# An efficient minimax implementation for noughts and crosses

Izaak van Dongen

July 12, 2018

## Contents



## List of Listings



## 1 Introduction

This is a set of programs that enable a user to play noughts and crosses "locally with a friend" (themselves), or to play against a computer, or even to watch the computer desperately trying to win against itself. It also supports arbitrarily sized boards, and probably several cases of overly optimised approaches to problems like the detection of a winning condition on a board.

Internally, the board is represented as one-dimensional array (arraylist) of any of {`None`, `True`, `False`}, corresponding to {Empty, Crosses, Noughts}. The single dimensionality is actually pretty natural, as a noughts and crosses board is not fundamentally a list of lists, but a set of tiles with some subsets considered to be "in a row". The board is enumerated from the top-left tile, left-to-right, top-to-bottom. When x-y indexing is used, the co-ordinate $(x, y)$ corresponds to the index $3y + x$. This has a couple of nice properties which will come up later.

Minmax is implemented by a single "optimise" function that recursively searches the tree, with several specialisations over general minmax. These include for example not having to keep maximising after encounting a win, as the evaluation heuristic is known to only be able to evaluate the board as {win, lose, draw}.

The tests for winning conditions are implemented in listing 2. They work by instantiating a single list of indices for each subset considered a "row", and then creating an array indexed by board indices of lists of these subsets, where the subsets stored at a board index represent all the rows that tile is a part of.

Further explanation of implementation details may be included in docstrings in the source.

I decided to develop in Python as I had some quite ambitious ideas about the algorithm, which I thought would be easier to express by using the more functional side of Python. Python is also nicely expressive, which makes it easier to develop without getting bogged down in syntax. Similarly, I wrote a command-line program because it doesn't involve using Lazarus.

## 2 Implementation

Listing 1 has some boilerplate classes that are shared throughought, to give some abstract representation of the game.

```
1  """
2  Base module with a couple of shared types
3  """
4
5  from enum import Enum
6
7  class GameFinish(Exception):
8      """
9      Exception to throw when a game is finished
10     """
11     pass
12
13 class Win(GameFinish):
14     """
15     If a game is won
16     """
17     pass
18
19 class Draw(GameFinish):
20     """
21     If a game is drawn
22     """
23     pass
24
25 class State(Enum):
26     """
27     Enum to represent the possible states of a board
28     """
29     DRAW = 0
30     X_WIN = 1
31     O_WIN = 2
```

```
32      NEUTRAL = 3
```

Listing 1: `base.py`: Some shared base classes

Listing 2 implements all of the "checking" - this is mostly the generation of rows and groups, and then some simple list comprehensions and generator expressions to evaluate a board using these.

The subsets are represented as Python `ranges`. This is possible as each row is a linear succession of indices in this linearisation of the board. These ranges are implemented very efficiently, using only integer arithmetic to generate intermediate range values or to determine membership.

