player.Player) → Reaction

class `axelrod.strategies.lookerup.LookupTable` (*lookup_dict: dict*) → None
 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.

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'

classmethod `from_pattern` (*pattern: tuple, player_depth: int, op_depth: int, op_openings_depth: int*)

get (*plays: tuple, op_plays: tuple, op_openings: tuple*) → typing.Any

op_depth

op_openings_depth

player_depth

table_depth

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** → None
A lookup table based strategy.

Names:

•Winner12: [Mathieu2015]

name = 'Winner12'

class axelrod.strategies.lookerup.**Winner21** → None
A lookup table based strategy.

Names:

•Winner21: [Mathieu2015]

name = 'Winner21'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'memory_depth': inf, 'inspects_source': False, 'long_**strategy** (*opponent: axelrod.player.Player*) → axelrod.action.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

name = '\$\phi\$'

ratio = 1.618033988749895

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

name = '\$\pi\$'

ratio = 3.141592653589793

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

name = '\$e\$'

ratio = 2.718281828459045

class `axelrod.strategies.memorytwo.MEM2` → None

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: [\[Li2014\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'MEM2'

strategy (*opponent*: `axelrod.player.Player`) → `axelrod.action.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: [\[Akin2015\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'ALLCorALLD'

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

class `axelrod.strategies.memoryone.FirmButFair` → None

A strategy that cooperates on the first move, and cooperates except after receiving a sucker payoff.

Names:

- Firm But Fair: [*Frean1994*]

name = 'Firm But Fair'

class `axelrod.strategies.memoryone.GTFT` (*p: float = None*) → None

Generous Tit For Tat Strategy.

Names:

- Generous Tit For Tat: [*Nowak1993*]
- Naive peace maker: [*Gaudesi2016*]
- Soft Joss: [*Gaudesi2016*]

classifier = {'manipulates_state': False, 'makes_use_of': {'game'}, 'stochastic': True, 'inspects_source': False, 'long

name = 'GTFT'

receive_match_attributes ()

class `axelrod.strategies.memoryone.MemoryOnePlayer` (*four_vector: typing.Tuple[float, float, float, float] = None, initial: axelrod.action.Action = C*) → None

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)), defaults to Win-Stay Lose-Shift. Intended to be used as an abstract base class or to at least be supplied with a initializing `four_vector`.

Names

- Memory One: [*Nowak1990*]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Generic Memory One Player'

set_four_vector (*four_vector: typing.Tuple[float, float, float, float]*)

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

class `axelrod.strategies.memoryone.ReactivePlayer` (*probabilities: typing.Tuple[float, float]*) → None

A generic reactive player. Defined by 2 probabilities conditional on the opponent's last move: P(C|C), P(C|D).

Names:

- Reactive: [*Nowak1989*]

name = 'Reactive Player'

class `axelrod.strategies.memoryone.SoftJoss` (*q: float = 0.9*) → None

Defects with probability 0.9 when the opponent defects, otherwise emulates Tit-For-Tat.

Names:

- Soft Joss: [*Prison1998*]

name = 'Soft Joss'

class `axelrod.strategies.memoryone.StochasticCooperator` → None

Stochastic Cooperator.

Names:

- Stochastic Cooperator: [\[Adami2013\]](#)

name = 'Stochastic Cooperator'

class `axelrod.strategies.memoryone.StochasticWSLS` (*ep: float = 0.05*) → None
 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

name = 'Stochastic WSLS'

class `axelrod.strategies.memoryone.WinShiftLoseStay` (*initial: axelrod.action.Action = D*)
 → None

Win-Shift Lose-Stay, also called Reverse Pavlov.

Names:

- WSLS: [\[Li2011\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'Win-Shift Lose-Stay'

class `axelrod.strategies.memoryone.WinStayLoseShift` (*initial: axelrod.action.Action = C*)
 → None

Win-Stay Lose-Shift, also called Pavlov.

Names:

- Win Stay Lose Shift: [\[Nowak1993\]](#)

- WSLS: [\[Stewart2012\]](#)

- Pavlov: [\[Kraines1989\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'Win-Stay Lose-Shift'

class `axelrod.strategies.meta.MetaHunter`

A player who uses a selection of hunters.

Names

- Meta Hunter: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

static meta_strategy (*results, opponent*)

name = 'Meta Hunter'

class `axelrod.strategies.meta.MetaHunterAggressive` (*team=None*)

A player who uses a selection of hunters.

Names

- Meta Hunter Aggressive: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

static meta_strategy (*results, opponent*)

name = 'Meta Hunter Aggressive'

class `axelrod.strategies.meta.MetaMajority` (*team=None*)
 A player who goes by the majority vote of all other non-meta players.

Names:

- Meta Marjority: Original name by Karol Langner

static meta_strategy (*results, opponent*)

name = 'Meta Majority'

class `axelrod.strategies.meta.MetaMajorityFiniteMemory`
 MetaMajority with the team of Finite Memory Players

Names

- Meta Majority Finite Memory: Original name by Marc Harper

name = 'Meta Majority Finite Memory'

class `axelrod.strategies.meta.MetaMajorityLongMemory`
 MetaMajority with the team of Long (infinite) Memory Players

Names

- Meta Majority Long Memory: Original name by Marc Harper

name = 'Meta Majority Long Memory'

class `axelrod.strategies.meta.MetaMajorityMemoryOne`
 MetaMajority with the team of Memory One players

Names

- Meta Majority Memory One: Original name by Marc Harper

name = 'Meta Majority Memory One'

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*)

name = 'Meta Minority'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

meta_strategy (*results, opponent*)

Using the `numpy.random.choice` function to sample with weights

name = 'Meta Mixer'

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

classifier = {'manipulates_state': False, 'makes_use_of': {'length', 'game'}, 'stochastic': True, 'inspects_source': False}

meta_strategy (*results, opponent*)

Determine the meta result based on results of all players. Override this function in child classes.

name = 'Meta Player'

strategy (*opponent*)

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*)

name = 'Meta Winner'

class `axelrod.strategies.meta.MetaWinnerDeterministic`

Meta Winner with the team of Deterministic Players.

Names

- Meta Winner Deterministic: Original name by Marc Harper

name = 'Meta Winner Deterministic'

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.

Names:

- Meta Winner Ensemble: Original name by Marc Harper

meta_strategy (*results, opponent*)

name = 'Meta Winner Ensemble'

class `axelrod.strategies.meta.MetaWinnerFiniteMemory`

MetaWinner with the team of Finite Memory Players

Names

- Meta Winner Finite Memory: Original name by Marc Harper

name = 'Meta Winner Finite Memory'

class `axelrod.strategies.meta.MetaWinnerLongMemory`

MetaWinner with the team of Long (infinite) Memory Players

Names

- Meta Winner Long Memory: Original name by Marc Harper

name = 'Meta Winner Long Memory'

class `axelrod.strategies.meta.MetaWinnerMemoryOne`

Meta Winner with the team of Memory One players

Names

- Meta Winner Memory Memory One: Original name by Marc Harper

name = 'Meta Winner Memory One'

class `axelrod.strategies.meta.MetaWinnerStochastic`

Meta Winner with the team of Stochastic Players.

Names

- Meta Winner Stochastic: Original name by Marc Harper

name = 'Meta Winner Stochastic'

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

name = 'NMWE Deterministic'

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

name = 'NMWE Finite Memory'

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

name = 'NMWE Long Memory'

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

name = 'NMWE Memory One'

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

name = 'NMWE Stochastic'

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

```
classifier = {'manipulates_state': False, 'makes_use_of': {'length', 'game'}, 'stochastic': True, 'inspects_source': False}
decorator = <axelrod.strategy_transformers.StrategyTransformerFactory.<locals>.Decorator object>
name = 'Nice Meta Winner'
original_class
    alias of MetaWinner
strategy (opponent)
```

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.

Names:

- Meta Winner Ensemble: Original name by Marc Harper

```
classifier = {'manipulates_state': False, 'makes_use_of': {'length', 'game'}, 'stochastic': True, 'inspects_source': False}
decorator = <axelrod.strategy_transformers.StrategyTransformerFactory.<locals>.Decorator object>
name = 'Nice Meta Winner Ensemble'
original_class
    alias of MetaWinnerEnsemble
strategy (opponent)
```

class axelrod.strategies.mindcontrol.**MindBender**

A player that changes the opponent's strategy by modifying the internal dictionary.

Names

- Mind Bender: Original name by Karol Langner

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': False}
name = 'Mind Bender'
static strategy (opponent: axelrod.player.Player) → axelrod.action.Action
```

class axelrod.strategies.mindcontrol.**MindController**

A player that changes the opponents strategy to cooperate.

Names

- Mind Controller: Original name by Karol Langner

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': False}
name = 'Mind Controller'
static strategy (opponent: axelrod.player.Player) → axelrod.action.Action
    Alters the opponents strategy method to be a lambda function which always returns C. This player will then always return D to take advantage of this
```

class axelrod.strategies.mindcontrol.**MindWarper**

A player that changes the opponent's strategy but blocks changes to its own.

Names

- Mind Warper: Original name by Karol Langner

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': False}
```


name = 'Mind Warper'

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

The player classes in this module do not obey standard rules of the IPD (as indicated by their classifier). We do not recommend putting a lot of time in to optimising them.

class axelrod.strategies.mindreader.**MindReader**

A player that looks ahead at what the opponent will do and decides what to do.

Names:

- Mind reader: Original name by Jason Young

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': True, 'long_run'

static foil_strategy_inspection () → axelrod.action.Action

Foils `_strategy_utils.inspect_strategy` and `_strategy_utils.look_ahead`

name = 'Mind Reader'

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

Pretends to play the opponent a number of times before each match. The primary purpose is to look far enough ahead to see if a defect will be punished by the opponent.

class axelrod.strategies.mindreader.**MirrorMindReader**

A player that will mirror whatever strategy it is playing against by cheating and calling the opponent's strategy function instead of its own.

Names:

- Protected Mind reader: Original name by Brice Fernandes

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': True, 'long_run'

static foil_strategy_inspection () → axelrod.action.Action

Foils `_strategy_utils.inspect_strategy` and `_strategy_utils.look_ahead`

name = 'Mirror Mind Reader'

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

Will read the mind of the opponent and play the opponent's strategy.

class axelrod.strategies.mindreader.**ProtectedMindReader**

A player that looks ahead at what the opponent will do and decides what to do. It is also protected from mind control strategies

Names:

- Protected Mind reader: Original name by Jason Young

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': True, 'long_run'

name = 'Protected Mind Reader'

class axelrod.strategies.mutual.**Desperate**

A player that only cooperates after mutual defection.

Names:

- Desperate: [\[Berg2015\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run'

name = 'Desperate'

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

class `axelrod.strategies.mutual.Hopeless`

A player that only defects after mutual cooperation.

Names:

•Hopeless: [*Berg2015*]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Hopeless'

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

class `axelrod.strategies.mutual.Willing`

A player that only defects after mutual defection.

Names:

•Willing: [*Berg2015*]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Willing'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Negation'

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

class `axelrod.strategies.oncebitten.FoolMeForever`

Fool me once, shame on me. Teach a man to fool me and I'll be fooled for the rest of my life.

Names:

•Fool Me Forever: Original name by Marc Harper

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Fool Me Forever'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Fool Me Once'

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

class `axelrod.strategies.oncebitten.ForgetfulFoolMeOnce` (*forget_probability: float = 0.05*) → None

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run': True}

name = 'Forgetful Fool Me Once'

reset ()

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

class `axelrod.strategies.oncebitten.OnceBitten` → None

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'Once Bitten'

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]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'memory_depth': inf, 'inspects_source': False, 'long_run': True}

name = 'CollectiveStrategy'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'memory_depth': inf, 'inspects_source': False, 'long_run': True}

name = 'Hard Prober'

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

class `axelrod.strategies.prober.NaiveProber` (*p: float = 0.1*) → None

Like tit-for-tat, but it occasionally defects with a small probability.

Names:

- Naive Prober: [Li2011]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run': True}

name = 'Naive Prober'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'memory_depth': inf, 'inspects_source': False, 'long_

name = 'Prober'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'memory_depth': inf, 'inspects_source': False, 'long_

name = 'Prober 2'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'memory_depth': inf, 'inspects_source': False, 'long_

name = 'Prober 3'

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

class axelrod.strategies.prober.**Prober4** → None

Plays C, C, D, C, D, D, D, C, C, D, C, D, C, 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*]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'memory_depth': inf, 'inspects_source': False, 'long_

name = 'Prober 4'

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

class axelrod.strategies.prober.**RemorsefulProber** (*p: float = 0.1*) → None

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: [\[Li2011\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Remorseful Prober'

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

class axelrod.strategies.punisher.**InversePunisher** → None

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 $1 \leq \text{mem_length} \leq 20$. This time mem_length is proportional to the amount of time the opponent has played C.

Names:

- Inverse Punisher: Original name by Geraint Palmer

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Inverse Punisher'

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: [\[Eckhart2015\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Level Punisher'

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

class axelrod.strategies.punisher.**Punisher** → None

A player starts by cooperating however will defect if at any point the opponent has defected, but forgets after mem_length matches, with $1 \leq \text{mem_length} \leq 20$ 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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Punisher'

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.qlearner.**ArrogantQLearner** → None

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

