MPRA
Munich Personal RePEc Archive

Social Consequences of Commitment

Isaac, Alan G

11 October 2006

Online at https://mpra.ub.uni-muenchen.de/414/
MPRA Paper No. 414, posted 12 Oct 2006 UTC

Social Consequences of Commitment

An Agent-Based Approach

Author: Alan G. Isaac
Organization: Department of Economics, American University
Contact: aisaac@american.edu
Date: 2006-10-11
keywords: ACE, agent-based, computational economics, iterated prisoner's dilemma, evolutionary prisoner's dilemma, commitment
JEL: C63, C73

Abstract

This paper begins with a detailed computational introduction to a classic ACE model: an evolutionary prisoner's dilemma. The paper presents a simple but fully coded object oriented implementation of this model. (We use the Python programming language, which is shown to be a natural ally for ACE research). Using these tools, we demonstrate that player type evolution is affected by cardinal payoffs. We then explore a possible social benefit to commitment, where 'commitment' denotes an unwillingness to surrender a reciprocal strategy.

Contents

1 Introduction 2

2 Object Oriented Programming 3

3 A Simple Game 7

4 The Prisoner's Dilemma 10

5 CD-I Game 11

6 Evolutionary Soup 14

7 Evolutionary Grid 16

8 Commitment 18

9 Conclusion 20

10 Appendix A 20

11 References 21

1 Introduction

Computer simulation of economic interactions has been gaining adherents for more than two decades. Researchers often promote simulation methods as a "third way" of doing social science, distinct from both pure theory and from statistical exploration [axelrod1997abm]. While many methodological questions remain areas of active inquiry, especially in the realms of model robustness and of model/data confrontations, economists have been attracted by the ease with which computational methods can shed light on intractable theoretical puzzles. Some economists have gone so far as to argue that pure theory has little left to offer economists; that for economics to progress, we need simulations rather than theorems [hahn1991ej]. One specific hope is that simulations will advance economics by enabling the creation more realistic economic models---models that incorporate important historical and psychological properties that "pure theory" has trouble accommodating, including learning, adaptation, and other interesting limits on the computational resources of economic actors. Another specific hope is that unexpected but useful (for prediction or understanding) aggregate outcomes will "emerge" from the interactions of autonomous actors. There are many examples of these hopes being met in surprising and useful ways [bonabeau2002pnas].

Economic simulations are considered to be "agent-based" when the outcomes of interest result from the repeated interaction of autonomous actors, called "agents". These agents are autonomous in the sense that each can select actions from its own feasible set based on its own state and behavioral criteria. Agents are often stylized representations of real-world actors: either individuals, such as consumers or entrepreneurs, or aggregates of individuals, such as a corporation or a monetary authority (which may in turn be explicitly constituted of agents).

Substantial research in agent-based computational economics (ACE) has been conducted in a variety of general purpose programming languages, including procedural languages such as Fortran, C or Pascal; object oriented languages such as Objective-C, Smalltalk, C++, or Java; very-high level matrix languages such as MATLAB or GAUSS; and even symbolic algebra languages such as Maple or Mathematica. There are also a variety of special purpose toolkits for ACE, including SWARM and MAML (built on top of Objective-C), Ascape and NetLogo and Repast (built on Java), and many others [gilbert_bankes2002pnas]. In addition, there are a few efforts at general environments in which a variety of ACE experiments can be run. (Bremer's GLOBUS model, Hughes's IFs model, and Tesfatsion's Trade Network Game model are salient examples.) Researchers occasionally express the hope that one of these might emerge as a lingua franca for ACE research [luna_stefansson2000swarm]. Working against this, as others note, are the interfaces of capable toolkits and environments, which are often as difficult to master as a full-blown programming language [gilbert_bankes2002pnas]. A variety of languages and toolkits remain in use, and this seems likely to persist.

In the face of this plethora of alternatives, an additional ACE research language may seem otiose. However this paper shows that Python offers a natural environment for ACE research that should be seriously considered for ACE research and teaching. Python is a very-high-level general-purpose programming language that is well suited to ACE. This paper illustrates that suitability by quickly and simply developing a standard ACE application: the evolutionary iterated prisoner's dilemma. The example is illustrative of the needs in ACE research, and it serves to highlight many of the advantages of Python for such research. We then show how to extend this example to explore the importance of cardinal payoffs and a possible social implication of commitment.

The iterated prisoner's dilemma (IPD) is familiar to most social scientists, and many ACE researchers have implemented some version of the IPD as an exercise or illustration. Readers with ACE backgrounds will especially appreciate how simple, readable, and intuitive our Python implementation proves when compared to similar projects implemented in other languages. This indeed is one of the key reasons to use Python for ACE: with a little care from the programmer and an extremely modest understanding of Python by the reader, Python code is often as readable as pseudocode. This is no accident: readability is an explicit design goal of the Python language.

This paper has three related goals. First, we offer an introduction to object oriented modeling in agent-based computational economics (ACE), presenting a simple yet usable treatment of a classic ACE model. Very simple but already useful Python code is presented in the context of an evolutionary prisoner's dilemma, illustrating our claim that ACE research finds a natural ally in the Python programming language. Second, it shows how the tools developed in this paper can be used to demonstrate the importance of cardinal payoffs in the evolutionary prisoner's dilemma. Finally, we use the same tools to explore a possible social benefit to commitment, where 'commitment' denotes an unwillingness to surrender a reciprocal strategy.

2 Object Oriented Programming

We will say an 'object' is a collection of characteristics (data) and behavior (methods). We will call a computer program 'object-based' to the extent that data and methods tend to be bundled together into useful objects. Many researchers consider models of interacting agents to be naturally object-based. For example, a player in a game is naturally conceived in terms of that individual player's characteristics and behavior. We may therefore bundle these together into a player object. In this paper, we will consider code to be 'object-oriented' to the extent that it relies on 'encapsulation'.[1] Encapsulation means that an object's interactions with the rest of the program is mediated by an interface that does not expose the object's implementation details. For example, a player object may be asked for its next move by communicating with its `move` method, without worrying about how that method is implemented, which of its own data the player relies on to produce a move, or whether the player delegates moving to another object.

Within this highly simplified taxonomy, we can say object-oriented programming (OOP) is possible in most computer languages. It is a matter of approach and practices: OOP programming focuses on defining encapsulating objects and determining how such objects will interact. Yet it is also the case that some languages facilitate OOP. We say a programming language is object-oriented to the extent that it facilitates object-oriented programming. Key facilities of any OOP language is inheritance and polymorphism. In the roughest sense, inheritance is just one way to implement the old practice of reusing existing code: one (sub) class of objects can inherit some of its characteristics (data) and behavior (methods) from another (super) class. Inheritance proves to be an extremely powerful way to reuse existing code, and it often facilitates writing simpler and easier to understand code. Part of that simplicity arises through polymorphism, where a class determines that its subclasses respond to certain messages, but each subclasses may respond differently. For example, all player classes may implement a `move` method, but players designed to interact on a grid may move differently than players designed for random encounters. (See the examples below.)

For ACE, the phenomenologically salient actors of economic theory will be natural objects. This adds intuitive appeal to the model: the actors in the artificial economy are autonomous units whose interactions produce the model outcomes, analogous to the real-world outcomes produced by real-world actors. This aspect of OOP may facilitate production of sensible models, and almost certainly helps with communication---two big advantages in a relatively new area of economic research. However, object

[1] A detailed discussion of object oriented programming is beyond the scope of this paper. See for example [kak2003oop]. Note that we have not mentioned inheritance, which is often considered a defining characteristic of OOP.

orientation cuts much deeper than this, and many important objects in our models will not have obvious real world counterparts. (For example, we may have objects representing player characteristics, or objects representing aggregates of economic actors, or objects storing a summary "history" of our economic model.)

Why Python?

Economists naturally care about how the choice of a programming environment affects productivity: how quickly and painlessly can we implement a given idea in a given environment? The choice of programming language will often be very personal, responding to individual modes of thinking and to existing human capital. However many ACE researchers have claimed that OOP languages facilitate ACE modeling. And as we will see, one of the great appeals of Python is its ease of use and natural syntax, which allows rapid learning and easy prototyping. Among object-oriented languages, Python has some specific advantages for agent-based modeling. No one advantage makes Python unique, but together they add up to an attractive combination of power and ease of use.