Because these groups are used often, they are stored in a registry and generated when needed. This is all handled by the module itself, so a user only needs to call `get_groups`.

```
1   """
2   Functions implementing grid checks, efficiently.
3   """
4
5   from argparse import ArgumentParser
6
7   from base import State
8   from formatting import get_board_template
9
10  def get_args():
11      """
12      Get size of board
13      """
14      parser = ArgumentParser(description=__doc__)
15      parser.add_argument("n", type=int, help="size of board to check")
16      return parser.parse_args()
17
18  def _make_groups(n):
19      """
20      This function should not be accessed directly - use get_groups.  Construct
21      the list of "groups" present on the board - rows, columns, diagonals. Groups
22      are represented as Python ranges because they're all around suitable -
23      efficient, linear etc. Return a list of lists of these groups, where
24      indexing by board position returns the list of groups that that position is
25      contained in. This is pretty memory-efficient as each actual list is only
26      stored once, and the remainder of the data structure is primarily pointers.
27      """
28      rows = ([r for i in range(n)
29                  for r in [range(i * n, i * n + n), range(i, i + n ** 2, n)]]
30            + [range(0, n ** 2, n + 1), range(n - 1, n ** 2 - n + 1, n - 1)])
31
32      return [[r for r in rows if i in r] for i in range(n ** 2)]
33
34  # The registry for groups
35  GROUP_REGISTRY = {}
36
37  def get_groups(n):
38      """
```

```
39      Access a list of groups. This checks if the groups have been generated and
40      cached already, and if not, does so, caching them in the process, and then
41      returns.
42      """
43      if n in GROUP_REGISTRY:
44          return GROUP_REGISTRY[n]
45      else:
46          GROUP_REGISTRY[n] = _make_groups(n)
47          return GROUP_REGISTRY[n]
48
49  def is_run(board, pos, n):
50      """
51      Check if one tile is a part of any complete groups.
52      """
53      return any(len(set(board[i] for i in group)) == 1 for group in get_groups(n)[
        ↪  pos])
54
55  def get_state(board, n):
56      """
57      Get state of board. This function is given no information about position
58      so is necessarily much slower. If you do know the last played tile, use
59      is_run instead, as this only checks all groups pertaining to that tile.
60      """
61      for pos, m in enumerate(board):
62          if m is not None:
63              if is_run(board, pos, n=n):
64                  if m:
65                      return State.X_WIN
66                  return State.O_WIN
67      if board.count(None) == 0:
68          return State.DRAW
69      return State.NEUTRAL
70
71  def show_groups(n):
72      """
73      Demo function which displays each groups using functions from formatting.py.
74      """
75      board_temp = get_board_template(n)
76      for ind, pos in enumerate(get_groups(n)):
77          print(ind, pos)
78          for g in pos:
79              dft = [" "] * n ** 2
80              for i in g:
81                  dft[i] = "G"
82              dft[ind] = 'M'
83              print(board_temp.format(*dft), end="\n\n")
84
85  if __name__ == "__main__":
86      args = get_args()
87      show_groups(args.n)
```

Listing 2: `checking.py`: Implementation of minimax

When the module is run directly, it dumps all of the groups for a given board size. Listing 3 has a sample of this behaviour (and also demonstrates the nice board formatting).

```
% python checking.py 3
0 [range(0, 3), range(0, 9, 3), range(0, 9, 4)]
   0   1   2
 0 M | G | G
  ---+---+---
 1   |   |
  ---+---+---
 2   |   |

   0   1   2
 0 M |   |
  ---+---+---
 1 G |   |
  ---+---+---
 2 G |   |

   0   1   2
 0 M |   |
  ---+---+---
 1   | G |
  ---+---+---
 2   |   | G

1 [range(0, 3), range(1, 10, 3)]
   0   1   2
 0 G | M | G
  ---+---+---
 1   |   |
  ---+---+---
 2   |   |

   0   1   2
 0   | M |
  ---+---+---
 1   | G |
  ---+---+---
 2   | G |

2 [range(0, 3), range(2, 11, 3), range(2, 7, 2)]
   0   1   2
 0 G | G | M
  ---+---+---
 1   |   |
  ---+---+---
 2   |   |

```

```
   0   1   2
0  |   | M
 ---+---+---
1  |   | G
 ---+---+---
2  |   | G

   0   1   2
0  |   | M
 ---+---+---
1  | G |
 ---+---+---
2 G |   |
```