```
discount_rate = 0.1
learning_rate = 0.9
name = 'Arrogant QLearner'
```

class `axelrod.strategies.qlearner.CautiousQLearner` → None

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

```
discount_rate = 0.1
learning_rate = 0.1
name = 'Cautious QLearner'
```

class `axelrod.strategies.qlearner.HesitantQLearner` → None

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

```
discount_rate = 0.9
learning_rate = 0.1
name = 'Hesitant QLearner'
```

class `axelrod.strategies.qlearner.RiskyQLearner` → None

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

```
action_selection_parameter = 0.1
classifier = {'manipulates_state': False, 'makes_use_of': {'game'}, 'stochastic': True, 'inspects_source': False, 'long-
discount_rate = 0.9
find_reward (opponent: axelrod.player.Player) → typing.Dict[axelrod.action.Action, typ-
ing.Dict[axelrod.action.Action, typing.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
learning_rate = 0.9
memory_length = 12
name = 'Risky QLearner'
perform_q_learning (prev_state: str, state: str, action: axelrod.action.Action, reward)
    Performs the qlearning algorithm
receive_match_attributes ()
```

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*) → None

A player who randomly chooses between cooperating and defecting.

This strategy came 15th in Axelrod's original tournament.

Names:

- Random: [Axelrod1980]

- Lunatic: [Tzafestas2000]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run': True}

name = 'Random'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'DoubleResurrection'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'Resurrection'

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

class axelrod.strategies.retaliate.**LimitedRetaliate** (*retaliation_threshold: float = 0.1,*

retaliation_limit: int = 20) → None

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'Limited Retaliate'

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) → None

LimitedRetaliate player with a threshold of 8 percent and a retaliation limit of 15.

Names:

- Limited Retaliate 2: Original name by Owen Campbell

name = 'Limited Retaliate 2'

class axelrod.strategies.retaliate.**LimitedRetaliate3** (*retaliation_threshold: float = 0.05,*
retaliation_limit: int = 20) → None

LimitedRetaliate player with a threshold of 5 percent and a retaliation limit of 20.

Names:

- Limited Retaliate 3: Original name by Owen Campbell

name = 'Limited Retaliate 3'

class axelrod.strategies.retaliate.**Retaliate** (*retaliation_threshold: float = 0.1*) → None

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': False}

name = 'Retaliate'

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*) → None

Retaliate player with a threshold of 8 percent.

Names:

- Retaliate 2: Original name by Owen Campbell

name = 'Retaliate 2'

class axelrod.strategies.retaliate.**Retaliate3** (*retaliation_threshold: float = 0.05*) → None

Retaliate player with a threshold of 5 percent.

Names:

- Retaliate 3: Original name by Owen Campbell

name = 'Retaliate 3'

class axelrod.strategies.sequence_player.**SequencePlayer** (*generator_function: function,*
generator_args: typing.Tuple
= ()) → None

Abstract base class for players that use a generated sequence to determine their plays.

Names:

- Sequence Player: Original name by Marc Harper

meta_strategy (*value: int*) → None

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

class axelrod.strategies.sequence_player.**ThueMorse** → None

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True, 'name': 'ThueMorse'}

class axelrod.strategies.sequence_player.**ThueMorseInverse** → None

A player who plays the inverse of the Thue-Morse sequence.

Names:

- Inverse Thue Morse: Original name by Geraint Palmer

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True, 'name': 'ThueMorseInverse'}

meta_strategy (*value: int*) → axelrod.action.Action

name = 'ThueMorseInverse'

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: [\[Andre2013\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True, 'name': 'ShortMem'}

static strategy (*opponent: axelrod.player.Player*) → axelrod.action.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 $0.95 > \text{abs}(f) > 0.3$, rational behavior; follows TitForTat algorithm. 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: [\[Andre2013\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run': True, 'name': 'SelfSteem'}

name = 'SelfSteem'

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

classifier = {'manipulates_state': False, 'makes_use_of': {'length', 'game'}, 'stochastic': True, 'inspects_source': False}

decorator = <*axelrod.strategy_transformers.StrategyTransformerFactory*.<locals>.Decorator object>

name = 'Stalker'

original_class

alias of *Stalker*

strategy (*opponent*)

class *axelrod.strategies.titfortat.AdaptiveTitForTat* (*rate*: *float = 0.5*) → None

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*]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}

name = 'Adaptive Tit For Tat'

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

world = 0.5

class `axelrod.strategies.titfortat.Alexei`
 Plays similar to Tit-for-Tat, but always defect on last turn.

Names:

- Alexei: [\[LessWrong2011\]](#)

```

classifier = {'manipulates_state': False, 'makes_use_of': {'length'}, 'stochastic': False, 'inspects_source': False, 'long_run': True}
decorator = <axelrod.strategy_transformers.StrategyTransformerFactory.<locals>.Decorator object>
name = 'Alexei'
original_class
  alias of Alexei
strategy (opponent)
  
```

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: [\[Hilbe2013\]](#)

```

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'Anti Tit For Tat'
static strategy (opponent: axelrod.player.Player) → axelrod.action.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: [\[Nachbar1992\]](#)

```

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'Bully'
static strategy (opponent: axelrod.player.Player) → axelrod.action.Action
  
```

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: [\[Axelrod1995\]](#)

```

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
decorator = <axelrod.strategy_transformers.StrategyTransformerFactory.<locals>.Decorator object>
name = 'Contrite Tit For Tat'
original_class
  alias of ContriteTitForTat
strategy (opponent)
  
```

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.

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Dynamic Two Tits For Tat'

static strategy (*opponent*)

class `axelrod.strategies.titfortat.EngineNier`

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*]

classifier = {'manipulates_state': False, 'makes_use_of': {'length'}, 'stochastic': False, 'inspects_source': False, 'long

decorator = <axelrod.strategy_transformers.StrategyTransformerFactory.<locals>.Decorator object>

name = 'EngineNier'

original_class

alias of *EngineNier*

strategy (*opponent*)

class `axelrod.strategies.titfortat.Gradual` → None

A player that punishes defections with a growing number of defections but after punishing enters a calming state and cooperates no matter what the opponent does for two rounds.

Names:

- Gradual: [*Beaufils1997*]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Gradual'

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

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Hard Tit For 2 Tats'

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

```

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'Hard Tit For Tat'
static strategy (opponent: axelrod.player.Player) → axelrod.action.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*]

```

classifier = {'manipulates_state': False, 'makes_use_of': {'length'}, 'stochastic': True, 'inspects_source': False, 'long_run': True}
decorator = <axelrod.strategy_transformers.StrategyTransformerFactory.<locals>.Decorator object>
name = 'Michaelos'
original_class
  alias of Michaelos
strategy (opponent)

```

class axelrod.strategies.titfortat.**NTitsForMTats** (*N: int = 3, M: int = 2*) → None

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

```

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'N Tit(s) For M Tat(s)'
strategy (opponent: axelrod.player.Player) → axelrod.action.Action

```

class axelrod.strategies.titfortat.**OmegaTFT** (*deadlock_threshold: int = 3, randomness_threshold: int = 8*) → None

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*]

```

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'Omega TFT'
strategy (opponent: axelrod.player.Player) → axelrod.action.Action

```

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*]

```

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run': True}
name = 'Slow Tit For Two Tats 2'
strategy (opponent: axelrod.player.Player) → axelrod.action.Action

```

class `axelrod.strategies.titfortat.SneakyTitForTat`

Tries defecting once and repents if punished.

Names:

- Sneaky Tit For Tat: Original name by Karol Langner

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Sneaky Tit For Tat'

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

class `axelrod.strategies.titfortat.SpitefulTitForTat` → None

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: [\[Prison1998\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Spiteful Tit For Tat'

strategy (*opponent*: `axelrod.player.Player`) → `axelrod.action.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: [\[Beaufils1997\]](#)

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Suspicious Tit For Tat'

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

class `axelrod.strategies.titfortat.TitFor2Tats`

A player starts by cooperating and then defects only after two defects by opponent.

Names:

- Tit for two Tats: [\[Axelrod1984\]](#)

- Slow tit for two tats: Original name by Ranjini Das

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run

name = 'Tit For 2 Tats'

static strategy (*opponent*: `axelrod.player.Player`) → `axelrod.action.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: [\[Axelrod1980\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Tit For Tat'
```

```
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: [\[Axelrod1984\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'Two Tits For Tat'
```

```
static strategy (opponent: axelrod.player.Player) → axelrod.action.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: [\[Andre2013\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': False, 'inspects_source': False, 'long_run'
name = 'VeryBad'
```

```
static strategy (opponent: axelrod.player.Player) → axelrod.action.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

```
classifier = {'manipulates_state': False, 'makes_use_of': {'length'}, 'stochastic': True, 'inspects_source': False, 'long_run'
name = 'Knowledgeable Worse and Worse'
```

```
strategy (opponent: axelrod.player.Player) → axelrod.action.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: [\[Prison1998\]](#)

```
classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run'
name = 'Worse and Worse'
```

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

class `axelrod.strategies.worse_and_worse.WorseAndWorse2`

Plays as tit for tat during the first 20 moves. Then defects with probability $(\text{current turn} - 20) / \text{current turn}$. Therefore it is more and more likely to defect as the round goes on.

Names:

- Worse and Worse 2: [*Prison1998*]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Worse and Worse 2'

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

class `axelrod.strategies.worse_and_worse.WorseAndWorse3`

Cooperates in the first turn. Then defects with probability $\text{no. of opponent defects} / (\text{current turn} - 1)$. Therefore it is more likely to defect when the opponent defects for a larger proportion of the turns.

Names:

- Worse and Worse 3: [*Prison1998*]

classifier = {'manipulates_state': False, 'makes_use_of': set(), 'stochastic': True, 'inspects_source': False, 'long_run

name = 'Worse and Worse 3'

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

class `axelrod.strategies.zero_determinant.LRPlayer` (*phi: float = 0.2, s: float = 0.1, l: float = 1*) → None

Abstraction for Linear Relation players. These players enforce a linear difference in stationary payoffs $s * (S_{xy} - l) = S_{yx} - l$, with $0 \leq l \leq R$. The parameter *s* is called the slope and the parameter *l* the baseline payoff. For extortionate strategies, the extortion factor is the inverse of the slope.

This parameterization is Equation 14 in <http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0077886>. See Figure 2 of the article for a more in-depth explanation.

Names:

- Linear Relation player: [*Hilbe2013*]

classifier = {'manipulates_state': False, 'makes_use_of': {'game'}, 'stochastic': True, 'inspects_source': False, 'long

name = 'LinearRelation'

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 = 0.11111111111111111, s: float = 0.5*) → None

An Extortionate Zero Determinant Strategy with $l=P$.

Names:

- Extort-2: [*Stewart2012*]

name = 'ZD-Extort-2'

receive_match_attributes ()

class `axelrod.strategies.zero_determinant.ZDExtort2v2` (*phi: float = 0.125, s: float = 0.5, l: float = 1*) → None

An Extortionate Zero Determinant Strategy with $l=1$.

Names:

- EXTORT2: [\[Kuhn2017\]](#)

name = 'ZD-Extort-2 v2'

class axelrod.strategies.zero_determinant.**ZDExtort3** (*phi: float = 0.11538461538461539, s: float = 0.3333333333333333, l: float = 1*) → None

An extortionate strategy from Press and Dyson's paper with an extortion factor of 3.

Names:

- ZDExtort3: Original name by Marc Harper

- Unnamed: [\[Press2012\]](#)

name = 'ZD-Extort3'

class axelrod.strategies.zero_determinant.**ZDExtort4** (*phi: float = 0.23529411764705882, s: float = 0.25, l: float = 1*) → None

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

name = 'ZD-Extort-4'

class axelrod.strategies.zero_determinant.**ZDExtortion** (*phi: float = 0.2, s: float = 0.1, l: float = 1*) → None

An example ZD Extortion player.