Interpreted Language It is widely acknowledged that interpreted languages can facilitate prototyping and rapid development. They can also facilitate learning (due to the immediate feedback from the interpreter). Python interpreters exist for every major platform. This means that Python code is largely platform independent, which in turn facilitates code-sharing and collaboration with other ACE researchers.

Dynamic Typing Like many interpreted languages, Python is dynamically typed: variables (names) do not have types; rather the values that are assigned to them do. Here we emphasize two positive aspects of dynamic typing: code becomes simpler to write and simpler to read, and prototyping and refactoring are facilitated. (Many statically typed languages lack type inference, which leads to programs that appear cluttered with type declarations and explicit casts.)

Duck Typing Python encourages programming to objects' methods rather than to their types. This in turn encourages a strong polymorphism known as duck typing: functions and classes interact objects based on their attributes (rather than their class). Again, rather than engaging the varied controversies over duck typing, we simply note a widely acknowledged advantage: it facilitates quick prototyping and refactoring.

Garbage Collection Many languages require the programmer to explicitly allocate and deallocate memory for any objects created. Memory leaks are a common problem in such languages, even for experienced programmers. Other languages, Python included, offer automatic memory management: garbage collection ensure that memory is freed up when the object stored there is no longer needed. This can involve some sacrifice of speed and memory usage, but the gain in convenience is considerable.

Powerful Standard Library Python comes with a powerful, and extremely well documented standard library. Two component of special interest to ACE researchers will be the utilities for platform independent interaction with the operating system and the extensive random-number services. (Random number generation uses the Mersenne Twister by default.)[2]

Flexible Data Structures Python provides a collection of powerful and flexible data structures. Here we mention sequence data types and dictionaries.[3] Note that, as a result of Python's strong object orientation, these types are easily subclassed and extended when needed.

Sequence data types are indexed collections of items, which may be heterogeneous. Lists and tuples are sequence types that are ubiquitous in Python programming. Lists are very flexible: they can grow or shrink, and can hold heterogeneous elements. An empty list can be created as `[]` or as `list()`. Elements can then be added with the append method. A populated list can also be created directly by listing its elements between brackets, for example `[1,'a']` is a two element list (with heterogeneous elements). A list of lists can be a natural representation of a two-dimensional array, such as a grid or a matrix.[4]

Sequences can be used for loop control: Python's `for` statement iterates over the items of a sequence. For example, if `x` is a list or tuple, the following code snippet prints a representation of each item in `x`:

[2] Details can be found in [matsumoto_kurita-1998-acmt], which contains a much cited algorithm for the Mersenne Twister.

```
for item in x:
  print item
```

Like many programming languages, Python uses zero-based indexing: if x is a sequence, then x[0] returns the first element of x.[5] To illustrate zero-based indexing in more detail, consider the statement `payoffmat = [ [(3,3),(0,5)] , [(5,0),(1,1)] ]` (which occurs in our code examples below). This is a list of two lists of two tuples each. Since we index sequence elements by postfixing an integer in brackets, `payoffmat[0]` is `[(3,3),(0,5)]`. Similarly, `payoffmat[0][1]` is `(0,5)`. (This tuple represents two player payoffs given moves 0 and 1.)

Lists are "mutable": you can replace list items by assignment. For example, if x is a list, then `x[0]=99` will set the first element of x to the integer 99. Tuples are "immutable": you cannot replace list items by assignment. For example, if x is a tuple, then `x[0]=99` will raise an error.

The dictionary is the basic Python mapping data type (resembling hash tables or associative arrays in other languages). A dictionary is a collection of key-value pairs, where the value can be retrieved by using the key as an index. If `x=dict(a=0,b=1)` then `x['a']` returns the value 0. The same dictionary can be created from a list of 2-tuples: `x=dict( [('a',0), ('b',1)] )`, which will often prove useful.

Useful "Built-In" Functions A few dozen very useful functions are built-in to Python, in the sense that they are always available. The user does not need to import any standard library modules to access these functions. We will mention three at this point: `range`, `enumerate`, and `zip`.

If n is a nonnegative integer, then `range(n)` creates a list of the first n nonnegative integers. It is idiomatic in Python to use `range` for loop control: e.g., `for i in range(10):  print i`.

If x is a sequence, then `enumerate(x)` pairs each element with its index. For example, if `x=[9,8,7]` then `enumerate(x)` will generate the tuples (0,9), (1,8), and (2,7). (Recall from above that Python uses zero-based indexing.)

If x is a sequence and y is a sequence, then `zip(x,y)` will produce a list of 2-tuples, pairing each element of x with the corresponding element of y. For example, If `x=(9,8,7)` and `y=(6,5,4)` then `zip(x,y)` creates the list `[(9,6), (8,5), (7,4)]`.

Let us call that last list z. If we now evaluate `zip(*z)` we create the list `[(9,8,7),(6,5,4)]`, which contains our two original tuples. The asterisk instructs Python to "unpack" z so as to use each element of z as an argument of `zip`. Note that this means that `zip` can be used to transpose a list of equal length lists.

We will often use `zip` to construct dictionaries in a particularly compact yet readable way. Suppose `players` is a list of players and `moves` is a corresponding list of moves. Then `zip(players,moves)` produces a list of 2-tuples by "zipping" together each player and its move. A dictionary can be initialized with such a list of 2-tuples. So `d=dict( zip(players,moves) )` produces a mapping from players to moves. We will use this compact and readable idiom repeatedly.[6]

Generator Expressions Python has evolved some very powerful and easy to read syntax for the creation of data structures. Rather than traditional looping constructs, "generator expressions" are often preferred for producing lists and tuples. Generator expressions are perhaps best introduced by example.

Suppose `players` is a tuple of players, each of whom has a playertype attribute, and we want to produce a corresponding tuple of playertypes. Here is a (fairly) traditional approach to this problem:

[3] For a detailed and generally excellent introduction, see the Python Tutorial.

[4] However, efficient computations with multi-dimensional arrays will generally use the numpy package.

[5] Zero-based indexing can initially feel odd to those with no programming experience (even though we use it all the time when referring, e.g., to the 21st century). In practice, it proves extremely convenient.

[6] There is a speed sacrifice here, most importantly because we create unneeded 2-tuples but also because `zip` creates an unneeded list. The latter is easily avoided by using `izip` from the `itertools` module. Avoiding the former requires an explicit loop. In this paper we will generally value compact readability over speed optimizations, and we will not introduce the `itertools` module.

```
ptypes = list()
for player in players:
  ptypes.append(player.playertype)
ptypes = tuple(ptypes)
```

Here we create an empty list, to which we sequentially append the playertype type of each player, and we finally create a tuple corresponding to the list of playertypes. (Note the attribute accesses, which use the "dot notation" common in object oriented languages.) We can accomplish the same thing more elegantly and efficiently with a generator expression:

```
ptypes = tuple(player.playertype for player in players)
```

This iterates through the generator `(player.playertype for player in playerlist)` to populate a tuple of player types, one playertype for each player.[7] The expression on the right reads very naturally: create a tuple that contains the player's playertype for each player in `players`. This is a beautifully readable and compact way to generate this tuple's elements.

Graphics Excellent and powerful graphics libraries are available. For example, users needing interactive two-dimensional graphs might use the Matplotlib package, which allows both ease of use and, for those who need it, a remarkably fine-grained control of graphical presentations.

Numerics The numpy package implements powerful multidimensional arrays along with the basic functionality required for scientific computing. The numpy-based SciPy package offers substantial additional functionality, resulting in a free and open source scientific computing environment competitive with many commercial offerings.

Extensibility As with other interpreted languages, speed concerns arise when Python programs make heavy use of looping constructs. The high speed of modern computers moderate this concern but for models with very large numbers of agents does not eliminate it. Fortunately it is easy to write Python performance enhancing extensions in relatively low-level programming languages such as C or Fortran.[8]

String Formatting Python string interpolation can be done in standard C language (`printf`) fashion, but interpolation values can also be provided as a dictionary (i.e., a hash table). This makes for convenient report generation (e.g., when objects are asked about their current state).

