python-chess

Release 0.3.0

August 15, 2014

Contents

CHAPTER 1

Introduction

This is the scholars mate in python-chess:

```
>>> import chess

>>> board = chess.Bitboard()

>>> board.push_san("e4")
Move.from_uci('e2e4')
>>> board.push_san("e5")
Move.from_uci('e7e5')
>>> board.push_san("Qh5")
Move.from_uci('d1h5')
>>> board.push_san("Nc6")
Move.from_uci('b8c6')
>>> board.push_san("Bc4")
Move.from_uci('f1c4')
>>> board.push_san("Nf6")
Move.from_uci('g8f6')
>>> board.push_san("Qxf7")
Move.from_uci('h5f7')

>>> board.is_checkmate()
True
```

CHAPTER 2

Documentation

https://python-chess.readthedocs.org/en/latest/

CHAPTER 3

Features

- Supports Python 2.7 and Python 3.
- Legal move generator and move validation. This includes all castling rules and en-passant captures.

```
>>> chess.Move.from_uci("a8a1") in board.legal_moves
False
```

- Make and unmake moves.

```
>>> Qf7 = board.pop() # Unmake last move (Qf7#)
>>> Qf7
Move.from_uci('h5f7')

>>> board.push(Qf7) # Restore
```

- Detects checkmates, stalemates and draws by insufficient material. Has a half-move clock.

```
>>> board.is_stalemate()
False
>>> board.is_insufficient_material()
False
>>> board.is_game_over()
True
>>> board.halfmove_clock
0
```

- Detects checks and attacks.

```
>>> board.is_check()
True
>>> board.is_attacked_by(chess.WHITE, chess.E8)
True

>>> attackers = board.attackers(chess.WHITE, chess.F3)
>>> attackers
SquareSet(0b100000001000000)
>>> chess.G2 in attackers
True
```

- Parses and creates SAN representation of moves.

```
>>> board = chess.Bitboard()
>>> board.san(chess.Move(chess.E2, chess.E4))
'e4'
```

- Parses and creates FENs.

```
>>> board.fen()
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1'
>>> board = chess.Bitboard("8/8/8/2k5/4K3/8/8/8 w - - 4 45")
>>> board.piece_at(chess.C5)
Piece.from_symbol('k')
```

- Parses and creates EPDs.

```
>>> board = chess.Bitboard()
>>> board.epd(bm=chess.Move.from_uci("d2d4"))
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - bm d4;'

>>> ops = board.set_epd("1k1r4/pp1b1R2/3q2pp/4p3/2B5/4Q3/PPP2B2/2K5 b - - bm Qd1+; id \"BK.01\";
>>> ops == {'bm': chess.Move.from_uci('d6d1'), 'id': 'BK.01'}
True
```

- Read Polyglot opening books.

```
>>> import chess.polyglot

>>> book = chess.polyglot.open_reader("data/opening-books/performance.bin")
>>> board = chess.Bitboard()
>>> first_entry = next(book.get_entries_for_position(board))
>>> first_entry.move()
Move.from_uci('e2e4')
>>> first_entry.learn
0
>>> first_entry.weight
1

>>> book.close()
```

- Read and write PGNs. Supports headers, comments, NAGs and a tree of variations.

```
>>> import chess.pgn

>>> from __future__ import print_function # Python 2 compability for
>>>                                       # this example.

>>> pgn = open("data/games/molinari-bordais-1979.pgn")
>>> first_game = chess.pgn.read_game(pgn)
>>> pgn.close()

>>> first_game.headers["White"]
'Molinari'
>>> first_game.headers["Black"]
'Bordais'

>>> # Iterate through the mainline of this embarrasingly short game.
>>> node = first_game
>>> while node.variations:
...     next_node = node.variation(0)
...     print(node.board().san(next_node.move))
...     node = next_node
e4
c5
c4
Nc6
```

```
Ne2
Nf6
Nbc3
Nb4
g3
Nd3#

>>> first_game.headers["Result"]
'0-1'
```

CHAPTER 4

Peformance

python-chess is not intended to be used by serious chess engines where performance is critical. The goal is rather to create a simple and relatively highlevel library.

However, even though bit fiddling in Python is not as fast as in C or C++, the current version is still much faster than previous attempts including the naive x88 move generation from libchess.

CHAPTER 5

Installing

- With pip:

  ```
  sudo pip install python-chess
  ```

- From current source code:

  ```
  python setup.py build
  sudo python setup.py install
  ```

License

python-chess is licensed under the GPL3. See the LICENSE file for the full copyright and license information.

Thanks to the developers of http://chessx.sourceforge.net/. Some of the core bitboard move generation parts are ported from there.