Names:

- ZDExtortion: [\[Roemheld2013\]](#)

name = 'ZD-Extortion'

class axelrod.strategies.zero_determinant.**ZDGTFT2** (*phi: float = 0.25, s: float = 0.5*) → None

A Generous Zero Determinant Strategy with l=R.

Names:

- ZDGTFT-2: [\[Stewart2012\]](#)

name = 'ZD-GTFT-2'

receive_match_attributes ()

class axelrod.strategies.zero_determinant.**ZDGen2** (*phi: float = 0.125, s: float = 0.5, l: float = 3*) → None

A Generous Zero Determinant Strategy with l=3.

Names:

- GEN2: [\[Kuhn2017\]](#)

name = 'ZD-GEN-2'

class axelrod.strategies.zero_determinant.**ZDMischief** (*phi: float = 0.1, s: float = 0.0, l: float = 1*) → None

An example ZD Mischief player.

Names:

- ZDMischief: [\[Roemheld2013\]](#)

```
name = 'ZD-Mischief'
```

```
class axelrod.strategies.zero_determinant.ZDSet2 (phi: float = 0.25, s: float = 0.0, l: float = 2) → None
```

A Generous Zero Determinant Strategy with l=2.

Names:

- SET2: [Kuhn2017]

```
name = 'ZD-SET-2'
```

Bibliography

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

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()
>>> p1.play(p2) # This constitutes two 'plays' (p1 plays and p2 plays).
```

This is equivalent to `p2.play(p1)`. Either function invokes both `p1.strategy(p2)` and `p2.strategy(p1)`.

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 10 turns:

```
>>> p1, p2 = axl.Cooperator(), axl.Defector()
>>> for turn in range(10):
...     p1.play(p2)
>>> p1.history, p2.history
([C, C, C, C, C, C, C, C, C, C], [D, D, D, D, D, D, D, 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.

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](http://contributor-covenant.org/version/1/4), version 1.4, available at <http://contributor-covenant.org/version/1/4>

Citing 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](#).

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](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. and Delahaye, J. (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.
- [Eckhart2015] Eckhart Arnold (2016) CoopSim v0.9.9 beta 6. <https://github.com/jecki/CoopSim/>
- [Freen1994] Freen, Marcus R. "The Prisoner's Dilemma without Synchrony." *Proceedings: Biological Sciences*, vol. 257, no. 1348, 1994, pp. 75–79. www.jstor.org/stable/50253.
- [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.
- [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
- [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>
- [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>
- [Nachbar1992] Nachbar J., Evolution in the finitely repeated prisoner's dilemma, *Journal of Economic Behavior & Organization*, 19(3): 307-326, 1992.

- [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>
- [PD2017] <http://www.prisoners-dilemma.com/competition.html> (Accessed: 6 June 2017)
- [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.

a

axelrod.strategies.adaptive, 62
axelrod.strategies.alternator, 63
axelrod.strategies.ann, 63
axelrod.strategies.apavlov, 64
axelrod.strategies.appeaser, 64
axelrod.strategies.averagecopier, 65
axelrod.strategies.axelrod_first, 65
axelrod.strategies.axelrod_second, 69
axelrod.strategies.backstabber, 72
axelrod.strategies.better_and_better,
72
axelrod.strategies.calculator, 73
axelrod.strategies.cooperator, 73
axelrod.strategies.cycler, 73
axelrod.strategies.darwin, 75
axelrod.strategies.dbs, 75
axelrod.strategies.defector, 77
axelrod.strategies.doubler, 77
axelrod.strategies.finite_state_machines,
78
axelrod.strategies.forgiver, 81
axelrod.strategies.gambler, 81
axelrod.strategies.geller, 82
axelrod.strategies.gobymajority, 83
axelrod.strategies.gradualkiller, 85
axelrod.strategies.grudger, 85
axelrod.strategies.grumpy, 87
axelrod.strategies.handshake, 87
axelrod.strategies.hmm, 88
axelrod.strategies.hunter, 88
axelrod.strategies.inverse, 90
axelrod.strategies.lookerup, 90
axelrod.strategies.mathematicalconstants,
93
axelrod.strategies.memoryone, 94
axelrod.strategies.memorytwo, 94
axelrod.strategies.meta, 96
axelrod.strategies.mindcontrol, 100
axelrod.strategies.mindreader, 101
axelrod.strategies.mutual, 101
axelrod.strategies.negation, 102
axelrod.strategies.oncebitten, 102
axelrod.strategies.prober, 103
axelrod.strategies.punisher, 105
axelrod.strategies.qlearner, 105
axelrod.strategies.rand, 107
axelrod.strategies.resurrection, 107
axelrod.strategies.retaliate, 107
axelrod.strategies.selfsteem, 109
axelrod.strategies.sequence_player, 108
axelrod.strategies.shortmem, 109
axelrod.strategies.stalker, 110
axelrod.strategies.titfortat, 110
axelrod.strategies.verybad, 115
axelrod.strategies.worse_and_worse, 115
axelrod.strategies.zero_determinant, 116

A

- action_selection_parameter (axelrod.strategies.qlearner.RiskyQLearner attribute), 106
 action_to_int() (in module axelrod.strategies.dbs), 77
 activate() (in module axelrod.strategies.ann), 64
 Adaptive (class in axelrod.strategies.adaptive), 62
 AdaptiveTitForTat (class in axelrod.strategies.titfortat), 110
 Aggravater (class in axelrod.strategies.grudger), 85
 Alexei (class in axelrod.strategies.titfortat), 110
 ALLCorALLD (class in axelrod.strategies.memoryone), 94
 Alternator (class in axelrod.strategies.alternator), 63
 AlternatorHunter (class in axelrod.strategies.hunter), 88
 ANN (class in axelrod.strategies.ann), 63
 AntiCycler (class in axelrod.strategies.cycler), 73
 AntiTitForTat (class in axelrod.strategies.titfortat), 111
 APavlov2006 (class in axelrod.strategies.apavlov), 64
 APavlov2011 (class in axelrod.strategies.apavlov), 64
 Appeaser (class in axelrod.strategies.appeaser), 64
 ArrogantQLearner (class in axelrod.strategies.qlearner), 105
 AverageCopier (class in axelrod.strategies.averagecopier), 65
 axelrod.strategies.adaptive (module), 62
 axelrod.strategies.alternator (module), 63
 axelrod.strategies.ann (module), 63
 axelrod.strategies.apavlov (module), 64
 axelrod.strategies.appeaser (module), 64
 axelrod.strategies.averagecopier (module), 65
 axelrod.strategies.axelrod_first (module), 65
 axelrod.strategies.axelrod_second (module), 69
 axelrod.strategies.backstabber (module), 72
 axelrod.strategies.better_and_better (module), 72
 axelrod.strategies.calculator (module), 73
 axelrod.strategies.cooperator (module), 73
 axelrod.strategies.cycler (module), 73
 axelrod.strategies.darwin (module), 75
 axelrod.strategies.dbs (module), 75
 axelrod.strategies.defector (module), 77
 axelrod.strategies.doubler (module), 77
 axelrod.strategies.finite_state_machines (module), 78
 axelrod.strategies.forgiver (module), 81
 axelrod.strategies.gambler (module), 81
 axelrod.strategies.geller (module), 82
 axelrod.strategies.gobymajority (module), 83
 axelrod.strategies.gradualkiller (module), 85
 axelrod.strategies.grudger (module), 85
 axelrod.strategies.grumpy (module), 87
 axelrod.strategies.handshake (module), 87
 axelrod.strategies.hmm (module), 88
 axelrod.strategies.hunter (module), 88
 axelrod.strategies.inverse (module), 90
 axelrod.strategies.lookerup (module), 90
 axelrod.strategies.mathematicalconstants (module), 93
 axelrod.strategies.memoryone (module), 94
 axelrod.strategies.memorytwo (module), 94
 axelrod.strategies.meta (module), 96
 axelrod.strategies.mindcontrol (module), 100
 axelrod.strategies.mindreader (module), 101
 axelrod.strategies.mutual (module), 101
 axelrod.strategies.negation (module), 102
 axelrod.strategies.oncebitten (module), 102
 axelrod.strategies.prober (module), 103
 axelrod.strategies.punisher (module), 105
 axelrod.strategies.qlearner (module), 105
 axelrod.strategies.rand (module), 107
 axelrod.strategies.resurrection (module), 107
 axelrod.strategies.retaliate (module), 107
 axelrod.strategies.selfsteem (module), 109
 axelrod.strategies.sequence_player (module), 108
 axelrod.strategies.shortmem (module), 109
 axelrod.strategies.stalker (module), 110
 axelrod.strategies.titfortat (module), 110
 axelrod.strategies.verybad (module), 115
 axelrod.strategies.worse_and_worse (module), 115
 axelrod.strategies.zero_determinant (module), 116

B

- BackStabber (class in axelrod.strategies.backstabber), 72
- BetterAndBetter (class in axelrod.strategies.better_and_better), 72
- Bully (class in axelrod.strategies.tiffortat), 111

C

- Calculator (class in axelrod.strategies.calculator), 73
- CautiousQLearner (class in axelrod.strategies.qlearner), 106
- Champion (class in axelrod.strategies.axelrod_second), 69
- classifier (axelrod.strategies.adaptive.Adaptive attribute), 63
- classifier (axelrod.strategies.alternator.Alternator attribute), 63
- classifier (axelrod.strategies.ann.ANN attribute), 63
- classifier (axelrod.strategies.apavlov.APavlov2006 attribute), 64
- classifier (axelrod.strategies.apavlov.APavlov2011 attribute), 64
- classifier (axelrod.strategies appeaser.Appeaser attribute), 65
- classifier (axelrod.strategies.averagecopier.AverageCopier attribute), 65
- classifier (axelrod.strategies.averagecopier.NiceAverageCopier attribute), 65
- classifier (axelrod.strategies.axelrod_first.Davis attribute), 65
- classifier (axelrod.strategies.axelrod_first.Feld attribute), 66
- classifier (axelrod.strategies.axelrod_first.Grofman attribute), 66
- classifier (axelrod.strategies.axelrod_first.Nydegger attribute), 67
- classifier (axelrod.strategies.axelrod_first.RevisedDowning attribute), 67
- classifier (axelrod.strategies.axelrod_first.Shubik attribute), 67
- classifier (axelrod.strategies.axelrod_first.SteinAndRapoport attribute), 68
- classifier (axelrod.strategies.axelrod_first.TidemanAndChieruzzi attribute), 68
- classifier (axelrod.strategies.axelrod_first.Tulloch attribute), 68
- classifier (axelrod.strategies.axelrod_first.UnnamedStrategy attribute), 69
- classifier (axelrod.strategies.axelrod_second.Champion attribute), 69
- classifier (axelrod.strategies.axelrod_second.Eatherley attribute), 69
- classifier (axelrod.strategies.axelrod_second.Gladstein attribute), 70
- classifier (axelrod.strategies.axelrod_second.MoreGrofman attribute), 70
- classifier (axelrod.strategies.axelrod_second.Tester attribute), 70
- classifier (axelrod.strategies.axelrod_second.Tranquilizer attribute), 72
- classifier (axelrod.strategies.backstabber.BackStabber attribute), 72
- classifier (axelrod.strategies.backstabber.DoubleCrosser attribute), 72
- classifier (axelrod.strategies.better_and_better.BetterAndBetter attribute), 73
- classifier (axelrod.strategies.calculator.Calculator attribute), 73
- classifier (axelrod.strategies.cooperator.Cooperator attribute), 73
- classifier (axelrod.strategies.cooperator.TrickyCooperator attribute), 73
- classifier (axelrod.strategies.cycler.AntiCycler attribute), 73
- classifier (axelrod.strategies.cycler.Cycler attribute), 74
- classifier (axelrod.strategies.cycler.CyclerCCCCCD attribute), 74
- classifier (axelrod.strategies.cycler.CyclerCCCD attribute), 74
- classifier (axelrod.strategies.cycler.CyclerCCCDCCD attribute), 74
- classifier (axelrod.strategies.cycler.CyclerCCD attribute), 74
- classifier (axelrod.strategies.cycler.CyclerDC attribute), 75
- classifier (axelrod.strategies.cycler.CyclerDDC attribute), 75
- classifier (axelrod.strategies.darwin.Darwin attribute), 75
- classifier (axelrod.strategies.dbs.DB5 attribute), 76
- classifier (axelrod.strategies.defector.Defector attribute), 77
- classifier (axelrod.strategies.defector.TrickyDefector attribute), 77
- classifier (axelrod.strategies.doubler.Doubler attribute), 78
- classifier (axelrod.strategies.finite_state_machines.EvolvedFSM16 attribute), 78
- classifier (axelrod.strategies.finite_state_machines.EvolvedFSM16Noise05 attribute), 78
- classifier (axelrod.strategies.finite_state_machines.EvolvedFSM4 attribute), 78
- classifier (axelrod.strategies.finite_state_machines.Fortress3 attribute), 79
- classifier (axelrod.strategies.finite_state_machines.Fortress4 attribute), 79
- classifier (axelrod.strategies.finite_state_machines.FSMPlayer attribute), 78
- classifier (axelrod.strategies.finite_state_machines.Predator attribute), 78