Listing 3: Output of `checking.py`

Listing 4 shows the implementation of minimax with all previously described bells and whistles. When called directly, it can demonstrate its approach to a particular board state, shown in listing 5. My favourite part of this program is the generator `generate_moves`.

```
"""
Implementation of the special case of the minmax algorithm, suited to noughts
and crosses.
"""

from textwrap import indent
from traceback import extract_stack

from base import Win
from checking import State, is_run
from formatting import print_board, strfboard, syms
from interface import SquareBoard, isqrt

from argparse import ArgumentParser

def get_args():
    """
    Get arguments if a demo run is being executed. A demo run will just
    determine the move to be used against a preset board, verbosely by default.
    """
    parser = ArgumentParser(description=__doc__)
    parser.add_argument("-b", "--board", type=SquareBoard, default='_o__x____',
                        help='The initial board state')
    parser.add_argument("-q", "--quiet", action="store_true",
                        help="do not print minmax tree")
    return parser.parse_args()

# boolean-indexed array to get states compactly and quickly
state_from_bool = [State.O_WIN, State.X_WIN]

def optimise(evaluations, is_crosses, minimise):
```

```
32      """
33      "Optimise" a sequence of results for a given player. This simultaneously
34      implements both minimisation and maximisation with a bit of Boolean logic.
35      It also knows how to short-circuit - if maximising, a win is known to be the
36      best possible case and vice verse.
37      """
38      LOSE_STATE, WIN_STATE = state_from_bool[minimise ^ (not is_crosses)],
        ↪  state_from_bool[minimise ^ is_crosses]
39      draw_seen = False
40      for e in evaluations:
41          if e == LOSE_STATE:
42              return e
43          elif e == State.DRAW:
44              draw_seen = True
45      if draw_seen:
46          return State.DRAW
47      return WIN_STATE
48
49  def generate_moves(board, is_crosses):
50      """
51      Generate possible moves on a board for a certain player. This works by
52      mutating the actual board array for each possible move, followed by yielding
53      both the move and the board. This is useful as it allows "iteration" over
54      moves, while also being memory-efficient (which leads to time efficiency as
55      there is no allocation overhead).
56      A finally clause implements the restoration of the board, which guarantees
57      that the board will retain its state from before after this function exits,
58      even if the function is interrupted by, for example, a break.
59      """
60      for ind, i in enumerate(board):
61          if i is None:
62              try:
63                  board[ind] = is_crosses
64                  yield ind, board
65              finally:
66                  board[ind] = None
67
68  def evaluate_board(board, is_crosses, crosses_playing, prev_move, depth, n,
69        verbose=False):
70      """
71      Evaluate a board-state for a given player. This recursively generates moves,
72      evaluates them and optimises them.
73      Allows printing diagnostics with the verbosity parameter. It will indent
74      depending on the current length of the callstack, which helps keep track of
75      the recursion.
76      """
77      verbose and print(indent("Examining as {} {}:\n{}"
78                               .format(syms[crosses_playing], depth,
79                                  indent(strfboard(board, n), ' ')),
80                           ' ' * len(extract_stack())))
81      if is_run(board, prev_move, n):
```

```
82          state = state_from_bool[not crosses_playing]
83          verbose and print(indent("State here: {}"
84                    .format(state), " " * len(extract_stack())))
85          return state
86      elif depth == len(board):
87          verbose and print(indent("Draw here", " " * len(extract_stack())))
88          return State.DRAW
89      else:
90          return optimise(
91                  (evaluate_board(board, is_crosses, not crosses_playing,
92                                  move, depth + 1, n, verbose=verbose)
93                    for move, board in generate_moves(board, crosses_playing)),
94                   is_crosses, not crosses_playing ^ is_crosses)
95
96  def get_computer_move(board, is_crosses, n, verbose=False):
97      """
98      Apply board evaluation to all possible moves and
99      select, in order:
100     - A winning move
101     - A drawing move
102     - Any move (which will be a losing move)
103     """
104     # optimisation: play here for an empty board, because searching through the
105     # whole board's tree is known to be unnecessary
106     if all(i is None for i in board):
107         return 0
108     WIN_STATE = state_from_bool[is_crosses]
109     moves = generate_moves(board, is_crosses)
110     draw = None
111     for move, board in moves:
112         ev = evaluate_board(board, is_crosses, not is_crosses, move,
113                             len(board) - board.count(None), n, verbose=verbose)
114         if ev == WIN_STATE:
115             verbose and print("Win incoming")
116             return move
117         elif ev == State.DRAW:
118             verbose and print("Draw forcable")
119             draw = move
120     if draw is not None:
121         return draw
122     return board.index(None)
123
124 def do_computer_move(board, is_crosses, n, verbose=False):
125     """
126     Wraps get_computer_move to print some stuff, mutate the board and check for
127     winning conditions.
128     """
129     move = get_computer_move(board, is_crosses, n, verbose=verbose)
130     board[move] = is_crosses
131     print("Computer plays at ({}, {})".format(move % n, move // n))
132     print_board(board, n)
```

```
133     if is_run(board, move, n):
134         raise Win("I'm sorry, Dave. I'm afraid I can't do that.")
135 
136 if __name__ == "__main__":
137     args = get_args()
138     board = args.board
139     n = isqrt(len(args.board))
140     print(board, n)
141     print_board(board, n)
142     do_computer_move(board, True, n, verbose=not args.quiet)
```