Documentation Online documentation is excellent and free. In addition Python encourages the creation of self-documenting objects, and most built-in functions provide excellent self-documentation.

Readability For write-once read-never code, readability is perhaps not an issue. For code that is to be shared with a research community and reread after the passage of time, readability is a crucial consideration. Python code is often as readable as pseudocode, as we shall see. Readability is aided by Python's syntax, especially the use of whitespace to delimit code blocks.[9] It is also aided by the Python "culture", which emphasizes code readability and disparages clever programming "tricks".

Free and Open Source It is easy to exaggerate the importance of the financial cost of software, which is usually a small relative the the cost of doing research. More important is the freedom to modify: the licenses of most Python packages allow users to change them as needed. This is even true of the excellent graphics and numerical packages. The open source nature of the software makes this freedom relevant: users often contribute enhancements to important packages. Finally, some researchers plausibly claim that the use of closed source software conflicts with the goal of verifiability and replicability in science.

[7] An attentive reader might wonder why the generator example has only a single pair of parentheses. When a generator expression when it is the only argument of a function, Python allows us to omit its parentheses.

[8] See for example the ctypes module, f2py, Pyrex, SWIG, and weave.

[9] Programmers accustomed to braces-delimited blocks are sometimes initially put off by whitespace-delimited blocks, but for newcomers to programming it proves intuitive and natural.

Other important features of Python---exception handling, support for unit testing, support for multiple inheritance, support for metaclasses, support for metaprogramming (via descriptors such as decorators and properties), support for sophisticated operator overloading, and memory conserving features such as slots---may prove useful in advanced applications. Some limitations of Python are relevant for projects with very large numbers of agents---as an interpreted language Python will lack the speed of many compiled languages, and Python's restrictions on running process-bound threads on multiple processors may prove a restriction (on a single machine) in very special circumstances.[10] These issues are beyond the scope of the current paper.

3 A Simple Game

When discussing computational economics, nothing substitutes for actual code, so we will now create a very simple game. We do not address game theoretic considerations at this point: our two players will randomly chose their moves and ultimately receive the payoffs determined by the game.

For this first very simple game, we introduce only two types of object: a game, and players. Each object will instantiate a class, which encapsulates its characteristics (accessible via data attributes) and behavior (elicited via method attributes).[11] To allow a simple introduction to Python classes, we will strive for almost trivial classes in our first simple game. For example, our first players will be instances of a `RandomMover` class. When asked for a move, a `RandomMover` instance will do little more than generate a random value and test it against a probability of defection.

```
# Listing: simple_game.py

from user_utils import * #utilities: mean, transpose, and random

class RandomMover:
    p = 0.5  #probability of defection
    def move(self, game):
        return random() < self.p
    def record(self, game):
        pass

class SimpleGame:
    payoffmat = [ [(3,3),(0,5)] , [(5,0),(1,1)] ]
    def __init__(self, player1, player2):
        self.players = [player1, player2]
        self.opponents = {player1:player2, player2:player1}
    def run(self, maxiter=4):
        #create an empty list to store the history of moves
        self.history = list()
        for ct in range(maxiter):
            #get the new move for each player
            newmove = tuple( player.move(self) for player in self.players )
            #append the new player moves to the history list
            self.history.append(newmove)
        for player in self.players:
            #each player can record game info (e.g., this game played)
            player.record(self)
    def payoff(self):
        #make tuple: a payoff-pair for each move-pair
        move_payoffs = tuple( self.payoffmat[m1][m2] for (m1,m2) in self.history )
        #transpose move payoffs to get list of player payoffs
        player_payoffs = transpose(move_payoffs)
        #compute mean payoff for each player
```

[10]See http://blog.ianbicking.org/gil-of-doom.html for an accessible discussion.

[11]In this paper we directly access data attributes, which simplifies the presentation considerably. Those concerned that this violates OOP encapsulation practices will wish to look at how Python allows the use of properties to trap attribute references.

```
        average_payoff = tuple( mean(payoff) for payoff in player_payoffs )
        #pair each player with its payoff
        players_with_payoffs = zip(self.players, average_payoff)
        #return a mapping: player -> payoff
        return dict(players_with_payoffs)
```

The listing labeled `simple_game.py` contains the defintion of the `RandomMover` class. (Readers familiar with class and function definitions can skim the next few paragraphs.) A class definition starts with the keyword `class`, followed by the name (`RandomMover`) that we are giving the class, followed by a colon.[12] So the line `class RandomMover:` begins our definition of the `RandomMover` class, which is completed by the indented statements that follow. The first statement, `p = 0.5`, creates a data attribute that will be shared by all instances of the class. The variable `p` will be the probability of choosing the "defect" strategy. Since this data is shared by all instances, we call it a "class attribute". (For more detail, see the Python tutorial.)

The next statement is a function definition. Recall that blocks are defined in Python by level of indentation (and never, e.g., by the use of braces), so these function definitions must be indented to make them part of the class definition. (Similarly, the body of each function definition is given an *additional* level of indentation.)

A function definition starts with the keyword `def`, followed by the name that we are giving the function, followed by a (possibly empty) list of function parameters in parentheses, followed by a colon. Naturally, the function parameters are names that are *local* to the function definition. (So they will not conflict with the same names used elsewhere in our program.)

So the line `def move(self,game):` begins our definition of the `move` function. Clearly we named the function `move` because it generates the player moves. The function has two arguments. The second argument, named `game`, will be ignored by `RandomMover` instances but used by other player types (below). The first argument is called `self`, for reasons that we now discuss.

Recall that the function parameter `self` is a name that is *local* to the function definition. We can therefore freely choose a name for this argument of our function, but when the function definition is part of a class definition, it is conventional name the first argument `self`. This is because functions defined in our class become the methods of instances of the class, and a method always receives as an implicit first argument the instance on which it is called. (This is likely to sound obscure to a reader with no OOP background, but it should be clear by the end of this section.)

Next comes the body of the function, which in this case is a single statement: `return random() < self.p`. When this return-statement is executed, the function returns the boolean (True or False) value `random() < self.p`. This value it is computed as follows. The expression `random()` evaluates to a draw from a standard uniform distribution, so it is a floating point number between zero and one.[13] The expression `self.p` will retrieve the `p` attribute of `self`. The inequality comparison value is either True or False. We interpret True as "defect" and False as "cooperate".

The other class in `simple_game.py` is `SimpleGame`. A `SimpleGame` instance has more data and more methods, and it uses some new data structures, so we will approach it more slowly.

As usual, we begin our class definition with the statement `class SimpleGame:`. We then create a payoff matrix as a class attribute: `payoffmat = [ [(3,3),(0,5)] , [(5,0),(1,1)] ]`. This payoff matrix is created using two useful Python array-like data types: lists and tuples. (See the discussion of data types above.) The payoff matrix is a list of two lists, where each inner list contains two tuples. These tuples hold the move-based payoffs for two players. For example, the tuple `(3,3)` gives the payoffs to the players if both cooperate: for that move, each player gets a payoff of 3. In subsequent sections, we will consider the payoff matrix in much more detail.

Next we do something new: we introduce a function to initialize instances of the `SimpleGame` class. The name `__init__` is special: it is the name that must be used for the function that does the "initialization" when a new instance is created. We will call this the 'initialization function' for the class. In the initialization function, we have an opportunity to set initial values for instance attributes: in this case, a data attribute of the particular instance. We initialize any `SimpleGame` instance with two player instances. These will be the players for that game.

The `__init__` function of `SimpleGame` has three parameters: `self`, `player1`, and `player2`. As discussed above, `self` will be the local name of the instance that is being initialized. Similarly, the player

[12] In this paper we consider only classic classes.