- attribute), 79
- classifier (axelrod.strategies.finite_state_machines.Pun1 attribute), 79
- classifier (axelrod.strategies.finite_state_machines.Raider attribute), 79
- classifier (axelrod.strategies.finite_state_machines.Ripoff attribute), 79
- classifier (axelrod.strategies.finite_state_machines.SolutionB1 attribute), 80
- classifier (axelrod.strategies.finite_state_machines.SolutionB5 attribute), 80
- classifier (axelrod.strategies.finite_state_machines.TF1 attribute), 80
- classifier (axelrod.strategies.finite_state_machines.TF2 attribute), 80
- classifier (axelrod.strategies.finite_state_machines.TF3 attribute), 80
- classifier (axelrod.strategies.finite_state_machines.Thumper attribute), 80
- classifier (axelrod.strategies.forgiver.Forgiver attribute), 81
- classifier (axelrod.strategies.forgiver.ForgivingTitForTat attribute), 81
- classifier (axelrod.strategies.gambler.Gambler attribute), 81
- classifier (axelrod.strategies.gambler.ZDMem2 attribute), 82
- classifier (axelrod.strategies.geller.Geller attribute), 82
- classifier (axelrod.strategies.geller.GellerCooperator attribute), 83
- classifier (axelrod.strategies.geller.GellerDefector attribute), 83
- classifier (axelrod.strategies.gobymajority.GoByMajority attribute), 83
- classifier (axelrod.strategies.gobymajority.GoByMajority10 attribute), 83
- classifier (axelrod.strategies.gobymajority.GoByMajority20 attribute), 84
- classifier (axelrod.strategies.gobymajority.GoByMajority40 attribute), 84
- classifier (axelrod.strategies.gobymajority.GoByMajority5 attribute), 84
- classifier (axelrod.strategies.gobymajority.HardGoByMajority10 attribute), 84
- classifier (axelrod.strategies.gobymajority.HardGoByMajority20 attribute), 84
- classifier (axelrod.strategies.gobymajority.HardGoByMajority40 attribute), 85
- classifier (axelrod.strategies.gobymajority.HardGoByMajority5 attribute), 85
- classifier (axelrod.strategies.gradualkiller.GradualKiller attribute), 85
- classifier (axelrod.strategies.grudger.Aggravater attribute), 85
- classifier (axelrod.strategies.grudger.EasyGo attribute), 85
- classifier (axelrod.strategies.grudger.ForgetfulGrudger attribute), 86
- classifier (axelrod.strategies.grudger.GeneralSoftGrudger attribute), 86
- classifier (axelrod.strategies.grudger.Grudger attribute), 86
- classifier (axelrod.strategies.grudger.GrudgerAlternator attribute), 87
- classifier (axelrod.strategies.grudger.OppositeGrudger attribute), 87
- classifier (axelrod.strategies.grudger.SoftGrudger attribute), 87
- classifier (axelrod.strategies.grumpy.Grumpy attribute), 87
- classifier (axelrod.strategies.handshake.Handshake attribute), 87
- classifier (axelrod.strategies.hmm.EvolvedHMM5 attribute), 88
- classifier (axelrod.strategies.hmm.HMMPlayer attribute), 88
- classifier (axelrod.strategies.hunter.AlternatorHunter attribute), 88
- classifier (axelrod.strategies.hunter.CooperatorHunter attribute), 89
- classifier (axelrod.strategies.hunter.CycleHunter attribute), 89
- classifier (axelrod.strategies.hunter.DefectorHunter attribute), 89
- classifier (axelrod.strategies.hunter.MathConstantHunter attribute), 89
- classifier (axelrod.strategies.hunter.RandomHunter attribute), 90
- classifier (axelrod.strategies.inverse.Inverse attribute), 90
- classifier (axelrod.strategies.lookerup.LookerUp attribute), 91
- classifier (axelrod.strategies.mathematicalconstants.CotoDeRatio attribute), 93
- classifier (axelrod.strategies.memoryone.ALLCorALLD attribute), 94
- classifier (axelrod.strategies.memoryone.GTFT attribute), 95
- classifier (axelrod.strategies.memoryone.MemoryOnePlayer attribute), 95
- classifier (axelrod.strategies.memoryone.WinShiftLoseStay attribute), 96
- classifier (axelrod.strategies.memoryone.WinStayLoseShift attribute), 96
- classifier (axelrod.strategies.memorytwo.MEM2 attribute), 94
- classifier (axelrod.strategies.meta.MetaHunter attribute), 96
- classifier (axelrod.strategies.meta.MetaHunterAggressive

- attribute), 96
- classifier (axelrod.strategies.meta.MetaMixer attribute), 97
- classifier (axelrod.strategies.meta.MetaPlayer attribute), 98
- classifier (axelrod.strategies.meta.NiceMetaWinner attribute), 100
- classifier (axelrod.strategies.meta.NiceMetaWinnerEnsemble attribute), 100
- classifier (axelrod.strategies.mindcontrol.MindBender attribute), 100
- classifier (axelrod.strategies.mindcontrol.MindController attribute), 100
- classifier (axelrod.strategies.mindcontrol.MindWarper attribute), 100
- classifier (axelrod.strategies.mindreader.MindReader attribute), 101
- classifier (axelrod.strategies.mindreader.MirrorMindReader attribute), 101
- classifier (axelrod.strategies.mindreader.ProtectedMindReader attribute), 101
- classifier (axelrod.strategies.mutual.Desperate attribute), 101
- classifier (axelrod.strategies.mutual.Hopeless attribute), 102
- classifier (axelrod.strategies.mutual.Willing attribute), 102
- classifier (axelrod.strategies.negation.Negation attribute), 102
- classifier (axelrod.strategies.oncebitten.FoolMeForever attribute), 102
- classifier (axelrod.strategies.oncebitten.FoolMeOnce attribute), 102
- classifier (axelrod.strategies.oncebitten.ForgetfulFoolMeOnce attribute), 103
- classifier (axelrod.strategies.oncebitten.OnceBitten attribute), 103
- classifier (axelrod.strategies.prober.CollectiveStrategy attribute), 103
- classifier (axelrod.strategies.prober.HardProber attribute), 103
- classifier (axelrod.strategies.prober.NaiveProber attribute), 103
- classifier (axelrod.strategies.prober.Prober attribute), 104
- classifier (axelrod.strategies.prober.Prober2 attribute), 104
- classifier (axelrod.strategies.prober.Prober3 attribute), 104
- classifier (axelrod.strategies.prober.Prober4 attribute), 104
- classifier (axelrod.strategies.prober.RemorsefulProber attribute), 105
- classifier (axelrod.strategies.punisher.InversePunisher attribute), 105
- classifier (axelrod.strategies.punisher.LevelPunisher attribute), 105
- classifier (axelrod.strategies.punisher.Punisher attribute), 105
- classifier (axelrod.strategies.qlearner.RiskyQLearner attribute), 106
- classifier (axelrod.strategies.rand.Random attribute), 107
- classifier (axelrod.strategies.resurrection.DoubleResurrection attribute), 107
- classifier (axelrod.strategies.resurrection.Resurrection attribute), 107
- classifier (axelrod.strategies.retaliate.LimitedRetaliate attribute), 107
- classifier (axelrod.strategies.retaliate.Retaliate attribute), 108
- classifier (axelrod.strategies.selfsteem.SelfSteem attribute), 109
- classifier (axelrod.strategies.sequence_player.ThueMorse attribute), 109
- classifier (axelrod.strategies.sequence_player.ThueMorseInverse attribute), 109
- classifier (axelrod.strategies.shortmem.ShortMem attribute), 109
- classifier (axelrod.strategies.stalker.Stalker attribute), 110
- classifier (axelrod.strategies.titfortat.AdaptiveTitForTat attribute), 110
- classifier (axelrod.strategies.titfortat.Alexei attribute), 111
- classifier (axelrod.strategies.titfortat.AntiTitForTat attribute), 111
- classifier (axelrod.strategies.titfortat.Bully attribute), 111
- classifier (axelrod.strategies.titfortat.ContriteTitForTat attribute), 111
- classifier (axelrod.strategies.titfortat.DynamicTwoTitsForTat attribute), 112
- classifier (axelrod.strategies.titfortat.EugeneNier attribute), 112
- classifier (axelrod.strategies.titfortat.Gradual attribute), 112
- classifier (axelrod.strategies.titfortat.HardTitFor2Tats attribute), 112
- classifier (axelrod.strategies.titfortat.HardTitForTat attribute), 112
- classifier (axelrod.strategies.titfortat.Michaelos attribute), 113
- classifier (axelrod.strategies.titfortat.NTitsForMTats attribute), 113
- classifier (axelrod.strategies.titfortat.OmegaTFT attribute), 113
- classifier (axelrod.strategies.titfortat.SlowTitForTwoTats2 attribute), 113
- classifier (axelrod.strategies.titfortat.SneakyTitForTat attribute), 114
- classifier (axelrod.strategies.titfortat.SpitefulTitForTat attribute), 114

- tribute), 114
- classifier (axelrod.strategies.titfortat.SuspiciousTitForTat attribute), 114
- classifier (axelrod.strategies.titfortat.TitFor2Tats attribute), 114
- classifier (axelrod.strategies.titfortat.TitForTat attribute), 115
- classifier (axelrod.strategies.titfortat.TwoTitsForTat attribute), 115
- classifier (axelrod.strategies.verybad.VeryBad attribute), 115
- classifier (axelrod.strategies.worse_and_worse.KnowledgeableWorseAndWorse attribute), 115
- classifier (axelrod.strategies.worse_and_worse.WorseAndWorse attribute), 115
- classifier (axelrod.strategies.worse_and_worse.WorseAndWorse2 attribute), 116
- classifier (axelrod.strategies.worse_and_worse.WorseAndWorse3 attribute), 116
- classifier (axelrod.strategies.zero_determinant.LRPlayer attribute), 116
- CollectiveStrategy (class in axelrod.strategies.prober), 103
- compute_features() (in module axelrod.strategies.ann), 64
- compute_prob_rule() (axelrod.strategies.dbs.DBS method), 76
- ContriteTitForTat (class in axelrod.strategies.titfortat), 111
- Cooperator (class in axelrod.strategies.cooperator), 73
- CooperatorHunter (class in axelrod.strategies.hunter), 89
- CotoDeRatio (class in axelrod.strategies.mathematicalconstants), 93
- create_lookup_table_keys() (in module axelrod.strategies.lookerup), 93
- create_policy() (in module axelrod.strategies.dbs), 77
- CycleHunter (class in axelrod.strategies.hunter), 89
- Cycler (class in axelrod.strategies.cycler), 74
- CyclerCCCCCCD (class in axelrod.strategies.cycler), 74
- CyclerCCCD (class in axelrod.strategies.cycler), 74
- CyclerCCDCD (class in axelrod.strategies.cycler), 74
- CyclerCCD (class in axelrod.strategies.cycler), 74
- CyclerDC (class in axelrod.strategies.cycler), 74
- CyclerDDC (class in axelrod.strategies.cycler), 75
- ## D
- Darwin (class in axelrod.strategies.darwin), 75
- Davis (class in axelrod.strategies.axelrod_first), 65
- DBS (class in axelrod.strategies.dbs), 75
- decorator (axelrod.strategies.axelrod_first.SteinAndRapoport attribute), 68
- decorator (axelrod.strategies.backstabber.BackStabber attribute), 72
- decorator (axelrod.strategies.backstabber.DoubleCrosser attribute), 72
- decorator (axelrod.strategies.gradualkiller.GradualKiller attribute), 85
- decorator (axelrod.strategies.meta.NiceMetaWinner attribute), 100
- decorator (axelrod.strategies.meta.NiceMetaWinnerEnsemble attribute), 100
- decorator (axelrod.strategies.stalker.Stalker attribute), 110
- decorator (axelrod.strategies.titfortat.Alexei attribute), 111
- decorator (axelrod.strategies.titfortat.ContriteTitForTat attribute), 111
- decorator (axelrod.strategies.titfortat.EugeneNier attribute), 112
- decorator (axelrod.strategies.titfortat.Michaelos attribute), 113
- default_tft_lookup_table (axelrod.strategies.lookerup.LookerUp attribute), 91
- Defector (class in axelrod.strategies.defector), 77
- DefectorHunter (class in axelrod.strategies.hunter), 89
- Desperate (class in axelrod.strategies.mutual), 101
- DeterministicNode (class in axelrod.strategies.dbs), 76
- dictionary (axelrod.strategies.lookerup.LookupTable attribute), 92
- discount_rate (axelrod.strategies.qlearner.ArrogantQLearner attribute), 105
- discount_rate (axelrod.strategies.qlearner.CautiousQLearner attribute), 106
- discount_rate (axelrod.strategies.qlearner.HesitantQLearner attribute), 106
- discount_rate (axelrod.strategies.qlearner.RiskyQLearner attribute), 106
- display() (axelrod.strategies.lookerup.LookupTable method), 92
- DoubleCrosser (class in axelrod.strategies.backstabber), 72
- Doubler (class in axelrod.strategies.doubler), 77
- DoubleResurrection (class in axelrod.strategies.resurrection), 107
- DynamicTwoTitsForTat (class in axelrod.strategies.titfortat), 111
- ## E
- e (class in axelrod.strategies.mathematicalconstants), 94
- EasyGo (class in axelrod.strategies.grudger), 85
- Eatherley (class in axelrod.strategies.axelrod_second), 69
- EugeneNier (class in axelrod.strategies.titfortat), 112
- EventualCycleHunter (class in axelrod.strategies.hunter), 89
- EvolvedANN (class in axelrod.strategies.ann), 63
- EvolvedANN5 (class in axelrod.strategies.ann), 63
- EvolvedANNNoise05 (class in axelrod.strategies.ann), 63