CHAPTER 7

Contents

7.1 Changelog for python-chess

This project is pretty young and maturing only slowly. At the current stage it is more important to get things right, than to be consistent with previous versions. Use this changelog to see what changed in a new release, because this might include API breaking changes.

7.1.1 New in v0.3.0

- Rename property *half_moves* of *Bitboard* to *halfmove_clock*.
- Rename property *ply* of *Bitboard* to *fullmove_number*.
- Let PGN parser handle symbols like *!*, *?*, *!?* and so on by converting them to NAGs.
- Add a human readable string representation for Bitboards.

```
>>> print(chess.Bitboard())
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
```

- Various documentation improvements.

7.1.2 New in v0.2.0

- Implement PGN parsing and writing.
- Hugely improve test coverage and use Travis CI for continuous integration and testing.
- Create an API documentation.
- Improve Polyglot opening-book handling.

7.1.3 New in v0.1.0

Apply the lessons learned from the previous releases, redesign the API and implement it in pure Python.

7.1.4 New in v0.0.4

Implement the basics in C++ and provide bindings for Python. Obviously performance was a lot better - but at the expense of having to compile code for the target platform.

7.1.5 Pre v0.0.4

First experiments with a way too slow pure Python API, creating way too many objects for basic operations.

7.2 Core

7.2.1 Colors

Constants for the side to move or the color of a piece.

`chess.`**`WHITE`** = **0**

`chess.`**`BLACK`** = **1**

You can get the opposite color using *color* ^ *1*.

7.2.2 Piece types

`chess.`**`NONE`** = **0**

`chess.`**`PAWN`**

`chess.`**`KNIGHT`**

`chess.`**`BISHOP`**

`chess.`**`ROOK`**

`chess.`**`QUEEN`**

`chess.`**`KING`**

7.2.3 Castling rights

The castling flags

`chess.`**`CASTLING_NONE`** = **0**

`chess.`**`CASTLING_WHITE_KINGSIDE`**

`chess.`**`CASTLING_BLACK_KINGSIDE`**

`chess.`**`CASTLING_WHITE_QUEENSIDE`**

`chess.`**`CASTLING_BLACK_QUEENSIDE`**

can be combined bitwise.

`chess.`**`CASTLING_WHITE`** = **CASTLING_WHITE_QUEENSIDE | CASTLING_WHITE_KINGSIDE**

`chess.`**`CASTLING_BLACK`** = **CASTLING_BLACK_QUEENSIDE | CASTLING_BLACK_KINGSIDE**

`chess.`**`CASTLING`** = **CASTLING_WHITE | CASTLING_BLACK**

7.2.4 Squares

`chess.`**`A1`** = **0**

`chess.`**`B1`** = **1**

and so on to

`chess.`**`H8`** = **63**

`chess.`**`SQUARES`** = **[A1, B1, ..., G8, H8]**

`chess.`**`SQUARE_NAMES`** = **['a1', 'b1', ..., 'g8', 'h8']**

`chess.`**`file_index`**(*square*)
Gets the file index of square where *0* is the a file.

`chess.`**`FILE_NAMES`** = **['a', 'b', ..., 'g', 'h']**

`chess.`**`rank_index`**(*square*)
Gets the rank index of the square where *0* is the first rank.

7.2.5 Pieces

class `chess.`**`Piece`**(*piece_type*, *color*)
A piece with type and color.

`piece_type`
The piece type.

`color`
The piece color.

`symbol`()
Gets the symbol *P*, *N*, *B*, *R*, *Q* or *K* for white pieces or the lower-case variants for the black pieces.

classmethod **`from_symbol`**(*symbol*)
Creates a piece instance from a piece symbol.

Raises *ValueError* if the symbol is invalid.

7.2.6 Moves

class `chess.`**`Move`**(*from_square*, *to_square*, *promotion=0*)
Represents a move from a square to a square and possibly the promotion piece type.

Castling moves are identified only by the movement of the king.

Null moves are supported.

`from_square`
The source square.

`to_square`
The target square.

promotion
The promotion piece type.

uci()
Gets an UCI string for the move.

For example a move from A7 to A8 would be *a7a8* or *a7a8q* if it is a promotion to a queen. The UCI representatin of null moves is *0000*.

classmethod from_uci(*uci*)
Parses an UCI string.

Raises *ValueError* if the UCI string is invalid.

classmethod null()
Gets a null move.

A null move just passes the turn to the other side (and possibly forfeits en-passant capturing). Null moves evaluate to *False* in boolean contexts.

```
>>> bool(chess.Move.null())
False
```

7.2.7 Bitboard

chess.**STARTING_FEN = 'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1'**
The FEN notation of the standard chess starting position.

class chess.**Bitboard**(*fen=None*)
A bitboard and additional information representing a position.

