# 基于硬币机的博弈模拟模型

## 摘要

本研究通过建立一种基于计算机模拟的硬币机博弈模型，探究了不同博弈策略对群体进化的影响。模型集中考虑了策略选择、成本、收益，以及策略分布对群体淘汰和进化的作用。通过调整关键参数，如博弈成本和收益，我们发现合作策略对于群体的长期生存和繁荣至关重要。模拟结果揭示了合作共赢的价值，为理解现实世界中的策略互动提供了洞见。

**关键词：博弈论 进化对策 计算机模拟**

## 问题重述

在当前这个快速变化的时代背景下，技术进步和全球化进程不仅推动了经济和社会的发展，也带来了新的挑战和竞争。信息时代的到来，让信息成为了影响决策的关键因素，而在众多决策理论中，博弈论以其对策略决策的深刻洞察而独树一帜。因此，对博弈论进行建模研究，不仅是理解和解决经济、政治、社会等领域中策略互动问题的关键，也是推动创新和促进可持续发展的重要工具。

随着人工智能、大数据等技术的发展，我们面临着更加复杂的决策环境。在这样的背景下，博弈论的建模与研究变得更加重要。它可以帮助我们在具有竞争和合作特点的复杂系统中，理解和预测不同参与者的行为和策略。例如，通过建立博弈论模型，我们可以研究企业之间的竞争策略，政府如何制定政策以应对公共卫生危机，乃至如何设计机制来促进社会公共资源的合理分配。

因此，博弈论的建模不仅是一个理论上的挑战，也是对实际问题深入理解和有效解决的重要工具。它能够帮助决策者在充满不确定性的环境中做出更加合理和有效的决策，对于推动社会进步和改善人类生活具有重要的意义。在这个信息爆炸和全球一体化的时代，深入研究博弈论并将其应用于实际问题，已经成为了一个不可或缺的任务。

对于博弈的建模由于现实情况的复杂性与不可预见性被公认为“极其困难”，但是我们可以将问题进行抽象与泛化，建立一个只有成本、收益、策略、不同策略的人数的博弈模型进行进化模拟，每次博弈游戏完成之后，进行一次淘汰，从而得出较为一般且泛用的博弈最佳策略。

## 问题分析

该模型应当可以作为重要参考简单地应用到典型的实际情况中，比如商业竞争、国际关系甚至是日常交互中的策略选择问题。

以商业竞争为例，公司之间的合作与竞争可以看作是一种博弈。在这种情境下，“合作”可以被视为企业间的合作协议或者战略联盟，“欺骗”则是违反协议以追求自身短期利益的行为。不同的企业采取的策略相当于模型中的四种策略类型。例如，一家始终遵守合作协议、重视长期合作关系的公司可以被视为“接受者”，而另一家经常违背合同条款以追求自身即时利益的公司则更像是“欺骗者”。

进化淘汰机制在商业竞争中同样适用。市场环境和消费者偏好的变化将不断淘汰那些不能适应市场、不能满足消费者需求的企业。适应度在这里可以理解为企业的市场份额、盈利能力或者品牌价值等指标，这些指标高的企业能够在竞争中生存下来并且进一步发展。

在这一分析框架下，我们可以进一步探讨在不同市场环境下（例如垄断、寡头竞争、完全竞争市场），不同策略的企业如何演化。在垄断市场中，“欺骗者”可能因为缺乏竞争对手而获得短期利益，但在完全竞争市场中，“接受者”通过构建长期信誉可能更能吸引忠实客户，从而在长期竞争中占据优势。此外，随着市场环境的变化，企业可能需要调整策略以适应新的竞争格局，这与模型中“狡猾者”的策略变化相类似。

通过这种方式，我们不仅能够理解商业竞争中策略选择的复杂性，也能够洞察在不断变化的市场环境中，如何通过策略的调整和演化来维持和增强竞争优势，进而提出更具针对性的商业战略建议。