Listing 4: `computer.py`: Implementation of minimax

```
1  % python computer.py -b "x__ _o_ o_x"
2  [True, None, None, None, False, None, False, None, True] 3
3     0   1   2
4   0 X |   |
5    ---+---+---
6   1   | O |
7    ---+---+---
8   2 O |   | X
9 
10     Examining as O 5:
11         0   1   2
12       0 X | X |
13        ---+---+---
14       1   | O |
15        ---+---+---
16       2 O |   | X
17        Examining as X 6:
18            0   1   2
19          0 X | X | O
20           ---+---+---
21          1   | O |
22           ---+---+---
23          2 O |   | X
24        State here: State.O_WIN
25     Examining as O 5:
26         0   1   2
27       0 X |   | X
28        ---+---+---
29       1   | O |
30        ---+---+---
31       2 O |   | X
32        Examining as X 6:
33            0   1   2
34          0 X | O | X
35           ---+---+---
36          1   | O |
37           ---+---+---
38          2 O |   | X
```

```
            Examining as O 7:
                0   1   2
              0 X | O | X
               ---+---+---
              1 X | O |
               ---+---+---
              2 O |   | X
               Examining as X 8:
                   0   1   2
                 0 X | O | X
                  ---+---+---
                 1 X | O | O
                  ---+---+---
                 2 O |   | X
                  Examining as O 9:
                      0   1   2
                    0 X | O | X
                     ---+---+---
                    1 X | O | O
                     ---+---+---
                    2 O | X | X
                   Draw here
               Examining as X 8:
                   0   1   2
                 0 X | O | X
                  ---+---+---
                 1 X | O |
                  ---+---+---
                 2 O | O | X
               State here: State.O_WIN
            Examining as O 7:
                0   1   2
              0 X | O | X
               ---+---+---
              1   | O | X
               ---+---+---
              2 O |   | X
            State here: State.X_WIN
          Examining as X 6:
              0   1   2
            0 X |   | X
             ---+---+---
            1 O | O |
             ---+---+---
            2 O |   | X
             Examining as O 7:
                 0   1   2
               0 X | X | X
                ---+---+---
               1 O | O |
                ---+---+---
```

```
             2 O |   | X
            State here: State.X_WIN
         Examining as X 6:
             0   1   2
           0 X |   | X
            ---+---+---
           1   | O | O
            ---+---+---
           2 O |   | X
            Examining as O 7:
                0   1   2
              0 X | X | X
               ---+---+---
              1   | O | O
               ---+---+---
              2 O |   | X
            State here: State.X_WIN
         Examining as X 6:
             0   1   2
           0 X |   | X
            ---+---+---
           1   | O |
            ---+---+---
           2 O | O | X
            Examining as O 7:
                0   1   2
              0 X | X | X
               ---+---+---
              1   | O |
               ---+---+---
              2 O | O | X
            State here: State.X_WIN
Win incoming
Computer plays at (2, 0)
   0   1   2
 0 X |   | X
  ---+---+---
 1   | O |
  ---+---+---
 2 O |   | X
```

Listing 5: Output of `computer.py`

Listing 6 contains the code that formats boards as seen in previous listings (such as 5 and 3). It works by generating a "template" string, which is a string that is in a format that can be string-interpolated by `str.format`. It uses a similar registry principle to listing 2.

