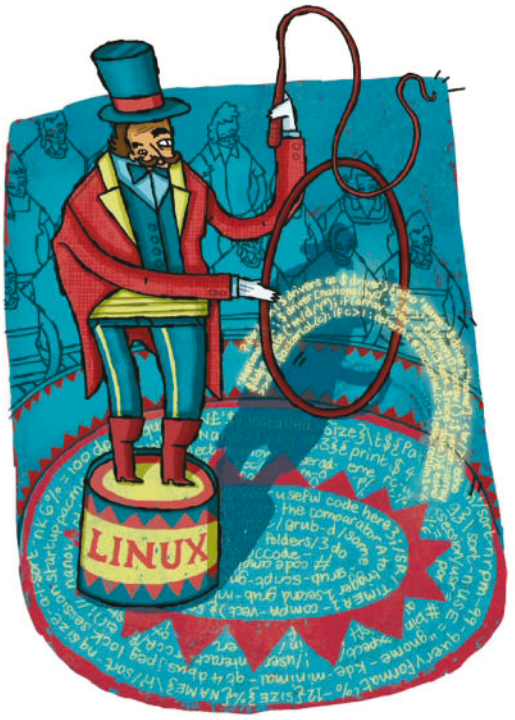
Python: Sunfish chess engine

Jonni Bidwell analyses the innards of a small but perfectly formed chess engine that bests him with alarming regularity.

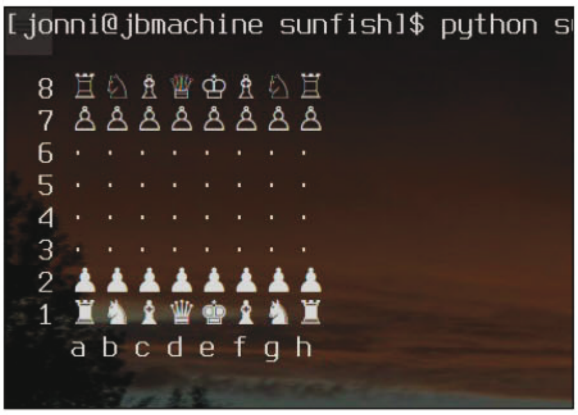
Our expert

Jonni Bidwell is rumoured to be a mechanical Turk, it would explain the rat-a-tat of gears as he produces words in exchange for bread and beer.



Legend tells of one Sissa ibn Dahir who invented the game of Chess for an Indian king. So impressed was that king that he offered Sissa anything he desired as a reward. Being of a calculating bent, Sissa replied "Then I wish that one grain of wheat shall be put on the first square of the chessboard, two on the second, and that the number of grains shall be doubled until the last square is reached: whatever the quantity this might be, I desire to receive it". The king soon realised that there was not enough wheat in the world to fulfil this demand, and once again was impressed. There are various endings to this story, in one Sissa is given a position within the king's court, in another he is executed for being a smart arse. Hopefully this tutorial's chess treatment will feature neither execution nor **LXF** towers being buried in mountains of wheat.

Chess is a complicated game – all the pieces move differently depending on their circumstances, there are various extraordinary moves (eg en passent pawn capture, castling) and pawns get promoted if they make it all the way to the other side. As a result, a surfeit of pitfalls present themselves to the chess-programming dilettante, so rather than spending a whole tutorial falling into traps we're going to borrow the code from Thomas Ahle's Sunfish – a complete chess engine programmed in Python. There's no shortage of chess engines: from the classic GNU Chess to the Kasparov-beating Deep Blue (1997) to the pack-leading Stockfish. Chess engines on their own generally do not come with their own GUI, their code being mostly devoted to the not inconsiderable problem of finding the best move for a given position. Some (including Sunfish) allow you to play via a text console, but most will talk to an external GUI, such as *xboard*, via a protocol such as the Universal Chess Interface (UCI) or WinBoard. Besides providing nice pictures of the action, this enables us to play against different engines from a single program. Furthermore, we can pit engine against engine and enjoy chess as a spectator sport.



```
[jonni@jbmachine sunfish]$ python s

 8 ♖ ♘ ♗ ♕ ♔ ♗ ♘ ♖
 7 ♙ ♙ ♙ ♙ ♙ ♙ ♙ ♙
 6 · · · · · · · ·
 5 · · · · · · · ·
 4 · · · · · · · ·
 3 · · · · · · · ·
 2 ♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟
 1 ♜ ♞ ♝ ♛ ♚ ♝ ♞ ♜
   a b c d e f g h
```