## **模型假设**

**参与者理性假设：**假设两位参与者都是理性的，意味着他们都会根据自己的信息和博弈对策最大化其个人利益。在此模型中表现为固定的博弈策略。

**策略限定假设：**模型中只考虑四种策略（接受者、欺骗者、复读者和狡猾者），忽略其他可能存在的策略。

**信息完全假设：**假设两位参与者都完全知道对方的策略和历史行为，没有信息的不对称。

**记忆一致假设：**在考虑历史行为时，假设复读者和狡猾者只根据最近一次的交互做出反应，并不考虑更久远的历史。

**博弈简单化假设：**假设博弈的环境是静态的，不随时间变化，每次博弈的结果只受当前决策影响。

**收益等价假设：**假设所有参与者对于收益的评价是一致的，即收益矩阵中的数值对所有参与者都具有相同的效用价值。

**进化稳定假设：**假设演化过程是足够慢的，使得每种策略都有足够的时间展现其效果，并且环境足够稳定以允许长期演化。

**淘汰规则简化假设：**假设淘汰机制仅基于适应度而非其他可能的社会动力学因素，如声誉、群体压力或外部变化等。

**单一性适应度假设：**假设一个策略的适应度仅由其在博弈中的累积收益决定，不考虑其他可能影响适应度的因素，如多样性、风险偏好等。

## 模型建立

在分析硬币游戏机的博弈情形和各种策略行为以及进化淘汰机制的基础上，本研究旨在建立一个数学模型，以模拟和预测不同策略在多轮博弈中的动态演化过程。模型的建立主要分为三个步骤：定义博弈情形、策略模拟与进化淘汰规则的形式化描述。

**4.1 对博弈情形的分析**

首先，将硬币游戏机的博弈情形抽象为一个二人博弈模型，其中参与者可以选择“合作”或“欺骗”。合作意味着双方都放入硬币，欺骗则意味着至少有一方选择不放入硬币。基于此，定义博弈的收益矩阵，包含四种情况：共赢收益、欺骗收益、受骗收益和皆输收益。



图 1 收益矩阵

**4.2 对常见不同博弈策略的分析**

接下来，形式化描述四种策略：接受者、欺骗者、复读者和狡猾者。每种策略根据其特点定义为一个决策规则，接受者总是选择合作，欺骗者总是选择欺骗，复读者依据对方上一次的选择做出响应，而狡猾者则在遭遇欺骗后改变其策略为持续欺骗。