Provides move generation, validation, parsing, attack generation, game end detection, move counters and the capability to make and unmake moves.

The bitboard is initialized to the starting position, unless otherwise specified in the optional *fen* argument.

turn
The side to move.

castling_rights
Bitmask of castling rights.

ep_square
The potential en-passant square on the third or sixth rank or *0*. It does not matter if en-passant would actually be possible on the next move.

fullmove_number
Counts move pairs. Starts at *1* and is incremented after every move of the black side.

halfmove_clock
The number of half moves since the last capture or pawn move.

pseudo_legal_moves = PseudoLegalMoveGenerator(self)
A dynamic list of pseudo legal moves.

Pseudo legal moves might leave or put the king in check, but are otherwise valid. Null moves are not pseudo legal. Castling moves are only included if they are completely legal.

For performance moves are generated on the fly and only when nescessary. The following operations do not just generate everything but map to more efficient methods.

```
>>> len(board.pseudo_legal_moves)
20

>>> bool(board.pseudo_legal_moves)
True

>>> move in board.pseudo_legal_moves
True
```

`legal_moves` = LegalMoveGenerator(self)
A dynamic list of completely legal moves, much like the pseudo legal move list.

`reset`()
Restores the starting position.

`clear`()
Clears the board.

Resets move stacks and move counters. The side to move is white. There are no rooks or kings, so castling is not allowed.

In order to be in a valid *status()* at least kings need to be put on the board. This is required for move generation and validation to work properly.

`piece_at`(*square*)
Gets the piece at the given square.

`piece_type_at`(*square*)
Gets the piece type at the given square.

`remove_piece_at`(*square*)
Removes a piece from the given square if present.

`set_piece_at`(*square*, *piece*)
Sets a piece at the given square. An existing piece is replaced.

`is_attacked_by`(*color*, *square*)
Checks if the given side attacks the given square. Pinned pieces still count as attackers.

`attackers`(*color*, *square*)
Gets a set of attackers of the given color for the given square.

Returns a set of squares.

`is_check`()
Checks if the current side to move is in check.

`is_into_check`(*move*)
Checks if the given move would move would leave the king in check or put it into check.

`was_into_check`()
Checks if the king of the other side is attacked. Such a position is not valid and could only be reached by an illegal move.

`is_game_over`()
Checks if the game is over due to checkmate, stalemate or insufficient mating material.

`is_checkmate`()
Checks if the current position is a checkmate.

`is_stalemate`()
Checks if the current position is a stalemate.

is_insufficient_material()
Checks for a draw due to insufficient mating material.

push(*move*)
Updates the position with the given move and puts it onto a stack.

Null moves just increment the move counters, switch turns and forfeit en passant capturing.

No validation is performed. For performance moves are assumed to be at least pseudo legal. Otherwise there is no guarantee that the previous board state can be restored. To check it yourself you can use:

```
>>> move in board.pseudo_legal_moves
True
```

pop()
Restores the previous position and returns the last move from the stack.

peek()
Gets the last move from the move stack.

set_epd(*epd*)
Parses the given EPD string and uses it to set the position.

If present the *hmvc* and the *fmvn* are used to set the half move clock and the fullmove number. Otherwise *0* and *1* are used.

Returns a dictionary of parsed operations. Values can be strings, integers, floats or move objects.

Raises *ValueError* if the EPD string is invalid.

epd(***operations*)
Gets an EPD representation of the current position.

EPD operations can be given as keyword arguments. Supported operands are strings, integers, floats and moves. All other operands are converted to strings.

hmvc and *fmvc* are *not* included by default. You can use:

```
>>> board.epd(hmvc=board.halfmove_clock, fmvc=board.fullmove_number)
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - hmvc 0; fmvc 1;'
```

set_fen(*fen*)
Parses a FEN and sets the position from it.

Rasies *ValueError* if the FEN string is invalid.

fen()
Gets the FEN representation of the position.

parse_san(*san*)
Uses the current position as the context to parse a move in standard algebraic notation and return the corresponding move object.

The returned move is guaranteed to be either legal or a null move.

Raises *ValueError* if the SAN is invalid or ambigous.

push_san(*san*)
Parses a move in standard algebraic notation, makes the move and puts it on the the move stack.

Raises *ValueError* if neither legal nor a null move.

Returns the move.

san (*move*)
Gets the standard algebraic notation of the given move in the context of the current position.