[13] The function `random` can be found in the `random` module of the Python standard library. (At the top of `simple_game.py` we imported it via a `user_utils` module, which is listed in an appendix.) Each call to the `random` function returns one draw from a uniform distribution over the unit interval.

instances we pass in will have the local names `player1` and `player2`. So the statement `self.players = [player1,player2]` creates an attribute named `players` for the instance `self` and assigns as its value a list of the players we pass to the `__init__` function. Similarly, the statement `self.opponents = {player1:player2, player2:player1}` creates a mapping (from each player to the other, i.e., to the "opponent") and assigns it to the `opponents` attribute of the game (`self.opponents`). (This mapping will prove useful in later sections.)

Once we have an initialized `SimpleGame` instance, we can do two things with it. We can run the game by calling the instance's `run` method, and after that we can get the game payoff for each player by calling the `payoff` method. The associated code is very heavily commented and therefore largely self explanatory. Note that each time we run a game, the game records the history of the moves made. Naturally, this history of moves determines the game's payoff.

Recalling our earlier discussion of generator expressions, we see that the definition of `run` includes the a generator expression in `tuple( player.move() for player in self.players )`. Our earlier discussion explained that this expression will create a tuple as follows: for each player in in the game (i.e., in `self.players`), create a tuple element that is that player's move.

Now that we have written our two classes, we want to use them to run a game. A file containing Python code that can be imported is called a module. We can `import` objects from a module into any program we wish. Consider the following code, which will run our first game. Note that it begins by importing everything (`*`) from the `simple_game` module, thus making available the `RandomMover` and `SimpleGame` classes.

```
# Listing: play_simple_game.py

from simple_game import *

#create two players
player1 = RandomMover()
player2 = RandomMover()
#create and run the game
game = SimpleGame(player1, player2)
game.run()
#retrieve and print the payoffs
payoffs = game.payoff()
print payoffs[player1], payoffs[player2]
```

Our program for running the game is very simple. We start by creating two "players"---that is, `RandomMover` instances. A class in Python is a callable object; it returns an instance of the class when called. In order to produce a `RandomMover` instance, we call the `RandomMover` class by appending parentheses (just as when calling a function). So the expression `RandomMover()` creates a `RandomMover` instance. The two players in our game are "instances" of the `RandomMover` class. (We say that each player "instantiates" the `RandomMover` class.)

Once we have our two players (named `player1` and `player2`), we can create a `SimpleGame` for them to play: `game = SimpleGame([player1,player2])`. Note that this we call the class `SimpleGame` with two arguments: the two player instances required by the initialization function. When a `SimpleGame` instance is created, its `__init__` method is called with these arguments to initialize the instance. We defined this initialization to do one thing: it sets the value of the `players` attribute of our `SimpleGame` instance to a list containing our players. Note that it is the instance, not the class, which owns this data.

Finally we are ready to run our game and print the resulting payoffs. The `run` method of an instance of `SimpleGame` runs the game (when called). The game requests moves from the players, records their moves as the game history, and determines the game payoffs as the average of the move payoffs.

Recall that our `SimpleGame` class has as a class variable the probability of defection. We can change that probability by assigning to the class attribute: e.g., `SimpleGame.p = 0.1`. If we do this for a variety of probabilities, we get results like those in Table 1. Notice how the payoffs of both players tend to fall as the probability of defection rises. This tendency will occupy us at several points in the current paper.

Table 1: Game Payoffs at Various Defection Probabilities

Game Payoffs	**P{defect}**
(3.00, 3.00)	0.0
(3.02, 2.72)	0.1
(2.58, 2.88)	0.2
(2.44, 2.34)	0.3
(2.36, 2.46)	0.4
(2.48, 1.88)	0.5
(1.96, 1.86)	0.6
(1.92, 1.52)	0.7
(1.48, 1.38)	0.8
(1.30, 1.00)	0.9
(1.00, 1.00)	1.0

After analyzing just a few lines of code, many readers may find that Python programs are already becoming easy to read, so even nonprogrammers may now be feeling ready to try their hand at writing some code. This readability and simplicity is quite an astonishing strength of Python. What is more, our barebones `SimpleGame` class has enough features that it will remain useful throughout this paper.

4 The Prisoner's Dilemma

Our next project is to explore the consequencies of strategy choice in an iterated prisoner's dilemma. In a simple prisoner's dilemma there are two players each of whom has two strategies, traditionally called 'cooperate' (C) and 'defect' (D). The payoffs are such that---regardless of how the other player behaves---a player always achieves a higher individual payoff by defecting. However the payoffs are also such that when both players defect they each get an individual payoff less than if both had cooperated.

Real world applications of the prisoner's dilemma are legion [poundstone1992]. One unusual illustration considers a practice of competitive wrestlers: food deprivation and even radical restriction of liquids in an effort to lose weight in the days before a weigh-in. This is followed by rehydrating and binge eating after the weigh-in but before the match. Whether or not other wrestlers also follow this regime, individual wrestlers perceive it to provide a competitive advantage. But it is plausible that all players would be better off if the practice were ended (perhaps by requiring a weigh-in immediately before the match).

In its game theoretic formulation, stripped of the social ramifications of betrayal, the prisoner's dilemma suffers from a misnomer: there is a dominant strategy so there is no dilemma, and each player chooses without difficulty the individually rational strategy. Economists may be inclined to call this game "the prisoners paradox", to highlight the failure of individually rational action to produce an efficient outcome. In the present paper we nevertheless stick with the conventional name.

The payoff matrix in our `SimpleGame` is a very common representation of the prisoner's dilemma, common enough to be called canonical. We can use these payoffs to highlight the paradoxical nature of the prisoner's dilemma. Imagine that the game is played in three "countries" filled with different kinds of players. The population of the Country A comprises random players, as in our `SimpleGame` above, that choose either of the two strategies with equal probability. The average payoff to individuals in Country A is 2.25. People in Country B are instrumentally rational individuals: they act individually so as to produce the best individual outcome, without knowledge of the other player's strategy. (That is, each player always chooses to defect.) The average payoff to individuals in Country B is 1. People in Country C have somehow been socialized to cooperate: each individual always chooses the cooperative strategy. The average payoff to individuals in Country C is 3.

Researchers have extensively investigated this apparent conflict between rationality and efficiency in a variety of contexts. In the next section, we will show how a standard generalization of player strategies can help us usefully taxonomize this conflict. (See [stefansson2000_luna_stefansson] for a more detailed discussion.)

Defining the Prisoner's Dilemma

Some authors define the prisoner's dilemma to include two additional constraints on the payoff matrix: the matrix must be symmetric, and the payoff sum when both cooperate must be at least the payoff sum when one cooperates and the other defects. [rapoport_chammah-1965] introduce the latter restriction, which is sometimes justified as capping the payoff to defection or reducing the attractiveness of a purely random strategy. Many authors simply use without discussion the traditional payoff matrix above, which satisfies these constraints. As we shall see, even within these constraints, payoffs matter.

5 CD-I Game

When game payoffs resemble a prisoner's dilemma, observations of cooperative behavior are provocative: they may indicate deviations from the individually rational equilibrium, or they may indicate that the payoffs have been incorrectly understood [rapoport_chammah-1965]. In 1971, [trivers-1971-qrbio] proposed reciprocity as a basis of cooperation. A decade later, political science and evolutionary biology joined hands to show that reciprocity could serve a basis for cooperation even in the face of a prisoner's dilemma [axelrod_hamilton-1981science]. This led to the exploration of the iterated prisoner's dilemma, wherein players play a prisoner's dilemma repeatedly. Unlike the static version, an iterated prisoner's dilemma can actually involve a dilemma: defecting will raise the one period payoff but may lower the ultimate payoff. (An evolutionary aspect arises when strategy success influences strategy persistence; we will address this in subsequent sections.)

We now consider slightly more complex player types, using a characterization that is common in the literature. A playertype is characterized by a rule for initial action and by rules for reacting to past events: the likelihood of defection depends on whether the player faced cooperation of defection on the previous move of the game. Such players care whether the other player cooperated (C), the other player defected (D), or players are making their initial move (I). We will therefore call this the CD-I playertype. The CD-I playertype will be parameterized by a 3-tuple of probabilities: the probability of defecting when the other cooperated on the previous move, the probability of defecting when the other defected, and the probability of defecting on the initial move. The game played will again be iterations of our prisoner's dilemma.