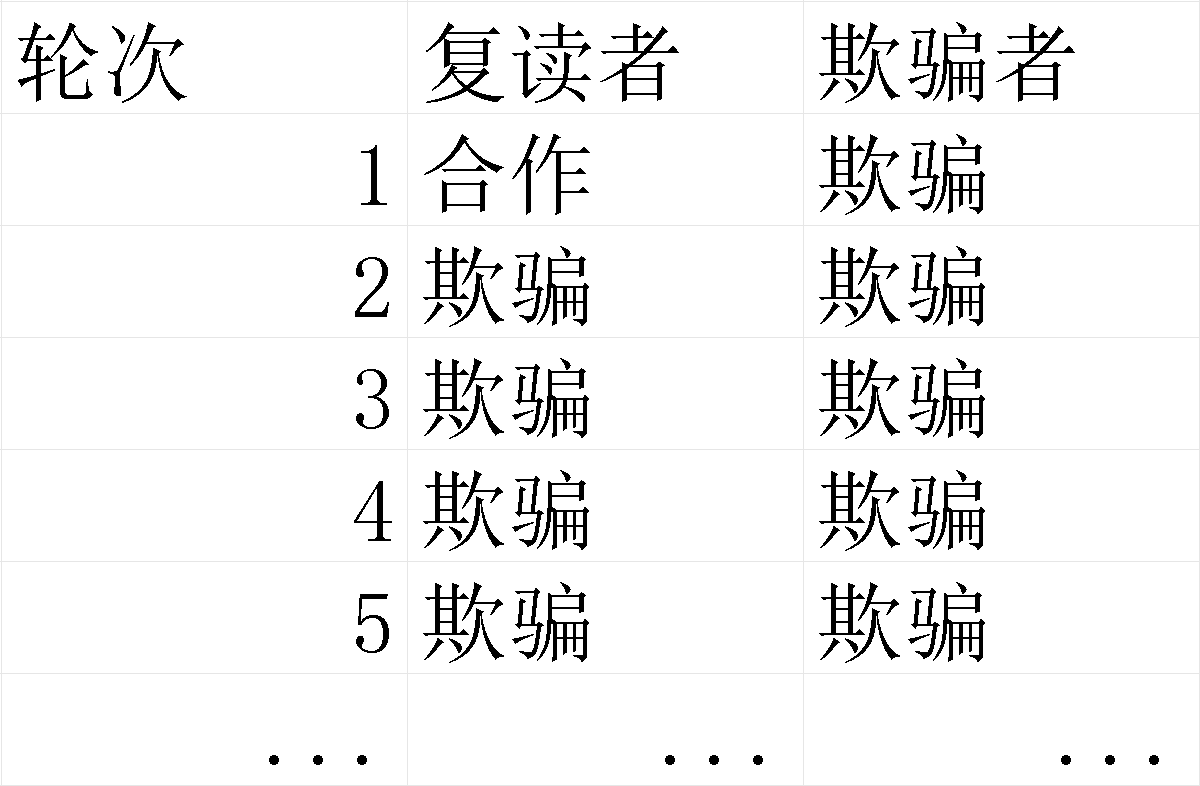


图 2 一次复读者与欺骗者的博弈结果

**4.3 对不同进化淘汰机制的分析**

最后，引入进化淘汰机制来模拟策略在多轮博弈中的演化。定义三种淘汰机制：保留前n者、淘汰后n者和淘汰所有最低分者。通过这三种淘汰机制决定哪些策略被保留或淘汰。

通过上述模型的建立，可以进行模拟实验来观察不同策略在各种环境和进化条件下的动态变化，以及哪些策略在长期演化中更有可能成为占优势的策略。这不仅为理解复杂博弈行为提供了一个有力的工具，也为研究博弈策略的演化提供了理论基础。

## 模型求解

在预设典型参数的情况下使用计算机技术进行模拟与淘汰进化。此下淘汰算法均为“淘汰所有最低分”，每一轮淘汰前两两各进行一场游戏，一场游戏中进行五轮博弈。在本轮无可淘汰（淘汰全部人）的情况下终止模拟。