› Unicode generously provides chess piece icons which enhance the experience of playing from the terminal.

The Sunfish engine

We'll assume that you know how to play chess, but if you don't you can practice by playing against Thomas's Sunfish engine. You'll find the code on the **LXFDVD** in the **Tutorials/Chess** directory. Copy this directory to your **home** folder, and then run it with:

```
$ cd ~/Chess
$ python sunfish.py
```

The program uses Unicode glyphs to display the pieces in the terminal, making it look a little more chess-like than *GNU Chess*. Check the box (*see Installing Xbound and Interfacing with Sunfish*) to see how to enjoy graphical play. Moves are inputted by specifying the starting and ending coordinates, so the aggressive opening which moves the king's pawn to e4 would be inputted e2e4. Note that this is slightly longer than the more common algebraic notation (in which the previous move would be written `e4`), but makes it much easier for

The Mechanical Turk and other chess-playing machines

In 1770 Baron Wolfgang von Kempelen wowed the Viennese court with 'The Turk', a clockwork automaton sat before a chessboard. Kempelen claimed that his invention would best any human chess player. Indeed, the Baron and the Turk travelled around Europe and wowed onlookers with the latter's prodigious talent.

The Turk was a hoax, and its talent actually belonged to the poor person hiding under the table. However, it inspired people to think more about chess playing machines, and in 1950 Shannon and Turing both published papers on the subject. By the 1960s computers were playing reasonable chess: John McCarthy (dubbed the father of AI) and Alan Kotok at MIT developed a program that would best most beginners. This program, running on an IBM 7090, played a correspondence match via telegraph against an M-2 machine run by Alexander Kronrod's team at ITEP in Moscow.

This was the first machine versus machine match in history, and the Soviets won 3-1. Their program evolved into KAISSA, after the goddess of chess, which became the computer chess champion in 1974. By the early 80s the chess community began to speculate that sooner or later a computer would defeat a world champion. Indeed, in 1988 IBM's Deep Thought shared first place at the US Open, though reigning world champion Garry Kasparov resoundingly defeated it the following year. In 1996 Deep Blue stunned the world by winning its first game against Kasparov, although the reigning world champion went on to win the match 4-2. The machine was upgraded and succeeded in beating Kasparov the following year, though not without controversy. Since then computers regularly beat their inferior meatbag competition, although their prowess is driven by algorithmic advances.

machines to understand what you mean. If you wish to castle then just specify that you want to move your king two places sideways, the machine knows the rules and will move the relevant rook as well, provided that castling is a legal move at that stage in the game. Depending on your skills you will win, lose or draw.

In **LXF203** we used PyGame to implement the ancient board game Gomoku. For this tutorial we'll see a slightly different approach. Have a look at the **sunfish.py** code: the shebang directive in the first line specifies that **sunfish.py** should be run with the Pypy compiler, rather than the standard Python interpreter. Installation of Pypy is trivial and will improve Sunfish's search-performance drastically, but for our purposes it will be fine to proceed without it. We import the `print_function` syntax for backwards compatibility with Python 2, as well as the needed parts of other modules. Then we initialise three global variables, which we needn't worry about here.