An implementation of this is found in the `simple_cdi.py` listing. Once again the basic idea will be to create a game with two players. We will use this opportunity to illustrate delegation and inheritance. Note that we replace the `RandomMover` class with two classes: the `SimplePlayer` class and the `SimpleCDIType` class. That is, we separate out the notion of a "player" and a "playertype".

```
# Listing: simple_cdi.py

from simple_game import *

class SimplePlayer:
    def __init__(self, playertype):
        self.playertype = playertype
        self.reset()
    def reset(self):
        self.games_played = list()   #empty list
        self.players_played = set()  #empty set
    def move(self,game):
        #delegate move to playertype
        return self.playertype.move(self, game)
    def record(self, game):
        self.games_played.append(game)
        self.players_played.add( game.opponents[self] )

class SimpleCDIType:
    p = (0.5,0.5,0.5)
    def move(self, player, game):
        other = game.opponents[player]
        other_move = game.fetch_move(other,-1)
```

```
            if other_move is None:
                newmove = (random() < self.p[-1])
            else:
                newmove = (random() < self.p[other_move])
            return newmove

    class CDIGame(SimpleGame):
        def fetch_move(self, player, t):
            if self.history:
                player_idx = self.players.index(player)
                move = self.history[t][player_idx]
            else:
                move = None
            return move
```

Let us first examine the `SimplePlayer` class. A `SimplePlayer` instance must have a playertype, as can be seen by the `__init__` function: initialization involves setting the `playertype` attribute for the `SimplePlayer` instance. (The line `self.playertype = playertype` assigns the value of the local variable named `playertype` to the instance attribute of the same name, which is accessed as `self.playertype`.) During initialization, we also call `reset` to create a `games_played` attribute and a `players_played` attribute. (Since games know their players, the `players_played` attribute is in a sense redundant, but it will prove convenient.) These will serve as player "memory", which will be added to when the players `record` method is invoked by a game. (Recall that the `run` method of a `SimpleGame` instance will invoke the `record` method for each player.) Looking at the `move` function, we see that a player delegates its moves to its playertype: this is our first use of delegation, a common strategy in object oriented programming. (Note that when a player calls the `move` method of its playertype instance, it passes itself to the playertype. This means that, when determining a move, the playertype has access to player specific information.)

Next consider our `SimpleCDIType` class, which constructs our player type instances. The `SimpleCDIType` class has a class attribute, `p`, just as the `RandomMover` class did. Indeed, since we set all the elements of `p` to 0.5 in this class attribute, the basic `SimpleCDIType` determines moves in the same way as our `RandomMover` did. This can be seen by working through the `move` function for this class. For the initial move, the probability of defection is given by the last element of the tuple `p` (which we can index with -1). After the initial move, the probability of defection is conditional on the last move of the other player. (Note that the playertype fetches this move from the history of the player's game; see the `CDIGame` code for the details.) If the other player cooperated, the first element of `p` is the probability of defection. If the other player defected, the second element of `p` is the probability of defection.

Finally we consider our `CDIGame` class, which constructs our game instances. Notice that the `CDIGame` class is only partly specified. When we begin our class definition as `class CDIGame(SimpleGame):` we "inherit" data and behavior from the `SimpleGame` class. Specifically, we inherit the `__init__`, `run` and `payoff` functions, as well as the `payoffmat` class attribute. The use of inheritance is the typical idiom for code reuse in object oriented programming: we reuse data and behavior specified in the `SimpleGame` class definition.

Recall from the `__init__` function in the `SimpleGame` class definition that a game is initialized with two players. When run, the game is recorded by each `SimplePlayer` instance in its `games_played` list. This is what allows a player type to determine the player's move: when a player delegates a move to its player type, the playertype fetches from the game the *other* player's last move (i.e., the move with index -1). This is accomplished by invoking the game's `fetch_move` method. (Recall that the probability of defection generally is conditional on the other player's last move.) The `fetch_move` function in the `CDIGame` class definition provides the needed move fetching functionality by returning the appropriate element of the game's move history.

At first exposure, these linkages may feel a bit circuitous. When a game asks a player for a move. the player delegates moving to its playertype. (We cannot have the game request a move directly from a playertype: since multiple players may share a single playertype, there is no mapping from playertypes to players.) As part of selecting a move, the playertype first fetches the move-history from the game. Once the playertype has the move-history, it determines the current move for the player who delegated it, which it returns to the player, who finally returns it to the game that asked for it. If we accept that the player delegates the move to the playertype, and that the game is the natural place to store the move

history, these linkages appear natural.[14]

With our new class definitions in hand, we are ready to create playertypes, players, and a game. Of course we can construct an endless variety of CD-I playertypes. The present paper considers only the pure strategies: each element of the strategy 3-tuple is either zero or one. This gives us 8 possible player types and therefore 36 different games. (There are 36 unique pairings, since player order is irrelevant.) The listing `play_simple_cdi.py` conducts a round robin tournament, in the sense that it produces the outcomes for all these possible pairings. We will work through the code and then suggest a conclusion.

```
# Listing: play_simple_cdi.py

from simple_cdi import *

C,D = 0,1 #always cooperate (0) or always defect (1)
PURE_STRATEGIES = [ (C,C,C), (D,C,C), (C,D,C), (D,D,C),
                    (C,C,D), (D,C,D), (C,D,D), (D,D,D) ]

#create two player type instances
ptype1, ptype2 = SimpleCDIType(), SimpleCDIType()
#create two players, one for each player type
player1, player2 = SimplePlayer(ptype1), SimplePlayer(ptype2)
#create a game for the two players to play
game = CDIGame(player1,player2)
#play a game for each pair of strategies
for i, strategy1 in enumerate(PURE_STRATEGIES):
    for j in range(i+1):
        strategy2 = PURE_STRATEGIES[j]
        #assign strategies to player types
        ptype1.p, ptype2.p = strategy1, strategy2
        print "\nRun Game"
        game.run()
        payoffs = game.payoff()
        print strategy1, payoffs[player1]
        print strategy2, payoffs[player2]
```

As a preliminary, we create a list of all eight pure strategies in the CDI game. We create two `SimpleCDIType` player type instances, and then two players (initialized with our two player types), and finally we create a game for our two players to play. We then play the game for each pair of strategies and print the results of these games.[15] Table 2 summarizes the payoffs to the first player in each of these 36 games.

Table 2: Player 1 Payoffs with Different Strategies

	CCC	**DCC**	**CCD**	**DCD**	**CDC**	**DDC**	**CDD**	**DDD**
CCC	3.00	0.75	2.25	0.00	3.00	0.75	2.25	0.00
DCC	4.50	2.00	3.25	0.00	2.25	1.00	2.25	0.00
CCD	3.50	2.00	2.50	1.00	2.75	1.25	1.75	0.25
DCD	5.00	5.00	3.50	2.00	2.25	1.50	2.25	0.25
CDC	3.00	2.25	2.75	2.25	3.00	1.25	2.50	0.75
DDC	4.50	3.50	3.75	2.75	2.50	1.50	1.75	0.75
CDD	3.50	2.25	3.00	2.25	2.50	1.75	1.00	1.00
DDD	5.00	5.00	4.00	4.00	2.00	2.00	1.00	1.00

Perhaps the most striking thing about Table 2 is that playing DD-D produces a maximal payoff in all

[14] We can also endow players with memories, which can deviate unreliably from the game history or be limited in capacity. We do not explore these possibilities in the present paper.

[15] Recall the discussion of the built-in `enumerate` function in the section Why Python?.

but two cases: when pitted against the strategies CD-C and CD-D. Put in simple terms, DD-D would be dominant if the imitative/reciprocal strategies where removed from the strategy space. (These strategies are imitative in that they always adopt the opponent's previous move. They are reciprocal in that they always respond in kind.) The first of these, CD-C, is the famous Tit-For-Tat strategy. The robustness of this strategy in an IPD context has been long recognized [axelrod_hamilton-1981science]. Neither of these two strategies dominates the other. It is however noteworthy that CD-D holds its own against DD-D, whereas CD-C does not. Note too that two CD-C players will do much better than two CD-D players.[16] The subsequent sections explore some consequences of these observations.