**5.1 常规收益，多数合作**

设定游戏参数为：常规收益



图 3 常规收益

设定博弈环境为各五人。模拟结果如下：

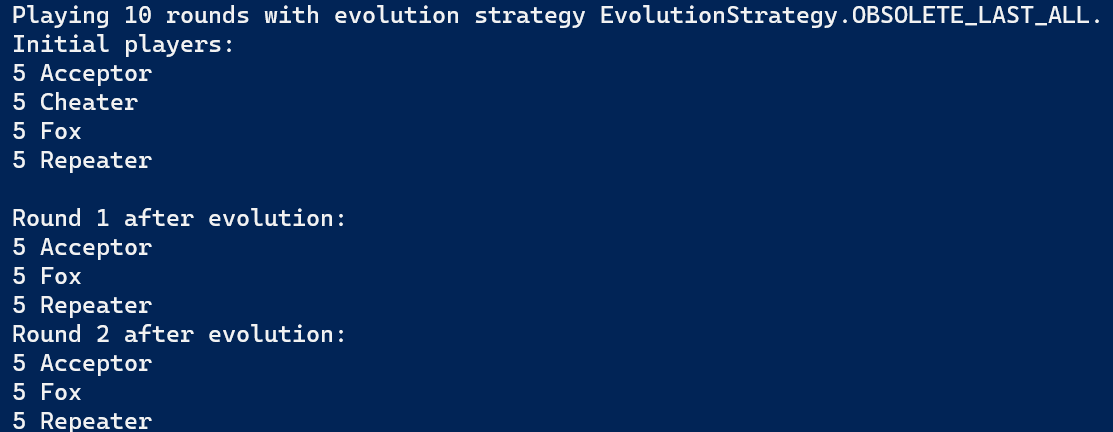


图 4 模拟结果

可见欺骗者被全部淘汰，剩下玩家中接受者占优势。

**5.2 常规收益，多数欺骗**

设定游戏参数为：常规收益

设定博弈环境为欺骗者十人，其他各五人。模拟结果如下：

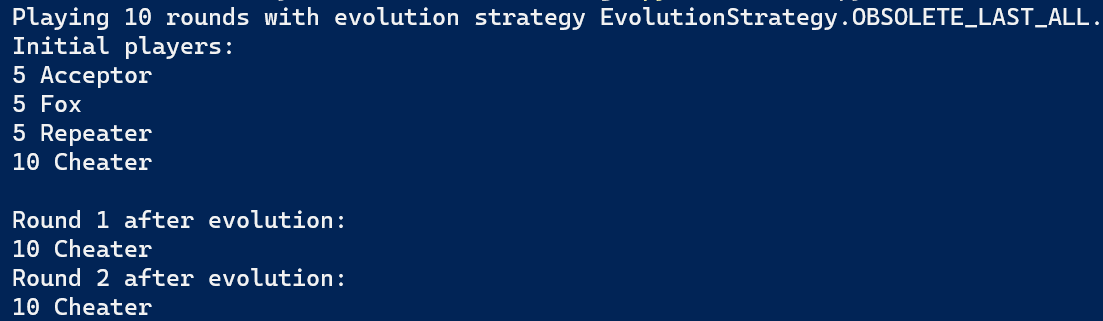


图 5 模拟结果

可见除了欺骗者其他玩家被全部淘汰。

**5.3 皆输惩罚，欺骗优势**

设定游戏参数为：皆输惩罚



图 6 皆输惩罚

设定博弈环境为欺骗者二十人，其他各五人。模拟结果如下：

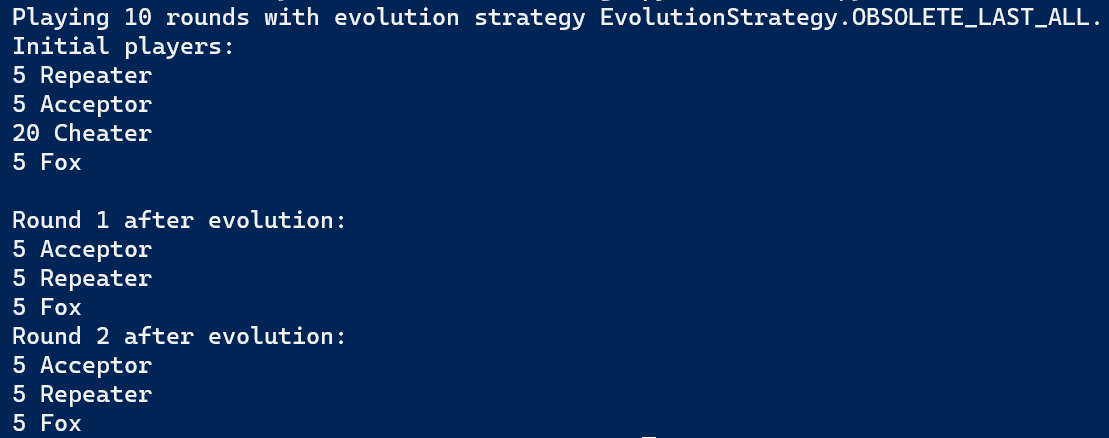


图 7 模拟结果

可见欺骗者被全部淘汰，剩下玩家中接受者占优势。将欺骗者人数不断上升至100，结果依然不变。

**5.4 双赢激励，欺骗优势**

设定游戏参数为：双赢激励



图 8 双赢激励

设定博弈环境为欺骗者二十人，其他各五人。模拟结果如下：

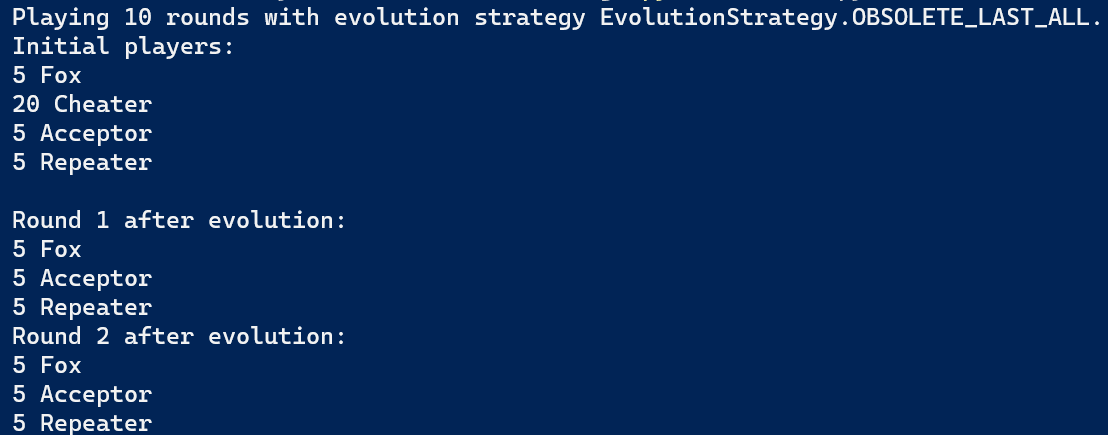


图 9 模拟结果

可见欺骗者被全部淘汰，剩下玩家中狡猾者占优势。将欺骗者人数不断上升至100，结果依然不变。

**5.5 总结陈述**

**策略有效性与收益结构：**在所有模拟实验中，策略的有效性与其产生的收益直接相关。在模拟实验1和2中，骗人的策略收益高于合作（双赢）策略，鼓励了欺骗者的行为。