Chairman of the board

Now we begin to describe our chessboard. Its starting state is stored as a 120-character string, `initial`, which may seem a little odd, especially if you remember how nice it was to store the GoMoku board as a two-dimensional list. Be that as it may, this representation turns out to be much more efficient. Before defining `initial` we specify what will be the indices of the corner squares using the standard layout, so A1 is the lower left corner and A8 the lower right etc. We divide the string into rows of 10 characters, remembering that the newline `\n` counts as a single character. The actual board starts on the third row, where we represent black's major pieces with the standard lowercase abbreviations, which we'll list below for completeness:

- p: Pawn
- r: Rook
- n: Knight
- b: Bishop
- q: Queen
- k: King

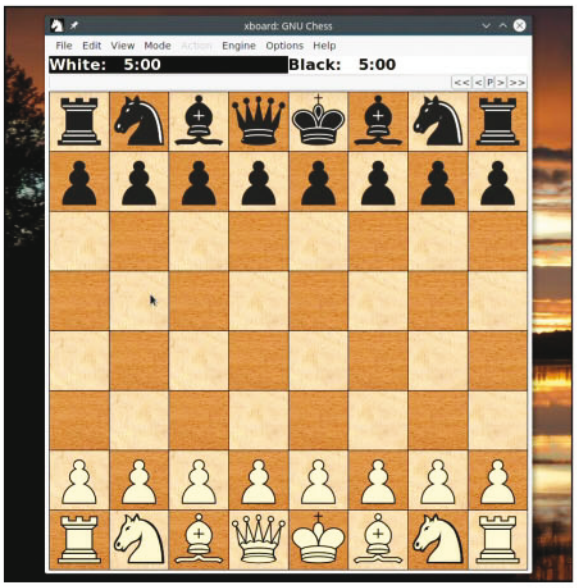
We have characters padding the beginning (a space) and the end (`\n`) of each row so we know that moving a piece one square vertically will involve adding or subtracting 10 from its index in the string. Dually, moving one square along the horizontal axis will be done by adding or subtracting 1, and we know that if the resulting index ends with a 0 (ie is 0 modulo 10) or 9 (ie is 9 modulo 10) then that position is not on the board. The vertical ranks 1-9 can also be read directly from the second digit of the index, and the horizontal rows can be translated linearly from the first. Empty spaces on the board are represented by periods (`.`) to avoid confusion with the empty squares represented by spaces.

Using the numerology (above) we describe unit movements in the compass directions with appropriately named variables, and then define the possible movements of each piece in the dictionary `directions`. Note that we only define the movements for white's material here (ie pawns go north), their opponents can be figured by a simple transposition. Note also that we describe all the possible directions they can move, even though this may not be permitted by the current position (eg pawns can only move diagonally when they are taking and can only move two squares on their first move. We don't take account of major pieces moving two or more squares in a straight line ('sliding') here, rather dealing with that instead in the move generation loop. Next, we define a lengthy dictionary `pst`. In a sense this is the data bank of the engine, it assigns a value to each piece for a given position on the board, so, eg, knights (`N`) tend to be more useful towards the centre of the board, whereas the queen is valueable anywhere. The king's values are

»

Quick tip

Sunfish used to be limited by the lack of a quiescence search. This meant that moves at the depth limit were not analysed, which can lead to so-called horizon effects, in that the engine can't see past blunders here. Thanks to a simple check, moves at this limit are analysed to ensure they result in quiescent positions.



› This is how every chess game starts, but after just four moves we could be in one of nearly 320 million different positions.