- EvolvedFSM16 (class in axelrod.strategies.finite_state_machines), 78
- EvolvedFSM16Noise05 (class in axelrod.strategies.finite_state_machines), 78
- EvolvedFSM4 (class in axelrod.strategies.finite_state_machines), 78
- EvolvedHMM5 (class in axelrod.strategies.hmm), 88
- EvolvedLookerUp1_1_1 (class in axelrod.strategies.lookerup), 90
- EvolvedLookerUp2_2_2 (class in axelrod.strategies.lookerup), 90
- extended_strategy() (axelrod.strategies.calculator.Calculator method), 73
- ## F
- Feld (class in axelrod.strategies.axelrod_first), 65
- find_reward() (axelrod.strategies.qlearner.RiskyQLearner method), 106
- find_state() (axelrod.strategies.qlearner.RiskyQLearner method), 106
- FirmButFair (class in axelrod.strategies.memoryone), 94
- foil_strategy_inspection() (axelrod.strategies.darwin.Darwin static method), 75
- foil_strategy_inspection() (axelrod.strategies.geller.Geller static method), 82
- foil_strategy_inspection() (axelrod.strategies.geller.GellerCooperator static method), 83
- foil_strategy_inspection() (axelrod.strategies.geller.GellerDefector static method), 83
- foil_strategy_inspection() (axelrod.strategies.mindreader.MindReader static method), 101
- foil_strategy_inspection() (axelrod.strategies.mindreader.MirrorMindReader static method), 101
- FoolMeForever (class in axelrod.strategies.oncebitten), 102
- FoolMeOnce (class in axelrod.strategies.oncebitten), 102
- ForgetfulFoolMeOnce (class in axelrod.strategies.oncebitten), 102
- ForgetfulGrudger (class in axelrod.strategies.grudger), 86
- Forgiver (class in axelrod.strategies.forgiver), 81
- ForgivingTitForTat (class in axelrod.strategies.forgiver), 81
- Fortress3 (class in axelrod.strategies.finite_state_machines), 78
- Fortress4 (class in axelrod.strategies.finite_state_machines), 79
- from_pattern() (axelrod.strategies.lookerup.LookupTable class method), 92
- FSMPlayer (class in axelrod.strategies.finite_state_machines), 78
- ## G
- Gambler (class in axelrod.strategies.gambler), 81
- Geller (class in axelrod.strategies.geller), 82
- GellerCooperator (class in axelrod.strategies.geller), 83
- GellerDefector (class in axelrod.strategies.geller), 83
- GeneralSoftGrudger (class in axelrod.strategies.grudger), 86
- genome (axelrod.strategies.darwin.Darwin attribute), 75
- get() (axelrod.strategies.lookerup.LookupTable method), 92
- get_last_n_plays() (in module axelrod.strategies.lookerup), 93
- get_new_itertools_cycle() (axelrod.strategies.cycler.Cycler method), 74
- get_siblings() (axelrod.strategies.dbs.DeterministicNode method), 76
- get_siblings() (axelrod.strategies.dbs.Node method), 76
- get_siblings() (axelrod.strategies.dbs.StochasticNode method), 77
- get_value() (axelrod.strategies.dbs.DeterministicNode method), 76
- Gladstein (class in axelrod.strategies.axelrod_second), 69
- GoByMajority (class in axelrod.strategies.gobymajority), 83
- GoByMajority10 (class in axelrod.strategies.gobymajority), 83
- GoByMajority20 (class in axelrod.strategies.gobymajority), 84
- GoByMajority40 (class in axelrod.strategies.gobymajority), 84
- GoByMajority5 (class in axelrod.strategies.gobymajority), 84
- Golden (class in axelrod.strategies.mathematicalconstants), 93
- Gradual (class in axelrod.strategies.titfortat), 112
- GradualKiller (class in axelrod.strategies.gradualkiller), 85
- Grofman (class in axelrod.strategies.axelrod_first), 66
- Grudger (class in axelrod.strategies.grudger), 86
- GrudgerAlternator (class in axelrod.strategies.grudger), 86
- Grumpy (class in axelrod.strategies.grumpy), 87
- GTFT (class in axelrod.strategies.memoryone), 95
- ## H
- Handshake (class in axelrod.strategies.handshake), 87
- HardGoByMajority (class in axelrod.strategies.gobymajority), 84

- HardGoByMajority10 (class in axelrod.strategies.gobymajority), 84
- HardGoByMajority20 (class in axelrod.strategies.gobymajority), 84
- HardGoByMajority40 (class in axelrod.strategies.gobymajority), 85
- HardGoByMajority5 (class in axelrod.strategies.gobymajority), 85
- HardProber (class in axelrod.strategies.prober), 103
- HardTitFor2Tats (class in axelrod.strategies.tiffortat), 112
- HardTitForTat (class in axelrod.strategies.tiffortat), 112
- HesitantQLearner (class in axelrod.strategies.qlearner), 106
- HMMPlayer (class in axelrod.strategies.hmm), 88
- Hopeless (class in axelrod.strategies.mutual), 101
- ## I
- Inverse (class in axelrod.strategies.inverse), 90
- InversePunisher (class in axelrod.strategies.punisher), 105
- is_alternator() (in module axelrod.strategies.hunter), 90
- is_stochastic() (axelrod.strategies.dbs.DeterministicNode method), 76
- is_stochastic() (axelrod.strategies.dbs.Node method), 77
- is_stochastic() (axelrod.strategies.dbs.StochasticNode method), 77
- is_stochastic() (axelrod.strategies.hmm.HMMPlayer method), 88
- is_stochastic_matrix() (in module axelrod.strategies.hmm), 88
- is_well_formed() (axelrod.strategies.hmm.SimpleHMM method), 88
- ## J
- Joss (class in axelrod.strategies.axelrod_first), 66
- ## K
- KnowledgeableWorseAndWorse (class in axelrod.strategies.worse_and_worse), 115
- ## L
- learning_rate (axelrod.strategies.qlearner.ArrogantQLearner attribute), 106
- learning_rate (axelrod.strategies.qlearner.CautiousQLearner attribute), 106
- learning_rate (axelrod.strategies.qlearner.HesitantQLearner attribute), 106
- learning_rate (axelrod.strategies.qlearner.RiskyQLearner attribute), 106
- LevelPunisher (class in axelrod.strategies.punisher), 105
- LimitedRetaliate (class in axelrod.strategies.retaliate), 107
- LimitedRetaliate2 (class in axelrod.strategies.retaliate), 108
- LimitedRetaliate3 (class in axelrod.strategies.retaliate), 108
- LookerUp (class in axelrod.strategies.lookerup), 90
- lookup_dict (axelrod.strategies.lookerup.LookerUp attribute), 91
- lookup_table_display() (axelrod.strategies.lookerup.LookerUp method), 91
- LookupTable (class in axelrod.strategies.lookerup), 91
- LRPlayer (class in axelrod.strategies.zero_determinant), 116
- ## M
- make_keys_into_plays() (in module axelrod.strategies.lookerup), 93
- MathConstantHunter (class in axelrod.strategies.hunter), 89
- MEM2 (class in axelrod.strategies.memorytwo), 94
- memory_length (axelrod.strategies.qlearner.RiskyQLearner attribute), 106
- MemoryOnePlayer (class in axelrod.strategies.memoryone), 95
- meta_strategy() (axelrod.strategies.meta.MetaHunter static method), 96
- meta_strategy() (axelrod.strategies.meta.MetaHunterAggressive static method), 96
- meta_strategy() (axelrod.strategies.meta.MetaMajority static method), 97
- meta_strategy() (axelrod.strategies.meta.MetaMinority static method), 97
- meta_strategy() (axelrod.strategies.meta.MetaMixer method), 97
- meta_strategy() (axelrod.strategies.meta.MetaPlayer method), 98
- meta_strategy() (axelrod.strategies.meta.MetaWinner method), 98
- meta_strategy() (axelrod.strategies.meta.MetaWinnerEnsemble method), 98
- meta_strategy() (axelrod.strategies.sequence_player.SequencePlayer method), 109
- meta_strategy() (axelrod.strategies.sequence_player.ThueMorseInverse method), 109
- MetaHunter (class in axelrod.strategies.meta), 96
- MetaHunterAggressive (class in axelrod.strategies.meta), 96
- MetaMajority (class in axelrod.strategies.meta), 96
- MetaMajorityFiniteMemory (class in axelrod.strategies.meta), 97
- MetaMajorityLongMemory (class in axelrod.strategies.meta), 97
- MetaMajorityMemoryOne (class in axelrod.strategies.meta), 97
- MetaMinority (class in axelrod.strategies.meta), 97
- MetaMixer (class in axelrod.strategies.meta), 97

MetaPlayer (class in axelrod.strategies.meta), 98
 MetaWinner (class in axelrod.strategies.meta), 98
 MetaWinnerDeterministic (class in axelrod.strategies.meta), 98
 MetaWinnerEnsemble (class in axelrod.strategies.meta), 98
 MetaWinnerFiniteMemory (class in axelrod.strategies.meta), 98
 MetaWinnerLongMemory (class in axelrod.strategies.meta), 98
 MetaWinnerMemoryOne (class in axelrod.strategies.meta), 99
 MetaWinnerStochastic (class in axelrod.strategies.meta), 99
 Michaelos (class in axelrod.strategies.tiffortat), 113
 MindBender (class in axelrod.strategies.mindcontrol), 100
 MindController (class in axelrod.strategies.mindcontrol), 100
 MindReader (class in axelrod.strategies.mindreader), 101
 MindWarper (class in axelrod.strategies.mindcontrol), 100
 minimax_tree_search() (in module axelrod.strategies.dbs), 77
 MirrorMindReader (class in axelrod.strategies.mindreader), 101
 MoreGrofman (class in axelrod.strategies.axelrod_second), 70
 move() (axelrod.strategies.finite_state_machines.SimpleFSM method), 79
 move() (axelrod.strategies.hmm.SimpleHMM method), 88
 move_gen() (in module axelrod.strategies.dbs), 77
 mutate() (axelrod.strategies.darwin.Darwin method), 75