然而，在模拟实验3和4中，改变收益结构使得欺骗行为得不到奖励（模拟实验3中有皆输惩罚，模拟实验4中合作收益高于欺骗），导致欺骗行为不再处于优势地位。

**群体动态：**在所有模拟实验中，群体行为的演化展示了群体内不同策略间的竞争和淘汰。当一个策略在群体中占据大多数时（如模拟实验2中的欺骗者），即使这个策略不是最优的，它也可能因为群体压力而胜出。这可能是因为多数派策略的参与者在相互作用时能够保持稳定收益，而少数派则因为策略不匹配而遭受损失。

**淘汰机制的作用：**“淘汰所有最低分”这一机制加剧了策略间的竞争压力。在每轮博弈中，表现最差的策略被淘汰，这加速了某一策略在群体中的消失，尤其是当该策略的玩家数量较少时。

**5.6 生活实际应用**

在真实世界中，这些模拟实验可归纳为以下应用：

**激励设计：**这些模拟模拟实验说明了激励措施对个体行为的强烈影响。政策制定者和企业可以利用这一原理设计激励机制，鼓励合作、减少欺骗行为，通过调整收益结构改变人们的行为模式。

**社会规范与压力：**社会规范和群体压力对个体行为有显著影响。在一个鼓励诚实和合作的社会环境中，即使欺骗可能带来短期利益，长期来看合作和诚实更可能成为主导策略。

**宏观调控市场竞争：**在商业环境中，企业之间的竞争类似于这些模拟实验中的策略竞争。特别是在激烈的市场淘汰环境中，如果放任欺骗收益明显高于双赢收益，势必会迟滞乃至遏制发展。

**在社会科学中的应用：**这些模拟实验结果可以用来解释演化博弈论在人类社会、经济系统和生态系统中的应用，如何通过改变激励来引导个体或群体行为朝向社会期望的方向发展。

综上所述，这些模拟实验不仅有助于我们理解策略选择和群体动态的基本原理，还提供了如何通过调整激励结构影响行为模式的见解，这在社会政策、商业策略和个人决策中都有着广泛的应用价值。