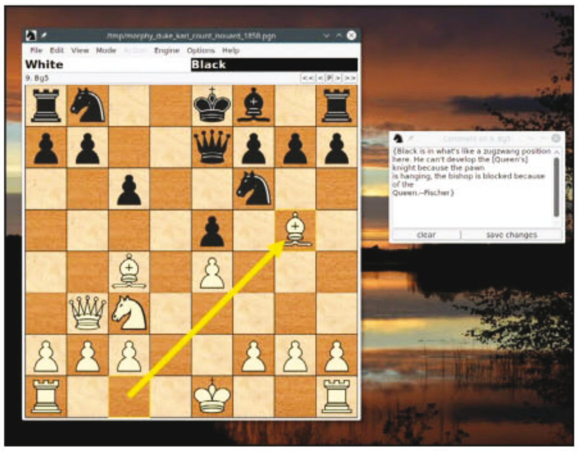
» disproportionately high so that the machine knows it can never be sacrificed.

Now we move on to the chess logic section and subclass the `namedtuple` construction to describe a given chess position. Using this datatype enables us to have a tuple (a fixed-length list) with named keys rather than numerical indices. We store the current `board` arrangement together with the evaluation `score` for that position. Then we have four extra elements to take care of the exceptional moves – castling and en passant pawn capture. The `gen_moves` function iterates over each square on the board and every possible move for each piece on the board. The loop is commenced as follows:

```
    for i, p in enumerate(self.board):
        if not p.isupper(): continue
        for d in directions[p]:
            for j in count(i+d, d):
                q = self.board[j]
                if self.board[j].isspace(): break
```

The `enumerate()` function (line 147) is a vital weapon in any Pythonista's arsenal, as it generates index-value pairs for a given list (or string in our case), useful when we are interested in list items' positions as well as their content. Because of the symmetry involved we only consider the moves of white's pieces, so we bail out if the relevant piece `p` in the string is not uppercase (line 148).

Fortuitously, the `.isupper` method also returns `False` for spaces and periods, so empty squares are efficiently thrown away early on in the proceedings. The `rotate()` function transposes the colours when it's black's turn. We look at all possible directions that the piece can move and then (line 150) extend these moves to account for those pieces allowed to slide (ie Rooks, Bishops and Queens). Finally, we discard any move that takes us off the board.



/tmp/morphy_duke_karl_count_isouard_1858.pgn
File Edit View Mode Action Engine Options Help
White
Black
9. Bg5
clear
save changes

{Black is in what's like a zugzwang position here. He can't develop the [Queen's] knight because the pawn is hanging, the bishop is blocked because of the Queen.--Fischer}

› This is from a game Paul Morphy (white) played against the Duke of Brunsick and Count Isouard in 1858. It's a so-called zugzwang for black (to move) – most moves are detrimental.

The next part of the function checks if castling is possible:

```
                if i == A1 and q == 'K' and self.wc[0]: yield (j, j-2)
                if i == H1 and q == 'K' and self.wc[1]: yield (j, j+2)
                if q.isupper(): break
```

Castling rights for both rooks are stored as a booleans in the list `wc`, which is part of our `Position` object. If we are considering either of white's corner squares and if white still has castling rights (so those squares are certainly occupied by rooks) then we `yield` the move which moves the king two spaces left or right. Our `gen_moves()` is what is an example of a generator function – it yields results which can be used in `for` or `while` loops. In our case, we generate a pair of indices – the pieces position and after the move. Finally, we break if the destination square is occupied by one of white's pieces, since friendly captures aren't allowed.

A pawn in the game

Next we consider matters peculiar to pawns:

```
                if p == 'P' and d in (N+W, N+E) and q == '.' and j not in (self.ep, self.kp): break
                if p == 'P' and d in (N, 2*N) and q != '.': break
                if p == 'P' and d == 2*N and (i < A1+N or self.board[i+N] != '.'): break
```

First, pawns cannot move diagonally into an empty square, unless an en passant capture can take place. Next, they can only move forwards (one or two squares, we check if the latter is allowed in the next line). The `i < A1 + N` comparison will return true for any pawn that has moved beyond the second row. Two-square rights are likewise denied if there's a piece in front of the pawn.