N

NaiveProber (class in axelrod.strategies.prober), 103
 name (axelrod.strategies.adaptive.Adaptive attribute), 63
 name (axelrod.strategies.alternator.Alternator attribute), 63
 name (axelrod.strategies.ann.ANN attribute), 63
 name (axelrod.strategies.ann.EvolvedANN attribute), 63
 name (axelrod.strategies.ann.EvolvedANN5 attribute), 63
 name (axelrod.strategies.ann.EvolvedANNNoise05 attribute), 64
 name (axelrod.strategies.apavlov.APavlov2006 attribute), 64
 name (axelrod.strategies.apavlov.APavlov2011 attribute), 64
 name (axelrod.strategies.appeaser.Appeaser attribute), 65
 name (axelrod.strategies.averagecopier.AverageCopier attribute), 65
 name (axelrod.strategies.averagecopier.NiceAverageCopier attribute), 65
 name (axelrod.strategies.axelrod_first.Davis attribute), 65
 name (axelrod.strategies.axelrod_first.Feld attribute), 66
 name (axelrod.strategies.axelrod_first.Grofman attribute), 66
 name (axelrod.strategies.axelrod_first.Joss attribute), 66
 name (axelrod.strategies.axelrod_first.Nydegger attribute), 67
 name (axelrod.strategies.axelrod_first.RevisedDowning attribute), 67
 name (axelrod.strategies.axelrod_first.Shubik attribute), 67
 name (axelrod.strategies.axelrod_first.SteinAndRapoport attribute), 68
 name (axelrod.strategies.axelrod_first.TidemanAndChieruzzi attribute), 68
 name (axelrod.strategies.axelrod_first.Tullock attribute), 68
 name (axelrod.strategies.axelrod_first.UnnamedStrategy attribute), 69
 name (axelrod.strategies.axelrod_second.Champion attribute), 69
 name (axelrod.strategies.axelrod_second.Eatherley attribute), 69
 name (axelrod.strategies.axelrod_second.Gladstein attribute), 70
 name (axelrod.strategies.axelrod_second.MoreGrofman attribute), 70
 name (axelrod.strategies.axelrod_second.Tester attribute), 70
 name (axelrod.strategies.axelrod_second.Tranquilizer attribute), 72
 name (axelrod.strategies.backstabber.BackStabber attribute), 72
 name (axelrod.strategies.backstabber.DoubleCrosser attribute), 72
 name (axelrod.strategies.better_and_better.BetterAndBetter attribute), 73
 name (axelrod.strategies.calculator.Calculator attribute), 73
 name (axelrod.strategies.cooperator.Cooperator attribute), 73
 name (axelrod.strategies.cooperator.TrickyCooperator attribute), 73
 name (axelrod.strategies.cycler.AntiCycler attribute), 73
 name (axelrod.strategies.cycler.Cycler attribute), 74
 name (axelrod.strategies.cycler.CyclerCCCCCD attribute), 74
 name (axelrod.strategies.cycler.CyclerCCCD attribute), 74
 name (axelrod.strategies.cycler.CyclerCCDCD attribute), 74
 name (axelrod.strategies.cycler.CyclerCCD attribute), 74
 name (axelrod.strategies.cycler.CyclerDC attribute), 75
 name (axelrod.strategies.cycler.CyclerDDC attribute), 75

name (axelrod.strategies.darwin.Darwin attribute), 75
 name (axelrod.strategies.dbs.DBS attribute), 76
 name (axelrod.strategies.defector.Defector attribute), 77
 name (axelrod.strategies.defector.TrickyDefector attribute), 77
 name (axelrod.strategies.doubler.Doubler attribute), 78
 name (axelrod.strategies.finite_state_machines.EvolvedFSM1 attribute), 78
 name (axelrod.strategies.finite_state_machines.EvolvedFSM16Noise05 attribute), 78
 name (axelrod.strategies.finite_state_machines.EvolvedFSM4 attribute), 78
 name (axelrod.strategies.finite_state_machines.Fortress3 attribute), 79
 name (axelrod.strategies.finite_state_machines.Fortress4 attribute), 79
 name (axelrod.strategies.finite_state_machines.FSMPlayer attribute), 78
 name (axelrod.strategies.finite_state_machines.Predator attribute), 79
 name (axelrod.strategies.finite_state_machines.Pun1 attribute), 79
 name (axelrod.strategies.finite_state_machines.Raider attribute), 79
 name (axelrod.strategies.finite_state_machines.Ripoff attribute), 79
 name (axelrod.strategies.finite_state_machines.SolutionB1 attribute), 80
 name (axelrod.strategies.finite_state_machines.SolutionB5 attribute), 80
 name (axelrod.strategies.finite_state_machines.TF1 attribute), 80
 name (axelrod.strategies.finite_state_machines.TF2 attribute), 80
 name (axelrod.strategies.finite_state_machines.TF3 attribute), 80
 name (axelrod.strategies.finite_state_machines.Thumper attribute), 81
 name (axelrod.strategies.forgiver.Forgiver attribute), 81
 name (axelrod.strategies.forgiver.ForgivingTitForTat attribute), 81
 name (axelrod.strategies.gambler.Gambler attribute), 81
 name (axelrod.strategies.gambler.PSOGambler1_1_1 attribute), 81
 name (axelrod.strategies.gambler.PSOGambler2_2_2 attribute), 82
 name (axelrod.strategies.gambler.PSOGambler2_2_2_Noise05 attribute), 82
 name (axelrod.strategies.gambler.PSOGamblerMem1 attribute), 82
 name (axelrod.strategies.gambler.ZDMem2 attribute), 82
 name (axelrod.strategies.geller.Geller attribute), 82
 name (axelrod.strategies.geller.GellerCooperator attribute), 83
 name (axelrod.strategies.geller.GellerDefector attribute), 83
 name (axelrod.strategies.gobymajority.GoByMajority attribute), 83
 name (axelrod.strategies.gobymajority.GoByMajority10 attribute), 83
 name (axelrod.strategies.gobymajority.GoByMajority20 attribute), 84
 name (axelrod.strategies.gobymajority.GoByMajority40 attribute), 84
 name (axelrod.strategies.gobymajority.GoByMajority5 attribute), 84
 name (axelrod.strategies.gobymajority.HardGoByMajority attribute), 84
 name (axelrod.strategies.gobymajority.HardGoByMajority10 attribute), 84
 name (axelrod.strategies.gobymajority.HardGoByMajority20 attribute), 84
 name (axelrod.strategies.gobymajority.HardGoByMajority40 attribute), 85
 name (axelrod.strategies.gobymajority.HardGoByMajority5 attribute), 85
 name (axelrod.strategies.gradualkiller.GradualKiller attribute), 85
 name (axelrod.strategies.grudger.Aggravater attribute), 85
 name (axelrod.strategies.grudger.EasyGo attribute), 85
 name (axelrod.strategies.grudger.ForgetfulGrudger attribute), 86
 name (axelrod.strategies.grudger.GeneralSoftGrudger attribute), 86
 name (axelrod.strategies.grudger.Grudger attribute), 86
 name (axelrod.strategies.grudger.GrudgerAlternator attribute), 87
 name (axelrod.strategies.grudger.OppositeGrudger attribute), 87
 name (axelrod.strategies.grudger.SoftGrudger attribute), 87
 name (axelrod.strategies.grumpy.Grumpy attribute), 87
 name (axelrod.strategies.handshake.Handshake attribute), 87
 name (axelrod.strategies.hmm.EvolvedHMM5 attribute), 88
 name (axelrod.strategies.hmm.HMMPlayer attribute), 88
 name (axelrod.strategies.hunter.AlternatorHunter attribute), 88
 name (axelrod.strategies.hunter.CooperatorHunter attribute), 89
 name (axelrod.strategies.hunter.CycleHunter attribute), 89
 name (axelrod.strategies.hunter.DefectorHunter attribute), 89
 name (axelrod.strategies.hunter.EventualCycleHunter attribute), 89
 name (axelrod.strategies.hunter.MathConstantHunter attribute), 89

tribute), 89
name (axelrod.strategies.hunter.RandomHunter attribute), 90
name (axelrod.strategies.inverse.Inverse attribute), 90
name (axelrod.strategies.lookerup.EvolvedLookerUp1_1_1 attribute), 90
name (axelrod.strategies.lookerup.EvolvedLookerUp2_2_2 attribute), 90
name (axelrod.strategies.lookerup.LookerUp attribute), 91
name (axelrod.strategies.lookerup.Winner12 attribute), 93
name (axelrod.strategies.lookerup.Winner21 attribute), 93
name (axelrod.strategies.mathematicalconstants.e attribute), 94
name (axelrod.strategies.mathematicalconstants.Golden attribute), 94
name (axelrod.strategies.mathematicalconstants.Pi attribute), 94
name (axelrod.strategies.memoryone.ALLCorALLD attribute), 94
name (axelrod.strategies.memoryone.FirmButFair attribute), 95
name (axelrod.strategies.memoryone.GTFT attribute), 95
name (axelrod.strategies.memoryone.MemoryOnePlayer attribute), 95
name (axelrod.strategies.memoryone.ReactivePlayer attribute), 95
name (axelrod.strategies.memoryone.SoftJoss attribute), 95
name (axelrod.strategies.memoryone.StochasticCooperator attribute), 96
name (axelrod.strategies.memoryone.StochasticWSLS attribute), 96
name (axelrod.strategies.memoryone.WinShiftLoseStay attribute), 96
name (axelrod.strategies.memoryone.WinStayLoseShift attribute), 96
name (axelrod.strategies.memorytwo.MEM2 attribute), 94
name (axelrod.strategies.meta.MetaHunter attribute), 96
name (axelrod.strategies.meta.MetaHunterAggressive attribute), 96
name (axelrod.strategies.meta.MetaMajority attribute), 97
name (axelrod.strategies.meta.MetaMajorityFiniteMemory attribute), 97
name (axelrod.strategies.meta.MetaMajorityLongMemory attribute), 97
name (axelrod.strategies.meta.MetaMajorityMemoryOne attribute), 97
name (axelrod.strategies.meta.MetaMinority attribute), 97
name (axelrod.strategies.meta.MetaMixer attribute), 98
name (axelrod.strategies.meta.MetaPlayer attribute), 98
name (axelrod.strategies.meta.MetaWinner attribute), 98
name (axelrod.strategies.meta.MetaWinnerDeterministic attribute), 98
name (axelrod.strategies.meta.MetaWinnerEnsemble attribute), 98
name (axelrod.strategies.meta.MetaWinnerFiniteMemory attribute), 98
name (axelrod.strategies.meta.MetaWinnerLongMemory attribute), 98
name (axelrod.strategies.meta.MetaWinnerMemoryOne attribute), 99
name (axelrod.strategies.meta.MetaWinnerStochastic attribute), 99
name (axelrod.strategies.meta.NiceMetaWinner attribute), 100
name (axelrod.strategies.meta.NiceMetaWinnerEnsemble attribute), 100
name (axelrod.strategies.meta.NMWEDeterministic attribute), 99
name (axelrod.strategies.meta.NMWEFiniteMemory attribute), 99
name (axelrod.strategies.meta.NMWELongMemory attribute), 99
name (axelrod.strategies.meta.NMWEMemoryOne attribute), 99
name (axelrod.strategies.meta.NMWEStochastic attribute), 99
name (axelrod.strategies.mindcontrol.MindBender attribute), 100
name (axelrod.strategies.mindcontrol.MindController attribute), 100
name (axelrod.strategies.mindcontrol.MindWarper attribute), 100
name (axelrod.strategies.mindreader.MindReader attribute), 101
name (axelrod.strategies.mindreader.MirrorMindReader attribute), 101
name (axelrod.strategies.mindreader.ProtectedMindReader attribute), 101
name (axelrod.strategies.mutual.Desperate attribute), 101
name (axelrod.strategies.mutual.Hopeless attribute), 102
name (axelrod.strategies.mutual.Willing attribute), 102
name (axelrod.strategies.negation.Negation attribute), 102
name (axelrod.strategies.oncebitten.FoolMeForever attribute), 102
name (axelrod.strategies.oncebitten.FoolMeOnce attribute), 102
name (axelrod.strategies.oncebitten.ForgetfulFoolMeOnce attribute), 103
name (axelrod.strategies.oncebitten.OnceBitten attribute), 103
name (axelrod.strategies.prober.CollectiveStrategy attribute), 103
name (axelrod.strategies.prober.HardProber attribute),