6 Evolutionary Soup

We now consider a simple evolutionary mechanism: players are inclined to adopt winning strategies. Our first implementation of these evolutionary considerations will be in the context of random encounters between players. We introduce simple representations of a round and a tournament. A tournament consists of several rounds. On each round, players are randomly sorted into pairs, and each pair plays a CDIGame. After playing a game, a player switches (with some probability) to a winning strategy (for that game). We will characterize how an initially diverse group of players evolves over time.

The file `simple_soup.py` contains new classes to implement this evolutionary tournament. There are three classes: `SoupPlayer`, `SimpleRound`, and `SimpleTournament`. (The game played will be unchanged: `CDIGame`.)

```
# Listing: simple_soup.py

from simple_cdi import *

class SoupPlayer(SimplePlayer):
    selection_pressure = 1
    def choose_next_type(self):
        players = set(self.players_played)
        players.add(self)
        best_types = find_best_playertypes(players)
        self.next_playertype = randomchoice(best_types)
    def evolve(self):
        if random() < self.selection_pressure:
            self.playertype = self.next_playertype
    def get_payoff(self): #called by find_best_playertypes()
        payoff = 0
        for game in self.games_played:
            payoff += game.payoff()[self]
        return payoff

class SimpleRound:
    def __init__(self, player_list):
        self.players = player_list
    def run(self):
        self.prepare_players()
        shuffle(self.players)
        for game_players in groupsof(self.players, 2):
            game = CDIGame( *game_players )
            game.run()
        self.evolve_players()
    def prepare_players(self):
        for player in self.players:
            player.reset()
    def evolve_players(self):
        for player in self.players:
```

[16] While most (29/36) pairs of player will quickly reach an "equilibrium", in the sense that each finds a single move to repeat, a CD-C/CD-D pair will switch moves every iteration for as long as they play.

```
                player.choose_next_type()
            for player in self.players:
                player.evolve()

class SimpleTournament:
    nrounds = 10
    Round = SimpleRound
    def __init__(self, player_list):
        self.players = player_list
    def run(self):
        players = self.players
        self.playertype_history = list() #empty list
        self.record() #record playertype counts
        round = self.Round(players)
        for i in range(self.nrounds):
            round.run()
            self.record() #record playertype counts
    def record(self):
        counts = count_player_types( self.players )
        self.playertype_history.append( counts )
```

The `SoupPlayer` class inherits much of its behavior from `SimplePlayer`, but it has new attributes deriving from our desire that it be able to evolve its playertype. When we call the `choose_next_type` method of a `SoupPlayer` instance, this sets the `next_player_type` attribute. (The `next_player_type` is always set to a winning playertype, as determined by a `find_best_playertypes` utility, which calls the `get_payoff` method of each player.) However a player instance does not change its playertype until its `evolve` method is called. At that point, the the `selection_pressure` attribute determines the probability of adopting the `next_player_type`. (We let `selection_pressure` be a class variable with value 1; with this default the player will always adopt its `next_player_type`.)

The `SimpleRound` class is initialized with a list of players. When we call the `run` method of a `SimpleRound` instance, it randomly sorts the players into pairs, plays a `CDIGame` for each of these pairs, and then evolves each player's playertype (as previously described).[17]

A `SimpleTournament` instance is initialized with a player list. It will run (by default) ten rounds, as just described. That is the core functionality of a tournament instance, but we also endow a tournament with a memory: it keeps track of the playertype counts of each round. (There a many ways of handling such bookkeeping; this way was chosen for present paper due to its transparency.)

Starting with 50 of each of the eight player types, randomly paired, Table 3 shows how the player type counts evolve over a typical ten round tournament. Our previous examination of Table 2 has prepared us for the results: after 10 rounds, all surviving players are playing "defect" every move of every game. The average payoff per player has fallen to 1.

Table 3: Player Type Counts by Round

CCD	DCC	CDD	DDD	DCD	CDC	DDC	CCC
50	50	50	50	50	50	50	50
29	36	59	88	62	35	77	14
5	19	69	153	45	21	88	0
1	6	70	222	22	8	71	0
0	2	71	280	1	2	44	0

[17]This can raise the question: how randomly can we sort players into pairs? One common strategy is concatenating lists of each type of player and then permuting them (e.g., Stefansson 2000). We will adopt this approach as appropriate for the "evolutionary soup" model in this paper, but it is worth noting that some interesting computational problems are posed by shuffling. These have been heavily studied because of the role of shuffling in card games. With only 52 cards in a deck, there are 52! (or almost 2ˆ226) different possible shuffles. Suppose we are working with a simple RNG algorithm based on a 32-bit random number generator: this yields 2ˆ32 possible shuffles (one for each seed). This is radically less than the number of permutations. Only a small fraction of the possible shuffles are achievable. (Even modern generators with long periods can face this problem. The Mersenne Twister MT19937 for example, claims a period of 2ˆ19937-1, but might be implemented with a 32 bit seed. However the Python implementation allows a much larger seed size.)

Table 3: Player Type Counts by Round

CCD	DCC	CDD	DDD	DCD	CDC	DDC	CCC
0	1	68	303	0	2	26	0
0	0	66	326	0	0	8	0
0	0	59	341	0	0	0	0
0	0	57	343	0	0	0	0
0	0	54	346	0	0	0	0
0	0	52	348	0	0	0	0

7 Evolutionary Grid

The evolutionary prisoner's dilemma is often played on a grid, which simplifies exploring the implications for player type evolution of various characterizations of the neighborhood in which a player resides.[18] In this section we make use of a grid that is torus-like (in that it wraps around its edges), and we define the neighborhood of a player to include the player above, the player below, and the player to each side.[19] Each player has a fixed location on the grid. This means that every player has the same four neighbors throughout a tournament, although the player types of these players can of course evolve over time.

The implementation of an evolutionary grid profits substantially from our earlier work at representing players, games, rounds, and tournaments. The code in the `simple_grid.py` listing is all we need to add (aside from a grid object with a `get_neighbors` method, which is provided in the `user_utils` module--see the appendix). A `GridPlayer` inherits almost all its behavior from `SoupPlayer`: A `GridPlayer` is just a `SoupPlayer` that has a grid that knows know how to determine its neighbors. Similarly, the only thing we need to do to produce a `GridRound` is inherit from `SimpleRound` and over-ride the `run` definition. Even the run method is very modestly changed: in effect a player now has more than one neighbor, and correspondingly plays a `CDIGame` once with each neighbor. Recall that a player will know which games it has played, because its `games_played` attribute is appended when a `CDIGame` is run.

```
# Listing: simple_grid.py

from simple_soup import *


class GridPlayer(SoupPlayer):
    def get_neighbors(self): #delegate to the grid
        return self.grid.get_neighbors(self)

class GridRound(SimpleRound):
    def run(self):
        self.prepare_players()
        for player in self.players:
            neighbors = player.get_neighbors()
            for neighbor in neighbors:
                if neighbor not in player.players_played:
                    game = CDIGame(player, neighbor)
                    game.run()
                    payoffs = game.payoff()
        self.evolve_players()
```

[18] It is not always appreciated that a two-dimensional grid is just a convenient and intuitive conceptualization of relationships that are easily characterized in terms of a one-dimensional array. The class implementing the grid therefore need not rely on any kind of two-dimensional container object, although the class we provide as a user utility (see the appendix) does in fact do so: lists of lists provide fast and convenient access. (Users of the `numpy` module might instead consider using object arrays, which offer numpy's powerful indexing facilities.)

[19] This is known as the von Neumann neighborhood of radius 1, where a von Neumann neighborhood of radius r corresponds to the closed ball of radius r under the 1-norm.

The result of running such a tournament has often been viewed as surprising and interesting. [axelrod_hamilton-1981sci note that the "Tit-for-Tat" strategy (CD-C) is very robust in such settings, and indeed we find that it often squeezes out all other strategies. Figure 1 summarizes a typical tournament. (Only two of the eight player types are plotted.)