There is no validation. It is only guaranteed to work if the move is legal or a null move.

status ()
Gets a bitmask of possible problems with the position. Move making, generation and validation are only guaranteed to work on a completely valid board.

zobrist_hash (*array=None*)
Returns a Zobrist hash of the current position.

A zobrist hash is an exclusive or of pseudo random values picked from an array. Which values are picked is decided by features of the position, such as piece positions, castling rights and en-passant squares. For this implementation an array of 781 values is required.

The default behaviour is to use values from *POLYGLOT_RANDOM_ARRAY*, which makes for hashes compatible with polyglot opening books.

7.3 PGN parsing and writing

7.3.1 Game model

Games are represented as a tree of moves. Each *GameNode* can have extra information such as comments. The root node of a game (*Game* extends *GameNode*) also holds general information, such as game headers.

class chess.pgn.**Game**
The root node of a game with extra information such as headers and the starting position.

By default the following 7 headers are provided in an ordered dictionary:

```
>>> game = chess.pgn.Game()
>>> game.headers["Event"]
'?'
>>> game.headers["Site"]
'?'
>>> game.headers["Date"]
'????.??.??'
>>> game.headers["Round"]
'?'
>>> game.headers["White"]
'?'
>>> game.headers["Black"]
'?'
>>> game.headers["Result"]
'*'
```

Also has all the other properties and methods of *GameNode*.

headers
A *collections.OrderedDict()* of game headers.

board ()
Gets the starting position of the game as a bitboard.

Unless the *SetUp* and *FEN* header tags are set this is the default starting position.

setup (*board*)
Setup a specific starting position. This sets (or resets) the *SetUp* and *FEN* header tags.

class `chess.pgn.`**`GameNode`**

`parent`
The parent node or *None* if this is the root node of the game.

`move`
The move leading to this node or *None* if this is the root node of the game.

`nags` **= set()**
A set of NAGs as integers. NAGs always go behind a move, so the root node of the game can have none.

`comment` **= ‘’**
A comment that goes behind the move leading to this node. The root node of the game can have no comment.

`starting_comment` **= ‘’**
A comment for the start of a variation or the game. Only nodes that actually start a variation (*starts_variation()*) and the game itself can have a starting comment.

`variations`
A list of child nodes.

`board``()`
Gets a bitboard with the position of the node.

Its a copy, so modifying the board will not alter the game.

`root``()`
Gets the root node, i.e. the game.

`end``()`
Follows the main variation to the end and returns the last node.

`starts_variation``()`
Checks if this node starts a variation (and can thus have a starting comment). The beginning of the game is also the start of a variation.

`is_main_line``()`
Checks if the node is in the main line of the game.

`is_main_variation``()`
Checks if this node is the first variation from the point of view of its parent. The root node also is in the main variation.

`variation``(`*move*`)`
Gets a child node by move or index.

`has_variation``(`*move*`)`
Checks if the given move appears as a variation.

`promote_to_main``(`*move*`)`
Promotes the given move to the main variation.

`promote``(`*move*`)`
Moves the given variation one up in the list of variations.

`demote``(`*move*`)`
Moves the given variation one down in the list of variations.

`remove_variation``(`*move*`)`
Removes a variation by move.

add_variation(*move*, *comment=''*, *starting_comment=''*, *nags=set([])*)
Creates a child node with the given attributes.

add_main_variation(*move*, *comment=''*)
Creates a child node with the given attributes and promotes it to the main variation.

7.3.2 Parsing

chess.pgn.**read_game**(*handle*)
Reads a game from a file opened in text mode.

By using text mode the parser does not need to handle encodings. It is the callers responsibility to open the file with the correct encoding. According to the specification PGN files should be ASCII. Also UTF-8 is common. So this is usually not a problem.

```
>>> pgn = open("data/games/kasparov-deep-blue-1997.pgn")
>>> first_game = chess.pgn.read_game(pgn)
>>> second_game = chess.pgn.read_game(pgn)
>>>
>>> first_game.headers["Event"]
'IBM Man-Machine, New York USA'
```

Use *StringIO* to parse games from a string.

```
>>> pgn_string = "1. e4 e5 2. Nf3 *"
>>>
>>> try:
>>>     from StringIO import StringIO # Python 2
>>> except ImportError:
>>>     from io import StringIO # Python 3
>>>
>>> pgn = StringIO(pgn_string)
>>> game = chess.pgn.read_game(pgn)
```

The end of a game is determined by a completely blank line or the end of the file. (Of course blank lines in comments are possible.)

According to the standard at least the usual 7 header tags are required for a valid game. This parser also handles games without any headers just fine.

Raises *ValueError* if invalid moves are encountered in the movetext.