The closing stanza of `gen_moves()` reads:

```
                yield (i, j)
                if p in ('P', 'N', 'K'): break
                if q.islower(): break
```

Having got all the constraints, we can now pass on the move under consideration, it may yet turn out not to be valid (eg if it doesn't alleviate a check situation) but it's passed the first level of filtration. Pawns, knights and rooks aren't allowed to slide, and those pieces that are have to stop doing so if they capture a piece (ie land on a lowercase entry in `board`).

We've already discussed the `rotate()` function. But impressively the board can be transposed (which results in the equivalent game with the colours switched and the board rotated 180 degrees) just by reversing the `board` string and switching cases. We must take care of the other parts of our

Installing Xboard and interfacing with Sunfish

There are a number of good chess graphical user interfaces for Linux, we're using *Xboard* as it's fairly ubiquitous amongst common distro repos, but be sure to check out PyChess as well. Installation will just be a matter of:

```
$ sudo apt-get install xboard
```

Now fire up *Xboard* and select Engine >Load New 1st Engine. Enter `Sunfish` in the Nickname field, for the Engine Directory use **/home/user/Chess** (replacing **user** with your username – *Xboard* doesn't seem to understand the `~` shorthand) and for the Engine Command use `python /home/user/Chess/xboard.py`. Leave all the other settings as they are and select 'OK'.

The default *XBoard* setup gives the human player white pieces and if all has gone well the window title should now read 'Sunfish'. Now you have your formidable opponent.

If you click one of your chess pieces then *Xboard* generously shows you where the piece can move to, which is not just useful for players who are starting out.

GNU Chess will be equally as trivial to install, and *Xboard* already includes an Engine List entry for it. You can load *GNU Chess* as the second engine, and then select Two Machines from the Mode menu. Battle ought immediately to commence, with *GNU Chess* even showing you some of its crazy thought process in the status bar.

object though. Specifically, we need the negative of the position's score, since we are still in opposition to the previous player, even though we're pretending to have adopted their colour. Castling rights are already separated, so they need no further treatment. The en passant positions are easily figured out by counting backwards from the end of the board.

The function `move()` deals with actually moving the pieces, when that time comes. We first get our beginning and end positions `i` and `j` and the occupants of those squares `p` and `q`. Line 178 defines a shorthand (lambda) function which replaces the piece at position `i` with the piece we are moving. We get and reset required class variables, to save us from typing `self` many times, and update the score by calling the valuation function. In line 184 we place the piece in its new position using our lambda function and then remove the piece from its original position with a further call.

Beginning at line 187, we update the castling rights: if a rook is moved then castling on that side is no longer allowed, the value for the other, stationary, rook is preserved. Castling itself is instigated by the king:

```
    if p == 'K':
        wc = (False, False)
        if abs(j-i) == 2:
            kp = (i+j)//2
            board = put(board, A1 if j < i else H1, '.')
            board = put(board, kp, 'R')
```

Once he has been moved castling rights are cancelled, regardless of whether the player intends to castle or not. If they do then they are moving the king two places sideways, with the rook on that side ending up on the square horizontally adjacent to him. This is calculated by rounding down the midpoint of positions `i` and `j`. We use some more shorthand to delete the rook's old position and the final line puts it in its new one.