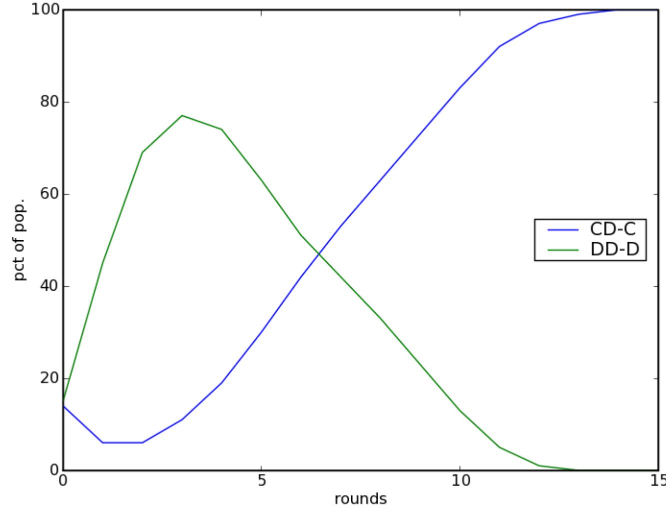

Figure 1: Evolving Strategy Frequencies: Evolutionary Grid

We can understand the success of the CD-C strategy that we see in Table 4 in terms of our results in Table 2. Consider two adjacent CD-C players on a grid that is otherwise entirely populated by DD-D players. Using the notation of [rapoport_chammah-1965], let the payoff matrix be represented symbolically as `[[(R,R),(S,T)],[(T,S),(P,P)]`.

Each of the two CD-C players will receive a payoff of R+3S from its first move (summed across all 4 neighbors) and a payoff of R+3P from its second and subsequent moves (summed across all 4 neighbors). Given a 4 move game, that is a total payoff of 4R+9P. In contrast a DD-D player who has a CD-C player as a neighbor will receive a payoff of T+3P from its first move (summed across all 4 neighbors). and a payoff of 4P for each move thereafter (summed across all 4 neighbors). Given a 4 move game, that is a total payoff of T+15P. So the CD-C type players get a higher payoff as long as 4R+3S>T+6P.

In the literature it is common to let each game run for at least 4 moves. Using the canonical prisoner's dilemma payoffs, `[[(R,R),(S,T)],[(T,S),(P,P)]=[[(3,3),(0,5)],[(5,0),(1,1)]`. This means that the highest average payoff will go to the CD-C players. (The two CD-C players in this case each receive a payoff of 21, while a DD-D neighbor will receive 20.) The neighboring DD-D players will therefore switch strategies (with probability determined by the selection pressure) thereby increasing the number of adjacent CD-C players. Naturally, the "Tit-for-Tat" strategy quickly takes over the grid!

Contrary to a common impression, this does *not* mean that CD-C is the best strategy on our evolutionary grid. Two obvious changes will make DD-D and CD-D win the evolutionary race. First, we could reduce the number of moves in each game to 3. (The two CD-C players in this case each receive a payoff of 3R+6P=15, while a DD-D neighbor will receive T+11P=16.) Alternatively, we could deviate from the canonical payoff matrix we have been using.

We will explore the second possibility. To keep the discussion focused, we will focus on a single parameter: P. Consider for example raising the value of P, retaining the canonical payoffs T=5, S=0, and R=3. Recall our isolated CD-C pair will beat their DD-D neighbors as long as 4R+9P+3S>T+15P: that is, as long as 7/6>P. If we raise P above this threshold, then the DD-D neighbors will win and the CD-C pair will switch strategies. Intuitively, if we reduce the punishment defection of defection, we expect defection to be more likely to persist as a strategy [rapoport_chammah-1965].

Raising P>7/6 eliminates the ability of isolated pairs to spread, but but it is still plausible that CD-C players will dominate in a given tournament. Larger groups of CD-C players may be part of an initial distribution of players, or may emerge temporarily as other player types vanish, and these will achieve higher payoffs. To make this concrete, consider four CD-C player arranged in a square, who are completely surrounded by DD-D players.

Each of the four CD-C players has two CD-C neighbors and two DD-D neighbors. Each therefore receives a payoff of 8R+6P+2S (one round, four moves per game). The neighboring DD-Ds each get a payoff of T+15P. The CD-Cs have the better payoff if 8R+6P+2S>T+15P. Again retaining T=5, R=3, and S=0, this means that as long as 19/9>P the CD-C players will have the higher payoff. All the DD-D neighbors will thereafter adopt CD-C strategies, and this expansion will continue outward from the initial square.

In more general terms, as we raise the payoff to mutual defection, it becomes harder and harder for the CD-C strategy to win out over the DD-D strategy on our evolutionary grid. The canonical payoff matrix makes it almost inevitable the the CD-C player type will come to dominate the grid. As we raise the value of P high enough, it becomes almost inevitable for the DD-D player type to come to dominate the grid. This happens relatively soon: for the payoff matrix `[[(3,3),(0,5)],[(5,0),(1.8,1.8)]]`, the outcome in Figure 2 is rather typical.

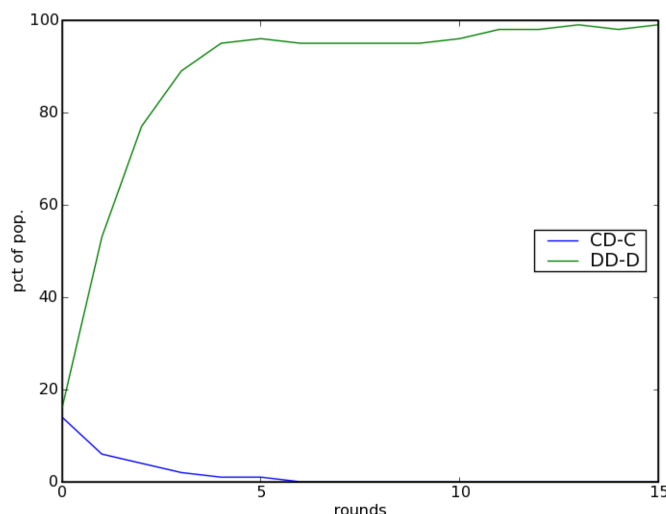

Figure 2: Evolution of Player Types: Less Punishment of Mutual Defection

8 Commitment

We have seen that, with a standard grid topology for an evolutionary iterated prisoner's dilemma, cardinal aspects of the payoff matrix affect the likelihood that a random mix of player types will evolve toward defection or cooperation. We now consider an additional influence on this outcome: commitment, considered as an unwillingness to imitate the strategies of others.

As a tentative application, consider software patents. Some software manufacturers and some researchers contend that the recent explosion of software patenting can damage software productivity---for example, by creating patent thickets. (See [isaac_park04_colombatto] for an extensive discussion.) The United States appears particularly susceptible to such problems: both manufacturers and researchers have argued that nonobviousness requirements are not being enforced by the US Patent and Trademark Office. Some firms have responded to the situation by asserting that they will not enforce certain collections of patent claims except in retaliation for patent claims against them. This may improve aggregate productivity and overall industry profits by increasing knowledge flows and reducing litigation costs, but at the individual firm level the temptation to defect will be present. Some firms, most famously IBM, have responded to this situation by publically a commitment not to "defect".

We offer an extremely simple representation of such commitment. Consider the listing from `commit_player.py`. The `CGridPlayer` is identical to our `GridPlayer` with one small change: if assigned a CD-C playertype there is some probability (`p_commit`) that we set `True` as the value of the player's `committed` attribute. We refer to such a player as a 'committed player'. A committed player will not change player type when we call its `evolve` method. Note that commitment is determined at the beginning of a tournament (during player initialization) and does not evolve. (Thus the name.)

```
# Listing: commit_player.py

from simple_grid import *

class CGridPlayer(GridPlayer):
    p_commit = 0
    def __init__(self, playertype):
        GridPlayer.__init__(self,playertype)
        if playertype.p == (0,1,0):
            self.committed = random() < self.p_commit
        else:
            self.committed = False
    def evolve(self):
        if not self.committed and random() < self.selection_pressure:
            self.playertype = self.next_playertype
```

The `CGridPlayer` class is defined so that instances behave identically to `GridPlayer` instances: the class variable `p_commit` is assigned a value of 0 in the class definition. However this behavior can be changed by assigning new values to `p_commit`. Intuition suggests that the existence on the grid of committed Tit-for-Tat players makes it more likely that a high payoff cluster of Tit-for-Tat players can form during the tournament.[20] An illustrative verification of this intuition is offered in Figure 3, which shows the concentration of DD-D and CD-C players over time when the probability of a CD-C player committing is 0.2. The outcomes closely resemble Figure 1 even though the payoffs are those used in producing Figure 2. Holding all other aspects of the game are constant across the two scenarios, but leaving `p_commit=0`, produces a outcome closely resembling Figure 2. The presence of committed players radically alters the evolution of playertypes.