- 103
- name (axelrod.strategies.prober.NaiveProber attribute), 103
- name (axelrod.strategies.prober.Prober attribute), 104
- name (axelrod.strategies.prober.Prober2 attribute), 104
- name (axelrod.strategies.prober.Prober3 attribute), 104
- name (axelrod.strategies.prober.Prober4 attribute), 104
- name (axelrod.strategies.prober.RemorsefulProber attribute), 105
- name (axelrod.strategies.punisher.InversePunisher attribute), 105
- name (axelrod.strategies.punisher.LevelPunisher attribute), 105
- name (axelrod.strategies.punisher.Punisher attribute), 105
- name (axelrod.strategies.qlearner.ArrogantQLearner attribute), 106
- name (axelrod.strategies.qlearner.CautiousQLearner attribute), 106
- name (axelrod.strategies.qlearner.HesitantQLearner attribute), 106
- name (axelrod.strategies.qlearner.RiskyQLearner attribute), 106
- name (axelrod.strategies.rand.Random attribute), 107
- name (axelrod.strategies.resurrection.DoubleResurrection attribute), 107
- name (axelrod.strategies.resurrection.Resurrection attribute), 107
- name (axelrod.strategies.retaliate.LimitedRetaliate attribute), 107
- name (axelrod.strategies.retaliate.LimitedRetaliate2 attribute), 108
- name (axelrod.strategies.retaliate.LimitedRetaliate3 attribute), 108
- name (axelrod.strategies.retaliate.Retaliate attribute), 108
- name (axelrod.strategies.retaliate.Retaliate2 attribute), 108
- name (axelrod.strategies.retaliate.Retaliate3 attribute), 108
- name (axelrod.strategies.selfsteem.SelfSteem attribute), 109
- name (axelrod.strategies.sequence_player.ThueMorse attribute), 109
- name (axelrod.strategies.sequence_player.ThueMorseInverse attribute), 109
- name (axelrod.strategies.shortmem.ShortMem attribute), 109
- name (axelrod.strategies.stalker.Stalker attribute), 110
- name (axelrod.strategies.titfortat.AdaptiveTitForTat attribute), 110
- name (axelrod.strategies.titfortat.Alexei attribute), 111
- name (axelrod.strategies.titfortat.AntiTitForTat attribute), 111
- name (axelrod.strategies.titfortat.Bully attribute), 111
- name (axelrod.strategies.titfortat.ContriteTitForTat attribute), 111
- name (axelrod.strategies.titfortat.DynamicTwoTitsForTat attribute), 112
- name (axelrod.strategies.titfortat.EugeneNier attribute), 112
- name (axelrod.strategies.titfortat.Gradual attribute), 112
- name (axelrod.strategies.titfortat.HardTitFor2Tats attribute), 112
- name (axelrod.strategies.titfortat.HardTitForTat attribute), 113
- name (axelrod.strategies.titfortat.Michaelos attribute), 113
- name (axelrod.strategies.titfortat.NTitsForMTats attribute), 113
- name (axelrod.strategies.titfortat.OmegaTFT attribute), 113
- name (axelrod.strategies.titfortat.SlowTitForTwoTats2 attribute), 113
- name (axelrod.strategies.titfortat.SneakyTitForTat attribute), 114
- name (axelrod.strategies.titfortat.SpitefulTitForTat attribute), 114
- name (axelrod.strategies.titfortat.SuspiciousTitForTat attribute), 114
- name (axelrod.strategies.titfortat.TitFor2Tats attribute), 114
- name (axelrod.strategies.titfortat.TitForTat attribute), 115
- name (axelrod.strategies.titfortat.TwoTitsForTat attribute), 115
- name (axelrod.strategies.verybad.VeryBad attribute), 115
- name (axelrod.strategies.worse_and_worse.KnowledgeableWorseAndWorse attribute), 115
- name (axelrod.strategies.worse_and_worse.WorseAndWorse attribute), 115
- name (axelrod.strategies.worse_and_worse.WorseAndWorse2 attribute), 116
- name (axelrod.strategies.worse_and_worse.WorseAndWorse3 attribute), 116
- name (axelrod.strategies.zero_determinant.LRPlayer attribute), 116
- name (axelrod.strategies.zero_determinant.ZDExtort2 attribute), 116
- name (axelrod.strategies.zero_determinant.ZDExtort2v2 attribute), 117
- name (axelrod.strategies.zero_determinant.ZDExtort3 attribute), 117
- name (axelrod.strategies.zero_determinant.ZDExtort4 attribute), 117
- name (axelrod.strategies.zero_determinant.ZDExtortion attribute), 117
- name (axelrod.strategies.zero_determinant.ZDGen2 attribute), 117
- name (axelrod.strategies.zero_determinant.ZDGTFT2 attribute), 117

name (axelrod.strategies.zero_determinant.ZDMischief attribute), 117

name (axelrod.strategies.zero_determinant.ZDSet2 attribute), 118

Negation (class in axelrod.strategies.negation), 102

NiceAverageCopier (class in axelrod.strategies.averagecopier), 65

NiceMetaWinner (class in axelrod.strategies.meta), 99

NiceMetaWinnerEnsemble (class in axelrod.strategies.meta), 100

NMWEDeterministic (class in axelrod.strategies.meta), 99

NMWEFiniteMemory (class in axelrod.strategies.meta), 99

NMWELongMemory (class in axelrod.strategies.meta), 99

NMWEMemoryOne (class in axelrod.strategies.meta), 99

NMWESTochastic (class in axelrod.strategies.meta), 99

Node (class in axelrod.strategies.dbs), 76

NTitsForMTats (class in axelrod.strategies.titfortat), 113

Nydegger (class in axelrod.strategies.axelrod_first), 66

O

OmegaTFT (class in axelrod.strategies.titfortat), 113

OnceBitten (class in axelrod.strategies.oncebitten), 103

op_depth (axelrod.strategies.lookerup.LookupTable attribute), 92

op_openings (axelrod.strategies.lookerup.Plays attribute), 92

op_openings_depth (axelrod.strategies.lookerup.LookupTable attribute), 92

op_plays (axelrod.strategies.lookerup.Plays attribute), 93

OppositeGrudger (class in axelrod.strategies.grudger), 87

original_class (axelrod.strategies.axelrod_first.SteinAndRappaport attribute), 68

original_class (axelrod.strategies.backstabber.BackStabber attribute), 72

original_class (axelrod.strategies.backstabber.DoubleCrosser attribute), 72

original_class (axelrod.strategies.gradualkiller.GradualKiller attribute), 85

original_class (axelrod.strategies.meta.NiceMetaWinner attribute), 100

original_class (axelrod.strategies.meta.NiceMetaWinnerEnsemble attribute), 100

original_class (axelrod.strategies.stalker.Stalker attribute), 110

original_class (axelrod.strategies.titfortat.Alexei attribute), 111

original_class (axelrod.strategies.titfortat.ContriteTitForTat attribute), 111

original_class (axelrod.strategies.titfortat.EugeneNier attribute), 112

original_class (axelrod.strategies.titfortat.Michaelos attribute), 113

P

perform_q_learning() (axelrod.strategies.qlearner.RiskyQLearner method), 106

Pi (class in axelrod.strategies.mathematicalconstants), 94

player_depth (axelrod.strategies.lookerup.LookupTable attribute), 92

Plays (class in axelrod.strategies.lookerup), 92

Predator (class in axelrod.strategies.finite_state_machines), 79

Prober (class in axelrod.strategies.prober), 104

Prober2 (class in axelrod.strategies.prober), 104

Prober3 (class in axelrod.strategies.prober), 104

Prober4 (class in axelrod.strategies.prober), 104

ProtectedMindReader (class in axelrod.strategies.mindreader), 101

PSOGambler1_1_1 (class in axelrod.strategies.gambler), 81

PSOGambler2_2_2 (class in axelrod.strategies.gambler), 81

PSOGambler2_2_2_Noise05 (class in axelrod.strategies.gambler), 82

PSOGamblerMem1 (class in axelrod.strategies.gambler), 82

Pun1 (class in axelrod.strategies.finite_state_machines), 79

Punisher (class in axelrod.strategies.punisher), 105

R

Raider (class in axelrod.strategies.finite_state_machines), 79

Random (class in axelrod.strategies.rand), 107

RandomHunter (class in axelrod.strategies.hunter), 89

ratio (axelrod.strategies.mathematicalconstants.e attribute), 94

ratio (axelrod.strategies.mathematicalconstants.Golden attribute), 94

ratio (axelrod.strategies.mathematicalconstants.Pi attribute), 94

ReactivePlayer (class in axelrod.strategies.memoryone), 95

receive_match_attributes() (axelrod.strategies.darwin.Darwin method), 75

receive_match_attributes() (axelrod.strategies.memoryone.GTFT method), 95

receive_match_attributes() (axelrod.strategies.qlearner.RiskyQLearner method), 106

receive_match_attributes() (axelrod.strategies.zero_determinant.LRPlayer

- method), 116
- receive_match_attributes() (axelrod.strategies.zero_determinant.ZDExtort2 method), 116
- receive_match_attributes() (axelrod.strategies.zero_determinant.ZDGTFT2 method), 117
- RemorsefulProber (class in axelrod.strategies.prober), 104
- reset() (axelrod.strategies.darwin.Darwin method), 75
- reset() (axelrod.strategies.oncebitten.ForgetfulFoolMeOnce method), 103
- reset_genome() (axelrod.strategies.darwin.Darwin static method), 75
- Resurrection (class in axelrod.strategies.resurrection), 107
- Retaliate (class in axelrod.strategies.retaliate), 108
- Retaliate2 (class in axelrod.strategies.retaliate), 108
- Retaliate3 (class in axelrod.strategies.retaliate), 108
- RevisedDowning (class in axelrod.strategies.axelrod_first), 67
- Ripoff (class in axelrod.strategies.finite_state_machines), 79
- RiskyQLearner (class in axelrod.strategies.qlearner), 106
- ## S
- score_history() (axelrod.strategies.axelrod_first.Nydegger static method), 67
- score_last_round() (axelrod.strategies.adaptive.Adaptive method), 63
- select_action() (axelrod.strategies.qlearner.RiskyQLearner method), 106
- self_plays (axelrod.strategies.lookerup.Plays attribute), 93
- SelfSteem (class in axelrod.strategies.selfsteem), 109
- SequencePlayer (class in axelrod.strategies.sequence_player), 108
- set_four_vector() (axelrod.strategies.memoryone.MemoryOnePlayer method), 95
- ShortMem (class in axelrod.strategies.shortmem), 109
- should_demote() (axelrod.strategies.dbs.DBS method), 76
- should_promote() (axelrod.strategies.dbs.DBS method), 76
- Shubik (class in axelrod.strategies.axelrod_first), 67
- SimpleFSM (class in axelrod.strategies.finite_state_machines), 79
- SimpleHMM (class in axelrod.strategies.hmm), 88
- SlowTitForTwoTats2 (class in axelrod.strategies.titfortat), 113
- SneakyTitForTat (class in axelrod.strategies.titfortat), 113
- SoftGrudger (class in axelrod.strategies.grudger), 87
- SoftJoss (class in axelrod.strategies.memoryone), 95
- SolutionB1 (class in axelrod.strategies.finite_state_machines), 80
- SolutionB5 (class in axelrod.strategies.finite_state_machines), 80
- SpitefulTitForTat (class in axelrod.strategies.titfortat), 114
- split_weights() (in module axelrod.strategies.ann), 64
- Stalker (class in axelrod.strategies.stalker), 110
- state (axelrod.strategies.finite_state_machines.SimpleFSM attribute), 80
- state_transitions (axelrod.strategies.finite_state_machines.SimpleFSM attribute), 80
- SteinAndRapoport (class in axelrod.strategies.axelrod_first), 67
- StochasticCooperator (class in axelrod.strategies.memoryone), 95
- StochasticNode (class in axelrod.strategies.dbs), 77
- StochasticWSLS (class in axelrod.strategies.memoryone), 96
- strategy() (axelrod.strategies.adaptive.Adaptive method), 63
- strategy() (axelrod.strategies.alternator.Alternator method), 63
- strategy() (axelrod.strategies.ann.ANN method), 63
- strategy() (axelrod.strategies.apavlov.APavlov2006 method), 64
- strategy() (axelrod.strategies.apavlov.APavlov2011 method), 64
- strategy() (axelrod.strategies appeaser.Appeaser method), 65
- strategy() (axelrod.strategies.averagecopier.AverageCopier method), 65
- strategy() (axelrod.strategies.averagecopier.NiceAverageCopier method), 65
- strategy() (axelrod.strategies.axelrod_first.Davis method), 65
- strategy() (axelrod.strategies.axelrod_first.Feld method), 66
- strategy() (axelrod.strategies.axelrod_first.Grofman method), 66
- strategy() (axelrod.strategies.axelrod_first.Nydegger method), 67
- strategy() (axelrod.strategies.axelrod_first.RevisedDowning method), 67
- strategy() (axelrod.strategies.axelrod_first.Shubik method), 67
- strategy() (axelrod.strategies.axelrod_first.SteinAndRapoport method), 68
- strategy() (axelrod.strategies.axelrod_first.TidemanAndChieruzzi method), 68
- strategy() (axelrod.strategies.axelrod_first.Tulloch method), 68
- strategy() (axelrod.strategies.axelrod_first.UnnamedStrategy static method), 69