It makes most judicious use of Python various inline iteration capability, relegating the template generation to a single lexical line of code.

It can also demo its functionality when run directly, as shown in listing 7

```
"""
Pretty-printing OXO boards, using pre-calculated templates.
"""

from argparse import ArgumentParser
from random import choices

def get_args():
    """
    Get size of demo board in case of demo run
    """
    parser = ArgumentParser(description=__doc__)
    parser.add_argument("n", type=int, help="size of demo board")
    return parser.parse_args()

def _make_board_template(n):
    """
    Do not use this function. Use get_board_template instead.
    Generate a str.format compatible template to format a noughts and crosses
    board. This is a lot easier and faster than dynamically generating all of
    the "structure" of the board every time.
    """
    return ("{}\n{}"
              .format("".join(map("{:4}".format, range(n))),
                  "  {}".format("+".join(["---"] * n))
                      .join("\n\n").
                          join(map("{0[0]:2}{0[1]}".format,
                                   enumerate(["|".join([" {} "] * n)] * n)))))

# registry to cache templates
BOARD_REGISTRY = {3: _make_board_template(3)}

def get_board_template(n):
    """
    Get a template by checking to see if it has already been calculated and
    cached, and otherwise doing so before returning it. This layer of
    abstraction prevents any arduous calculation on module import, but rather
    incurs a slight penalty on first usage of the function (which is more likely
    through another function).
    """
    if n in BOARD_REGISTRY:
        return BOARD_REGISTRY[n]
    else:
        BOARD_REGISTRY[n] = _make_board_template(n)
        return BOARD_REGISTRY[n]

def strfboard(board, n):
    """
    Format standard board representation as string.
    """
    return get_board_template(n).format(*map(get_sym, board))
```

```
52
53  def print_board(board, n):
54      """
55      Print standard board representation as string.
56      """
57      print("{}\n".format(strfboard(board, n=n)))
58
59  syms = "OX"
60
61  def get_sym(i):
62      """
63      Translate (None, True, False) to " XO"
64      """
65      if i is None:
66          return " "
67      return syms[i]
68
69  if __name__ == "__main__":
70      args = get_args()
71      n = args.n
72      print("{0}x{0} template:\n{1}".format(n, get_board_template(n)))
73      print("\nrandom {0}x{0} board:".format(n))
74      print(strfboard(choices([None, True, False], k=n**2), n=n))
```

Listing 6: `formatting.py`: Formatting internal board arrays as strings

```
1  % python formatting.py 5
2  5x5 template:
3     0   1   2   3   4
4   0 {} | {} | {} | {} | {}
5    ---+---+---+---+---
6   1 {} | {} | {} | {} | {}
7    ---+---+---+---+---
8   2 {} | {} | {} | {} | {}
9    ---+---+---+---+---
10  3 {} | {} | {} | {} | {}
11   ---+---+---+---+---
12  4 {} | {} | {} | {} | {}
13
14 random 5x5 board:
15    0   1   2   3   4
16  0 X |   | X | X | X
17   ---+---+---+---+---
18  1 X | X | O | X |
19   ---+---+---+---+---
20  2   | X | X | O | O
21   ---+---+---+---+---
22  3 X |   |   |   |
23   ---+---+---+---+---
24  4 O | X | O | X | X
```