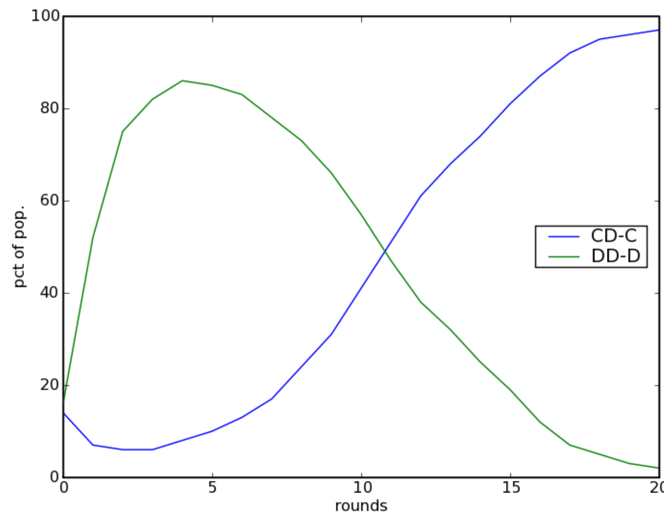

Figure 3: Evolution of Player Types: With Committed Players

Naturally this verification of our intuition is only illustrative. The initial configuration of players on the grid is random, and a given value of `p_commit` cannot tip the balance for all distributions. (For example, even a high probability that Tit-for-Tat players are committed cannot help if chance dictates that there are few of them and they are each surrounded by players that will exploit them, such as DD-D players.) The important point, however, is that a modest probability of commitment can radically change the playertype evolution. Correspondingly, there is a change in the average payoff received by players on the grid. This may be considered a social consequence of commitment.

[20] We could elaborate on this idea by making commitment less permanent. For example, rather than simply sticking with its strategy, a committed player might have some relatively high probability of retaining its strategy. As one would expect, decreases in this probability can within limits be offset by increases in `p_commit`.

9 Conclusion

Researchers using computer simulation of economic interactions often promote ACE methods as a "third way" of doing social science, distinct from both pure theory and from statistical exploration [axelrod1997abm]. One hope of ACE researchers is that unexpected but useful (for prediction or understanding) aggregate outcomes will "emerge" from the interactions of autonomous actors. The iterated prisoner's dilemma is a good illustration of the fruition of this hope: more than two decades of computational exploration have delivered many interesting and surprising results. Classic among these is the high fitness of the "Tit-for-Tat" player type on an evolutionary grid.

This paper uses the iterated prisoner's dilemma as a spring board for exposition and exploration. The paper demonstrates that useful ACE models can be constructed with surprising ease in a general purpose programming language, if that language has the flexibility and readability of Python. Python's combination of compactness and readability means that a full listing of the code needed to produce the half dozen simulations described in this paper takes only a few pages.

The simulations presented accomplish several purposes. The initial simulations lay bare the underpinnings of some classic results, demonstrating in the process the usefulness of an object approach to modeling IPDs. We then show that, by manipulating a single entry of the payoff matrix while remaining within the standard definition of the prisoner's dilemma, payoff matrix values are crucial to outcomes on an evolutionary grid. This acts as a cautionary tale for those relying on the canonical payoffs for their simulations. Finally, we introduce a simple notion of commitment, and we find dramatic implications for the evolution of strategies over time. Paradoxically, although committed players do not imitate the most successful neighboring strategy, they may generate social outcomes that increase the payoffs for all players (including the committed players).

10 Appendix A

This appendix contains a small number of user utilities that were used to simplify the code samples in the main text.

```
# File: user_utils.py

from random import random, shuffle
from random import choice as randomchoice  #a reader convenience
import itertools

################################################################
###########  simple utilities  ################################
################################################################

def mean(lst):  #simplest computation of mean
    n = len(lst)*1.
    return sum(lst)/n

def transpose(seqseq): #simple 2-dimensional transpose
    return zip(*seqseq)

def find_best_playertypes(players): #called by SoupPlayer
    best_players = list()
    best_payoff = -1e9
    for player in players:
        payoff = player.get_payoff()
        if payoff == best_payoff:
            best_players.append(player)
        if payoff > best_payoff:
            best_payoff = payoff
            best_players = [player]
    return list( player.playertype for player in best_players )
```

```
def count_player_types(players): #called by SimpleTournament
    summary = dict() #empty dictionary
    for player in players:
        ptype = player.playertype
        summary[ptype] = summary.get(ptype,0) + 1
    return summary

def groupsof(lst,n): #used to populate SimpleTorus by row
    """Return len(self)//n groups of n, discarding last len(self)%n players."""
    #use itertools to avoid creating unneeded lists
    return itertools.izip(*[iter(lst)]*n)

###############################################################
############## very simple grid class  #####################
###############################################################
class SimpleTorus:
    neighborhood = [(0,1),(1,0),(0,-1),(-1,0)]
    def __init__(self,(nrows,ncols)):
        self.nrows = nrows
        self.ncols = ncols
        self.neighbors = dict() #empty dictionary
    def populate(self,players):
        self.players = list( list(group) for group in groupsof(players, self.ncols)
        for r in range(self.nrows):
            for c in range(self.ncols):
                player = self.players[r][c]
                player.grid = self
                player.loc = r,c
    def get_neighbors(self,player):
        if player in self.neighbors:
            neighbors = self.neighbors[player]
        else:
            player_row, player_col = player.loc
            neighbors = list() #empty list
            for offset in self.neighborhood:
                dc, dr = offset #bc x,y neighborhood
                r = (player_row + dr) % self.nrows
                c = (player_col + dc) % self.ncols
                neighbor = self.players[r][c]
                neighbors.append(neighbor)
        return neighbors
```

11

References

[axelrod1997abm] Axelrod, Robert. 1997. The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration.

[hahn1991ej] Hahn, Frank. 1991. The Next Hundred Years. ej 101, 47--50.

[bonabeau2002pnas] Bonabeau, Eric. 2002. Agent-based Modeling: Methods and Techniques for Simulating Human Systems. pnas 99, 7280--7.

[gilbert_bankes2002pnas] Gilbert, Nigel, and Steven Bankes. 2002. Platforms and Methods for Agent-Based Modeling. pnas 99, 7197--7198.

[luna_stefansson2000swarm] Luna, Francesco, and Benedikt Stefansson. 2000. Economic Simulations in {Swarm}: Agent-Based Modelling and Object Oriented Programming.

[kak2003oop] Kak, Avinash. 2003. Programming with Objects: A Comparative Presentation of Object Oriented Programming with C++ and Java.

[matsumoto_kurita-1998-acmt] Matsumoto, M, and Y Kurita. 1998. Mersene Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM Transactions on Modeling and Computer Simulation 8, 3--30.

[poundstone1992] Poundstone, W. 1992. Prinsoner's Dilemma.

[stefansson2000_luna_stefansson] Stefansson, Benedikt. 2000. Simulating Economic Agents in {Swarm}.

[rapoport_chammah-1965] Rapoport, Anatol, and Albert M Chammah. 1965. Prisoner's Dilemma: A Study in Conflict and Cooperation.

[trivers-1971-qrbio] Trivers, Robert L. 1971. The Evolution of Reciprocal Altruism. Quarterly Review of Biology 46, 35--57.

[axelrod_hamilton-1981science] Axelrod, Robert, and William D Hamilton. 1981. The Evolution of Cooperation. Science 211, 1390--6.

[isaac_park04_colombatto] Isaac, Alan G, and Walter G Park. 2004. On Intellectual Property Rights: Patents vs. Free and Open Development.