- strategy() (axelrod.strategies.axelrod_second.Champion method), 69
- strategy() (axelrod.strategies.axelrod_second.Eatherley static method), 69
- strategy() (axelrod.strategies.axelrod_second.Gladstein method), 70
- strategy() (axelrod.strategies.axelrod_second.MoreGrofman method), 70
- strategy() (axelrod.strategies.axelrod_second.Tester method), 70
- strategy() (axelrod.strategies.axelrod_second.Tranquilizer method), 72
- strategy() (axelrod.strategies.backstabber.BackStabber method), 72
- strategy() (axelrod.strategies.backstabber.DoubleCrosser method), 72
- strategy() (axelrod.strategies.better_and_better.BetterAndBest method), 73
- strategy() (axelrod.strategies.calculator.Calculator method), 73
- strategy() (axelrod.strategies.cooperator.Cooperator static method), 73
- strategy() (axelrod.strategies.cooperator.TrickyCooperator method), 73
- strategy() (axelrod.strategies.cycler.AntiCycler method), 74
- strategy() (axelrod.strategies.cycler.Cycler method), 74
- strategy() (axelrod.strategies.darwin.Darwin method), 75
- strategy() (axelrod.strategies.dbs.DBS method), 76
- strategy() (axelrod.strategies.defector.Defector static method), 77
- strategy() (axelrod.strategies.defector.TrickyDefector method), 77
- strategy() (axelrod.strategies.doubler.Doubler method), 78
- strategy() (axelrod.strategies.finite_state_machines.FSMPlayer method), 78
- strategy() (axelrod.strategies.forgiver.Forgiver method), 81
- strategy() (axelrod.strategies.forgiver.ForgivingTitForTat method), 81
- strategy() (axelrod.strategies.gambler.Gambler method), 81
- strategy() (axelrod.strategies.geller.Geller method), 82
- strategy() (axelrod.strategies.gobymajority.GoByMajority method), 83
- strategy() (axelrod.strategies.gradualkiller.GradualKiller method), 85
- strategy() (axelrod.strategies.grudger.Aggravater static method), 85
- strategy() (axelrod.strategies.grudger.EasyGo static method), 85
- strategy() (axelrod.strategies.grudger.ForgetfulGrudger method), 86
- strategy() (axelrod.strategies.grudger.GeneralSoftGrudger method), 86
- strategy() (axelrod.strategies.grudger.Grudger static method), 86
- strategy() (axelrod.strategies.grudger.GrudgerAlternator method), 87
- strategy() (axelrod.strategies.grudger.OppositeGrudger static method), 87
- strategy() (axelrod.strategies.grudger.SoftGrudger method), 87
- strategy() (axelrod.strategies.grumpy.Grumpy method), 87
- strategy() (axelrod.strategies.handshake.Handshake method), 87
- strategy() (axelrod.strategies.hmm.HMMPlayer method), 88
- strategy() (axelrod.strategies.hunter.AlternatorHunter method), 89
- strategy() (axelrod.strategies.hunter.CooperatorHunter method), 89
- strategy() (axelrod.strategies.hunter.CycleHunter method), 89
- strategy() (axelrod.strategies.hunter.DefectorHunter method), 89
- strategy() (axelrod.strategies.hunter.EventualCycleHunter method), 89
- strategy() (axelrod.strategies.hunter.MathConstantHunter method), 89
- strategy() (axelrod.strategies.hunter.RandomHunter method), 90
- strategy() (axelrod.strategies.inverse.Inverse static method), 90
- strategy() (axelrod.strategies.lookerup.LookerUp method), 91
- strategy() (axelrod.strategies.mathematicalconstants.CotoDeRatio method), 93
- strategy() (axelrod.strategies.memoryone.ALLCorALLD method), 94
- strategy() (axelrod.strategies.memoryone.MemoryOnePlayer method), 95
- strategy() (axelrod.strategies.memorytwo.MEM2 method), 94
- strategy() (axelrod.strategies.meta.MetaPlayer method), 98
- strategy() (axelrod.strategies.meta.NiceMetaWinner method), 100
- strategy() (axelrod.strategies.meta.NiceMetaWinnerEnsemble method), 100
- strategy() (axelrod.strategies.mindcontrol.MindBender static method), 100
- strategy() (axelrod.strategies.mindcontrol.MindController static method), 100
- strategy() (axelrod.strategies.mindcontrol.MindWarper static method), 101

- strategy() (axelrod.strategies.mindreader.MindReader method), 101
- strategy() (axelrod.strategies.mindreader.MirrorMindReader method), 101
- strategy() (axelrod.strategies.mutual.Desperate method), 101
- strategy() (axelrod.strategies.mutual.Hopeless method), 102
- strategy() (axelrod.strategies.mutual.Willing method), 102
- strategy() (axelrod.strategies.negation.Negation method), 102
- strategy() (axelrod.strategies.oncebitten.FoolMeForever static method), 102
- strategy() (axelrod.strategies.oncebitten.FoolMeOnce static method), 102
- strategy() (axelrod.strategies.oncebitten.ForgetfulFoolMeOnce static method), 103
- strategy() (axelrod.strategies.oncebitten.OnceBitten method), 103
- strategy() (axelrod.strategies.prober.CollectiveStrategy method), 103
- strategy() (axelrod.strategies.prober.HardProber method), 103
- strategy() (axelrod.strategies.prober.NaiveProber method), 104
- strategy() (axelrod.strategies.prober.Prober method), 104
- strategy() (axelrod.strategies.prober.Prober2 method), 104
- strategy() (axelrod.strategies.prober.Prober3 method), 104
- strategy() (axelrod.strategies.prober.Prober4 method), 104
- strategy() (axelrod.strategies.prober.RemorsefulProber method), 105
- strategy() (axelrod.strategies.punisher.InversePunisher method), 105
- strategy() (axelrod.strategies.punisher.LevelPunisher method), 105
- strategy() (axelrod.strategies.punisher.Punisher method), 105
- strategy() (axelrod.strategies.qlearner.RiskyQLearner method), 107
- strategy() (axelrod.strategies.rand.Random method), 107
- strategy() (axelrod.strategies.resurrection.DoubleResurrection method), 107
- strategy() (axelrod.strategies.resurrection.Resurrection method), 107
- strategy() (axelrod.strategies.retaliate.LimitedRetaliate method), 107
- strategy() (axelrod.strategies.retaliate.Retaliate method), 108
- strategy() (axelrod.strategies.selfsteem.SelfSteem method), 110
- strategy() (axelrod.strategies.sequence_player.SequencePlayer method), 109
- strategy() (axelrod.strategies.shortmem.ShortMem static method), 109
- strategy() (axelrod.strategies.stalker.Stalker method), 110
- strategy() (axelrod.strategies.titfortat.AdaptiveTitForTat method), 110
- strategy() (axelrod.strategies.titfortat.Alexei method), 111
- strategy() (axelrod.strategies.titfortat.AntiTitForTat static method), 111
- strategy() (axelrod.strategies.titfortat.Bully static method), 111
- strategy() (axelrod.strategies.titfortat.ContriteTitForTat method), 111
- strategy() (axelrod.strategies.titfortat.DynamicTwoTitsForTat static method), 112
- strategy() (axelrod.strategies.titfortat.EugeneNier method), 112
- strategy() (axelrod.strategies.titfortat.Gradual method), 112
- strategy() (axelrod.strategies.titfortat.HardTitFor2Tats static method), 112
- strategy() (axelrod.strategies.titfortat.HardTitForTat static method), 113
- strategy() (axelrod.strategies.titfortat.Michaelos method), 113
- strategy() (axelrod.strategies.titfortat.NTitsForMTats method), 113
- strategy() (axelrod.strategies.titfortat.OmegaTFT method), 113
- strategy() (axelrod.strategies.titfortat.SlowTitForTwoTats2 method), 113
- strategy() (axelrod.strategies.titfortat.SneakyTitForTat method), 114
- strategy() (axelrod.strategies.titfortat.SpitefulTitForTat method), 114
- strategy() (axelrod.strategies.titfortat.SuspiciousTitForTat static method), 114
- strategy() (axelrod.strategies.titfortat.TitFor2Tats static method), 114
- strategy() (axelrod.strategies.titfortat.TitForTat method), 115
- strategy() (axelrod.strategies.titfortat.TwoTitsForTat static method), 115
- strategy() (axelrod.strategies.verybad.VeryBad static method), 115
- strategy() (axelrod.strategies.worse_and_worse.KnowledgeableWorseAndWorse method), 115
- strategy() (axelrod.strategies.worse_and_worse.WorseAndWorse method), 115
- strategy() (axelrod.strategies.worse_and_worse.WorseAndWorse2 method), 116
- strategy() (axelrod.strategies.worse_and_worse.WorseAndWorse3 method), 116

SuspiciousTitForTat (class in axelrod.strategies.titfortat), 114

T

table_depth (axelrod.strategies.lookerup.LookupTable attribute), 92

Tester (class in axelrod.strategies.axelrod_second), 70

TF1 (class in axelrod.strategies.finite_state_machines), 80

TF2 (class in axelrod.strategies.finite_state_machines), 80

TF3 (class in axelrod.strategies.finite_state_machines), 80

ThueMorse (class in axelrod.strategies.sequence_player), 109

ThueMorseInverse (class in axelrod.strategies.sequence_player), 109

Thumper (class in axelrod.strategies.finite_state_machines), 80

TidemanAndChieruzzi (class in axelrod.strategies.axelrod_first), 68

TitFor2Tats (class in axelrod.strategies.titfortat), 114

TitForTat (class in axelrod.strategies.titfortat), 114

Tranquilizer (class in axelrod.strategies.axelrod_second), 70

TrickyCooperator (class in axelrod.strategies.cooperator), 73

TrickyDefector (class in axelrod.strategies.defector), 77

Tullock (class in axelrod.strategies.axelrod_first), 68

TwoTitsForTat (class in axelrod.strategies.titfortat), 115

U

UnnamedStrategy (class in axelrod.strategies.axelrod_first), 68

update_history_by_cond() (axelrod.strategies.dbs.DBS method), 76

update_state() (axelrod.strategies.axelrod_second.Tranquilizer method), 72

V

valid_callers (axelrod.strategies.darwin.Darwin attribute), 75

VeryBad (class in axelrod.strategies.verybad), 115

W

Willing (class in axelrod.strategies.mutual), 102

Winner12 (class in axelrod.strategies.lookerup), 93

Winner21 (class in axelrod.strategies.lookerup), 93

WinShiftLoseStay (class in axelrod.strategies.memoryone), 96

WinStayLoseShift (class in axelrod.strategies.memoryone), 96

world (axelrod.strategies.titfortat.AdaptiveTitForTat attribute), 110

WorseAndWorse (class in axelrod.strategies.worse_and_worse), 115

WorseAndWorse2 (class in axelrod.strategies.worse_and_worse), 115

WorseAndWorse3 (class in axelrod.strategies.worse_and_worse), 116

Z

ZDExtort2 (class in axelrod.strategies.zero_determinant), 116

ZDExtort2v2 (class in axelrod.strategies.zero_determinant), 116

ZDExtort3 (class in axelrod.strategies.zero_determinant), 117

ZDExtort4 (class in axelrod.strategies.zero_determinant), 117

ZDExtortion (class in axelrod.strategies.zero_determinant), 117

ZDGen2 (class in axelrod.strategies.zero_determinant), 117

ZDGTFT2 (class in axelrod.strategies.zero_determinant), 117

ZDMem2 (class in axelrod.strategies.gambler), 82

ZDMischief (class in axelrod.strategies.zero_determinant), 117

ZDSet2 (class in axelrod.strategies.zero_determinant), 118