Listing 7: Output of `formatting.py`

Listing 8 deals with some of the really dull stuff, like getting user input.

```
"""
Handling and verifying user input
"""

from base import Win
from formatting import print_board, syms
from checking import is_run

# boolean-indexed array to get a string name for player
name_from_bool = ["noughts", "crosses"]

# dictionary to translate symbols to internal representation of tile states
state_from_string = {"_": None, 'x': True, 'o': False}

def isqrt(n):
    """
    Calculate the integer square root of a number using the "bit-shift"
    algorithm.
    """
    if n < 2:
        return n
    else:
        small = isqrt(n >> 2) << 1
        large = small + 1
        if large ** 2 > n:
            return small
        else:
            return large

def SquareInt(s):
    """
    Acts as a "parser" for perfect square integers for argparse
    """
    n = int(s)
    if isqrt(n) ** 2 != n:
        raise ValueError("{!r} is not a square number".format(s))
    return n

def SquareBoard(board):
    """
    Acts as a "parser" for strings representing square boards, similar to
    SquareInt. Ignores all non-interesting characters and demands squareness.
    """
    b = [state_from_string[c] for c in board if c in state_from_string]
    if isqrt(len(b)) ** 2 != len(b):
        raise ValueError('The board must be square')
    return b

def get_pos(s, n):
```

```
50      """
51      Get position in 1d list from 2d coordinate reference
52      """
53      x, y = map(int, s.split())
54      if not all(0 <= c < n for c in (x, y)):
55          raise ValueError("Not in range [0,{})".format(n))
56
57      return y * n + x
58
59  def get_input(board, is_crosses, n):
60      """
61      Get user input of where to play on a board.
62      """
63      print("You are playing as {}".format(name_from_bool[is_crosses]))
64      while True:
65          try:
66              mov = get_pos(input("Enter the position you want to play in > "), n)
67              if board[mov] is not None:
68                  raise ValueError("This position is already taken")
69          except ValueError as ve:
70              print(ve)
71              continue
72          return mov
73
74  def do_player_move(board, is_crosses, n):
75      """
76      Execute player move - assumes board is valid at start of turn.
77      """
78      print_board(board, n)
79      try:
80          pos = get_input(board, is_crosses, n)
81      except (KeyboardInterrupt, EOFError):
82          raise Win("\n{} wins because {} is a coward"
83                    .format(syms[not is_crosses], syms[is_crosses]))
84      board[pos] = is_crosses
85      if is_run(board, pos, n):
86          raise Win("{} wins".format(syms[is_crosses]))
```

Listing 8: `interface.py`: Dealing with user input

Listing 9 ties it all together, providing a pretty sophisticated interface through command-line arguments (see the `get_args` function). It allows you to play against yourself, the computer, or even just for you to watch the computer instantly force itself into a draw, if that's your thing.

The only other application of the `--battle` mode is to change the size of the board to anything more than 3, and watch your CPU melt. This is due to the size of the search tree growing with $O(2^{(n^2)})$, which, however, nicely implemented your tree search is, is a bit of a party-pooper.