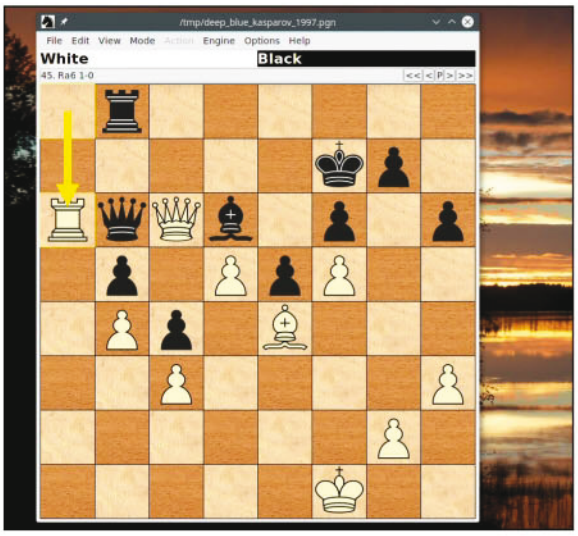
Manipulating the pawns

Next we deal with pawns. Sunfish doesn't do minor promotion, ie pawns are only promoted to queens if they reach row 8 (line 201). If a pawn moves two squares then we keep track of the square behind, in case an en passant capture is possible. If the pawn makes an en passant capture then the appropriate square is obliterated. We return a new Position object, remembering to rotate it to account for the next player's point of view.

The valuation function `value()` calculates the relative value of a given move. We start by calculating the difference between the value piece's positions before and after the move. If the player has captured a piece then that piece's value at its capture position is added. If castling results in the king being checked, then the value will go sky high (precluding that move). If castling did take place, then the value needs to be adjusted according to the rook's new position. Finally, we account for pawn promotion and en passant capture.

We'll give an overview of the search logic section at the end, but it's worth having a brief look at the user interface section (which starts at line 338). The `parse()` function converts from a two digit co-ordinate string (such as `a4`) to the relevant list index (61 in this case). The `render()` function does the opposite. The `print_pos()` function nicely prints the board, complete with Unicode characters in lieu of actual graphics and labelled axes for the ranks and files.

The final function `main()` sets up the initial layout and defines the main game loop. Each iteration starts by displaying the board and asking for a move. We use the regular expression `'([a-h][1-8])'*2` (line 369) to check that the move is of the correct form, ie a pair of co-ordinates. If it is, and that move is legal for the current position (it's generated by `pos.gen_moves()`) then we proceed, otherwise we ask again. Then we reverse the board for the computers turn and use the engine's `search` function to find the best move. If this move results in a checkmate, or fails to resolve a checkmate then the game is done, otherwise the move is printed and the board updated.


/tmp/deep_blue_kasparov_1997.pgn
File Edit View Mode Action Engine Options Help
White
Black
45. Ra6 1-0

› Deep Blue v Kasparov (1997 - Round 2). Kasparov resigned after the machine shocked him with this move, throwing his performance for the rest of the match. It turns out he could have forced a draw from here, d'oh.

The code we've discussed so far can easily be adapted to a two-player game like we saw in the Gomoku tutorial. However, what is much more interesting is the code that figures out the machine's next move. Amazingly, the engine itself (ignoring the lengthy `pst` dictionary and all the code we've already covered) occupies less than a hundred lines.

Sunfish is based on the MTD(f) minimax search algorithm introduced in 1994, adapted to use binary trees. MTD uses so-called alpha-beta pruning for evaluating the game tree, so we build up a tree of possible moves and discard any which we can show lead to positions that are provably worse off than others we have evaluated. A technique called iterative deepening is used to temper the depth of the search, so that we don't go too far down one particular rabbit hole. Calculating a move begins with a call to the `search()` function. We limit both the depth (line 305) and the breadth (initially using the `NODES_SEARCHED` variable) of the search to stop things getting out of control. The real magic happens in the `bound()` function.

The move tree is stored in the ordered dictionary `tp`, which is indexed by our position string `pos` and so previously calculated positions can be looked up efficiently. When we come to analysing all possible moves (line 270), we sort the generated positions in reverse order by their value, which ensures copacetic positions get the attention they deserve. We use `bound()` recursively to construct a game tree from each possible move, adding appropriate moves to `tp` (line 293). Alas, the end of the page approaches so here is where we sign off.

You'll find some helpful comments to aid your understanding of the engine code, so why not experiment by tweaking parameters and seeing what breaks? If you want to learn more about the intricacies of programming chess, make sure and check out the Chess Programming wiki (**https://chessprogramming.wikispaces.com**), it will prove a valuable resource. LXF