## 模型评价

**模型优点：**

简洁性：模型通过抽象化简化了真实世界情景，将复杂的博弈行为归纳为四种基本策略，易于理解和分析。

策略多样性：模型包含多种策略，这允许观察和比较不同行为模式如何在不同条件下进行演化。

横向拓展性：模型的设计使得进化模拟与博弈对策实现高度解耦，向模型添加新的博弈对策与进化条件非常容易。

动态分析能力：通过引入进化淘汰机制，模型能够模拟策略随时间的动态变化，这在理解长期博弈动态方面是有价值的。

**缺点：**

过度简化：模型过度简化了复杂的人类行为和博弈情境，虽然可以得出高层次上的结论，但无法通过简单的横向拓展准确捕捉所有现实世界的动态。

局限的互动模式：只考虑了“合作”和“欺骗”两种行为模式，现实生活中的互动可能更加丰富和复杂。

非动态的收益矩阵：收益矩阵在同一次模拟中是固定的，现实世界中收益通常会因情景和环境变化而变化。

策略刚性：策略定义较为固定，真实世界中个体可能会更加灵活地调整其策略。

**改进建议：**

引入更多的策略和行为：考虑更多可能的策略和行为，如条件合作、报复等，以更好地模拟复杂的博弈情境。可以引入“信誉记录中心”与机器学习作为改进。

动态收益矩阵：允许收益矩阵随时间和策略的成功而变化，以更好地反映动态博弈情境。

混合策略：允许参与者采取混合策略，其中他们可以根据概率在不同策略间切换，而不是始终坚持单一策略。

环境因素的引入：模型中应考虑外部环境因素，如信息不对称、多方博弈等，这些都可能影响策略的选择和成功。

综上所述，该模型在简化和抽象博弈行为方面提供了一个有效的起点，但同时也需要进一步的细化和扩展，以便能够更加精确地模拟复杂的博弈情境和行为动态。

## 附录

项目链接：[Joxos/module\_building (github.com)](https://github.com/Joxos/module_building/tree/master)

main.py（参数为最后一次设置的情形）

from players import Repeater, Fox, Acceptor, Cheater, PlayerGroup

from headers import EvolutionStrategy, GameSettings

def generate\_players(repeater\_num, fox\_num, acceptor\_num, cheater\_num, game\_settings):

    return PlayerGroup(

        [Repeater()] \* repeater\_num

        + [Fox()] \* fox\_num

        + [Acceptor()] \* acceptor\_num

        + [Cheater()] \* cheater\_num,

        game\_settings,

    )

def main():

    common\_settings = GameSettings(

        cost=0,

        win\_win=5,

        one\_loss=0,

        one\_win=3,

        both\_loss=0,

        times=5,

        reset\_points=True,

    )

    players = generate\_players(

        repeater\_num=5,

        fox\_num=5,

        acceptor\_num=5,

        cheater\_num=20,

        game\_settings=common\_settings,

    )

    players.play\_and\_evolve(10, EvolutionStrategy.OBSOLETE\_LAST\_ALL)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

players.py

from game\_logging import logger

from headers import GameSettings, EvolutionStrategy

class Player:

def \_\_init\_\_(self):

self.points = 0

self.was\_cheated = False

def next\_choice(self):

return True

def update\_status(self, cheated):

pass

class Repeater(Player):

"""

A repeater is a man who always repeats other's last choice.

"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

def next\_choice(self):

if self.was\_cheated:

return False

else:

return True

def update\_status(self, cheated):

self.was\_cheated = cheated

class Cheater(Player):

"""

A cheater is a man who always cheats.

"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

def next\_choice(self):

return False

class Acceptor(Player):

"""

An acceptor is a man who never cheats.

"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

def next\_choice(self):

return True

class Fox(Player):

"""

A fox is a man who cheats if has been cheated before.

"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

def next\_choice(self):

if self.was\_cheated:

return False

else:

return True

def update\_status(self, was\_cheated):

if was\_cheated:

self.was\_cheated = True

def coin\_test(a: Player, b: Player, game\_settings):

for \_ in range(game\_settings.times):

a\_put = a.next\_choice()

b\_put = b.next\_choice()

logger.trace(

f"A is a {a.\_\_class\_\_.\_\_name\_\_} and B is a {b.\_\_class\_\_.\_\_name\_\_}."

)

logger.trace(f"{a\_put} {b\_put}.", end=" ")

a.points += game\_settings.cost

b.points += game\_settings.cost

if a\_put and b\_put:

logger.trace("Both won.")

a.points += game\_settings.win\_win

b.points += game\_settings.win\_win

elif not a\_put and not b\_put:

logger.trace("Both cheated.")

a.points += game\_settings.both\_loss

b.points += game\_settings.both\_loss

elif a\_put:

logger.trace("B won.")

b.points += game\_settings.one\_win

a.points += game\_settings.one\_loss

elif b\_put:

logger.trace("A won.")