A sample of play is provided in listing 10

```
1   """
2   Play noughts and crosses. Incorporates both human and computer players.
```

```
3  """
4
5  from argparse import ArgumentParser
6  from itertools import cycle, repeat
7
8  from base import GameFinish, Draw
9  from interface import do_player_move
10 from computer import do_computer_move as _do_computer_move
11
12 def get_args():
13     """
14     Get configuration for the program. See the help text for details.
15     """
16     parser = ArgumentParser(description=__doc__)
17     mode = parser.add_mutually_exclusive_group()
18     mode.add_argument("-c", "--computer", action="store_true",
19                       help="play against computer opponent")
20     mode.add_argument("-b", "--battle", action="store_true",
21                       help="computer plays against itself")
22     parser.add_argument("--headstart", action="store_true",
23                       help="start first when playing against computer")
24     parser.add_argument("--noughts-start", action="store_true",
25                       help="noughts to start instead of crosses")
26     parser.add_argument("-s", "--size", type=int, default=3,
27                       help="size of board to play on")
28     parser.add_argument("-v", "--verbose", action="store_true",
29                       help="Show minmax thought process")
30     return parser.parse_args()
31
32 def play(board, players, noughts_start, n):
33     """
34     Play a game of noughts and crosses until a finishing condition or a draw,
35     given an infinite iterable of players.
36     """
37     is_crosses = not noughts_start
38     try:
39         for player in players:
40             player(board, is_crosses, n)
41             is_crosses = not is_crosses
42             if board.count(None) == 0:
43                 raise Draw("Nobody wins!")
44     except GameFinish as gf:
45         print("{}: {}".format(type(gf).__name__, gf))
46
47 if __name__ == "__main__":
48     args = get_args()
49     # Correctly initialise the infinite iterable of players according to
50     # arguments
51     vb = args.verbose
52     do_computer_move = lambda *args: _do_computer_move(*args, verbose=vb)
53     if args.computer:
```

```
54          if args.headstart:
55              players = cycle([do_player_move, do_computer_move])
56          else:
57              players = cycle([do_computer_move, do_player_move])
58      elif args.battle:
59          players = repeat(do_computer_move)
60      else:
61          players = repeat(do_player_move)
62      play([None] * args.size ** 2, players, args.noughts_start, args.size)
```

Listing 9: `play.py`: Bringing it all together to play the game

```
Computer plays at (0, 0)
   0   1   2
 0 X |   |
  ---+---+---
 1   |   |
  ---+---+---
 2   |   |

   0   1   2
 0 X |   |
  ---+---+---
 1   |   |
  ---+---+---
 2   |   |

You are playing as noughts
Enter the position you want to play in > 1 0
Computer plays at (0, 1)
   0   1   2
 0 X | O |
  ---+---+---
 1 X |   |
  ---+---+---
 2   |   |

   0   1   2
 0 X | O |
  ---+---+---
 1 X |   |
  ---+---+---
 2   |   |

You are playing as noughts
Enter the position you want to play in > 0 2
Computer plays at (1, 1)
   0   1   2
 0 X | O |
  ---+---+---
 1 X | X |
```

```
   ---+---+---
  2 O |   |

    0   1   2
  0 X | O |
   ---+---+---
  1 X | X |
   ---+---+---
  2 O |   |

You are playing as noughts
Enter the position you want to play in > 2 2
Computer plays at (2, 1)
    0   1   2
  0 X | O |
   ---+---+---
  1 X | X | X
   ---+---+---
  2 O |   | O

Win: I'm sorry, Dave. I'm afraid I can't do that.
```

Listing 10: Output of `play.py`

## 3 Interface

Also provided is an elegant (bare) interface written in Processing, for ease of translation of Python and graphics primitives. It reuses most of the same code, stripping out parts that print, and involved a slight rewrite of `generate_moves`, due to Python 2's alternative handling of finally clauses. A diff of the library code adapted for graphical Python 2 is shown in listing 11.

This was written with very little time to spare, so excuse my slightly square crosses and sloppy code here.

```
$ for i in *.py; do echo $i; diff $i src/$i; done
base.py
4a5,6
> from enum import Enum
>
23c25
< class State(object):
---
> class State(Enum):
checking.py
7a8
> from formatting import get_board_template
81a83
>             print(board_temp.format(*dft), end="\n\n")
computer.py
5a6,8
> from textwrap import indent
```

```
> from traceback import extract_stack
>
7a11
> from formatting import print_board, strfboard, syms
72a77,80
>     verbose and print(indent("Examining as {} {}:\n{}"
>                              .format(syms[crosses_playing], depth,
>                                 indent(strfboard(board, n), ' ')),
>                          ' ' * len(extract_stack())))
74a83,84
>         verbose and print(indent("State here: {}"
>                   .format(state), " " * len(extract_stack())))
76a87
>         verbose and print(indent("Draw here", " " * len(extract_stack())))
82c93
<                     for move, board in generate_moves(board[:], crosses_playing)),
---
>                     for move, board in generate_moves(board, crosses_playing)),
103a115
>             verbose and print("Win incoming")
105a118
>             verbose and print("Draw forcable")
117a131,132
>     print("Computer plays at ({}, {})".format(move % n, move // n))
>     print_board(board, n)
interface.py
5a6
> from formatting import print_board, syms
```

Listing 11: Diff of processing code vs core src

The drawing code is shown in listing 12

```
from random import choice

from computer import get_computer_move
from checking import is_run

DRAW_WIDTH = 0.8

def draw_board(board, n, w):
    pushMatrix()
    scale(w)
    for ind, tile in enumerate(board):
        if tile is not None:
            x, y = ind % n, ind // n
            if tile:
                rect(x + 0.5, y + 0.5, DRAW_WIDTH, DRAW_WIDTH)
            else:
                ellipse(x + 0.5, y + 0.5, DRAW_WIDTH, DRAW_WIDTH)
    popMatrix()
```

```
19
20  def setup():
21      global board, gameover
22      size(800, 800)
23      fill(255)
24      rectMode(CENTER)
25      noStroke()
26      background(0)
27      board = [None] * 9
28
29      gameover = None
30
31      cmov = get_computer_move(board, True, 3)
32      board[cmov] = True
33
34
35  def draw():
36      global board
37      if gameover is None:
38          background(0)
39      elif gameover == True:
40          background(255, 0, 0)
41      else:
42          background(255, 255, 0)
43      draw_board(board, 3, width / 3.0)
44
45  def mouseClicked():
46      global gameover
47      if not gameover:
48          x, y = int(3 * mouseX / width), int(3 * mouseY / height)
49          pos = 3 * y + x
50          if board[pos] is None:
51              print("click")
52              board[pos] = False
53              cmov = get_computer_move(board, True, 3)
54              board[cmov] = True
55              if is_run(board, cmov, 3):
56                  gameover = True
57              elif board.count(None) == 0:
58                  gameover = False
59
60  def keyPressed():
61      if keyCode == ord("R"):
62          setup()
```

Listing 12: Processing code to handle user interaction and drawing of board

The game's modern design is shown in figure 1

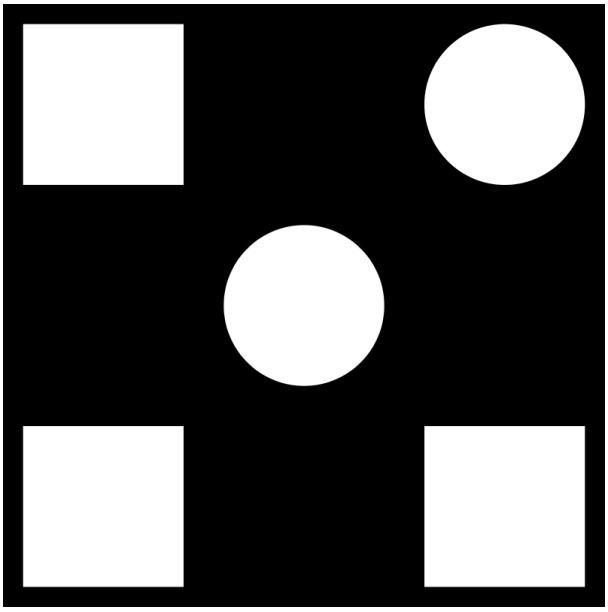

Figure 1: The new way to play noughts and crosses

## 4 Source

The full project in its directory structure, including this document (as a full-colour PDF and TEX file), can be found at `https://github.com/elterminad0r/noughtsandcrosses`.