a.points += game\_settings.one\_win

b.points += game\_settings.one\_loss

a.update\_status(not b\_put)

b.update\_status(not a\_put)

logger.trace(f"{a.points} {b.points}.\n")

class PlayerGroup:

def \_\_init\_\_(self, players, game\_settings):

self.players = players

self.game\_settings = game\_settings

def show\_players\_number(self):

types = set(p.\_\_class\_\_.\_\_name\_\_ for p in self.players)

player\_counts = {t: 0 for t in types}

for player in self.players:

player\_counts[player.\_\_class\_\_.\_\_name\_\_] += 1

for type in types:

logger.info(f"{player\_counts[type]} {type}")

def show\_points(self):

for player in self.players:

logger.info(f"{player.\_\_class\_\_.\_\_name\_\_} has {player.points} points.")

def play\_all(self):

for p1\_i in range(len(self.players)):

for p2\_i in range(p1\_i + 1, len(self.players)):

coin\_test(self.players[p1\_i], self.players[p2\_i], self.game\_settings)

# reset information about cheating

for player in self.players:

player.was\_cheated = False

self.sort()

def play\_and\_evolve(self, times, strategy, num=0):

logger.info(f"Playing {times} rounds with evolution strategy {strategy}.")

logger.info(f"Initial players:")

self.show\_players\_number()

print()

for time in range(times):

logger.info(f"Round {time+1} after evolution:")

self.play\_all()

go\_on = self.evolve(strategy, num)

self.show\_players\_number()

if self.game\_settings.reset\_points:

for player in self.players:

player.points = 0

if not go\_on:

break

def evolve(self, strategy, num=0):

"""Eliminate last players."""

# if len(self.players) <= num:

# return False

if strategy == EvolutionStrategy.KEEP\_BEST:

return self.keep\_best\_evolve(num)

elif strategy == EvolutionStrategy.OBSOLETE\_LAST:

return self.obsolete\_last\_evolve(num)

elif strategy == EvolutionStrategy.OBSOLETE\_LAST\_ALL:

return self.obsolete\_last\_all\_evolve()

else:

raise ValueError(f"Unknown evolution strategy: {strategy}")

def obsolete\_last\_all\_evolve(self):

last\_point = self.players[-1].points

highest\_point = self.players[0].points

# self.show\_points()

if last\_point == highest\_point:

return False

for player\_i in range(len(self.players)):

if self.players[player\_i].points == last\_point:

self.players = self.players[:player\_i]

break

return True

def keep\_best\_evolve(self, keep\_num):

"""Keep best players and eliminate last players."""

self.players = self.players[:keep\_num]

return True

def obsolete\_last\_evolve(self, obsolete\_num):

"""Obsolete last players and eliminate last players."""

self.players = self.players[:-obsolete\_num]

return True

def sort(self):

"""Sort by points in descending order."""

self.players.sort(key=lambda p: p.points, reverse=True)

def add(self, player):

self.players.append(player)

def remove(self, player):

self.players.remove(player)

def \_\_len\_\_(self):

return len(self.players)

def \_\_getitem\_\_(self, index):

return self.players[index]

def \_\_setitem\_\_(self, index, value):

self.players[index] = value

def \_\_delitem\_\_(self, index):

del self.players[index]

def \_\_iter\_\_(self):

return iter(self.players)

def \_\_reversed\_\_(self):

return reversed(self.players)

def \_\_contains\_\_(self, item):

return item in self.players

def \_\_str\_\_(self):

return str(self.players)

def \_\_repr\_\_(self):

return f"PlayerGroup({self.players})"

headers.py

from enum import Enum, auto

# game settings

class GameSettings:

def \_\_init\_\_(

self, cost, win\_win, one\_win, one\_loss, both\_loss, times, reset\_points

):

self.cost = cost

self.win\_win = win\_win

self.one\_win = one\_win

self.one\_loss = one\_loss

self.both\_loss = both\_loss

self.times = times

self.reset\_points = reset\_points

# evolutioin settings

class EvolutionStrategy(Enum):

KEEP\_BEST = auto()

OBSOLETE\_LAST = auto()

OBSOLETE\_LAST\_ALL = auto()

game\_logging.py

from loguru import logger

from sys import stderr

logger.remove()

logger.add(

stderr,

colorize=True,

format="<level>{message}</level>",

level="INFO",

)

requirements.txt

loguru