Returns the parsed game or *None* if the EOF is reached.

chess.pgn.**scan_offsets**(*handle*)
Scan a PGN file opened in text mode.

Yields the starting offsets of all the games, so that they can be seeked later. Since actually parsing many games from a big file is relatively expensive, this is a better way to read only a specific game.

```
>>> pgn = open("mega.pgn")
>>> offsets = chess.pgn.scan_offsets(pgn)
>>> first_game_offset = next(offsets)
>>> second_game_offset = next(offsets)
>>> pgn.seek(second_game_offset)
>>> second_game = chess.pgn.read_game(pgn)
```

The PGN standard requires each game to start with an Event-tag. So does this scanner.

7.3.3 Writing

If you want to export your game game with all headers, comments and variations you can use:

```
>>> print(game)
[Event "?"]
[Site "?"]
[Date "????.??.??"]
[Round "?"]
[White "?"]
[Black "?"]
[Result "*"]

1. e4 e5 { Comment } *
```

Remember that games in files should be separated with extra blank lines.

```
>>> print(game, file=handle, end="\n\n")
```

Use exporter objects if you need more control. Exporter objects are used to allow extensible formatting of PGN like data.

class `chess.pgn.`**`StringExporter`**(*columns=80*)

Allows exporting a game as a string.

The export method of *Game* also provides options to include or exclude headers, variations or comments. By default everything is included.

```
>>> exporter = chess.pgn.StringExporter()
>>> game.export(exporter, headers=True, variations=True, comments=True)
>>> pgn_string = str(exporter)
```

Only *columns* characters are written per line. If *columns* is *None* then the entire movetext will be on a single line. This does not affect header tags and comments.

There will be no newlines at the end of the string.

class `chess.pgn.`**`FileExporter`**(*handle*, *columns=80*)

Like a StringExporter, but games are written directly to a text file.

There will always be a blank line after each game. Handling encodings is up to the caller.

```
>>> new_pgn = open("new.pgn", "w")
>>> exporter = chess.pgn.FileExporter(new_pgn)
>>> game.export(exporter)
```

7.3.4 NAGs

Numeric anotation glyphs describe moves and positions using standardized codes that are understood by many chess programs. During PGN parsing, annotations like *!*, *?*, *!!*, etc. are also converted to NAGs.

`NAG_NULL` **= 0**

`NAG_GOOD_MOVE` **= 1**

A good move. Can also be indicated by *!* in PGN notation.

`NAG_MISTAKE` **= 2**

A mistake. Can also be indicated by *?* in PGN notation.

`NAG_BRILLIANT_MOVE` **= 3**

A brilliant move. Can also be indicated by *!!* in PGN notation.

NAG_BLUNDER = 4
A blunder. Can also be indicated by *??* in PGN notation.

NAG_SPECULATIVE_MOVE = 5
A speculative move. Can also be indicated by *!?* in PGN notation.

NAG_DUBIOUS_MOVE = 6
A dubious move. Can also be indicated by *?!* in PGN notation.

7.4 Polyglot opening book reading

chess.polyglot.**open_reader**(*path*)
Creates a reader for the file at the given path.

```
>>> with open_reader("data/opening-books/performance.bin") as reader:
>>>     entries = reader.get_entries_for_position(board)
```

class chess.polyglot.**Entry**(*key*, *raw_move*, *weight*, *learn*)
An entry from a polyglot opening book.

key
The Zobrist hash of the position.

raw_move
The raw binary representation of the move. Use the *move()* method to extract a move object from this.

weight
An integer value that can be used as the weight for this entry.

learn
Another integer value that can be used for extra information.

move()
Gets the move (as a *Move* object).

class chess.polyglot.**Reader**(*handle*)
A reader for a polyglot opening book opened in binary mode. The file has to be seekable.

Provides methods to seek entries for specific positions but also ways to efficiently use the opening book like a list.

```
>>> # Get the number of entries
>>> len(reader)
92954

>>> # Get the nth entry
>>> entry = reader[n]

>>> # Iteration
>>> for entry in reader:
>>>     pass

>>> # Backwards iteration
>>> for entry in reversed(reader):
>>>     pass
```

seek_entry(*offset*, *whence=0*)
Seek an entry by its index.

Translated directly to a low level seek on the binary file. *whence* is equivalent.

`seek_position`(*position*)
Seek the first entry for the given position.

Raises *KeyError* if there are no entries for the position.

`next_raw`()
Reads the next raw entry as a tuple.

Raises *StopIteration* at the EOF.

`next`()
Reads the next *Entry*.

Raises *StopIteration* at the EOF.

`get_entries_for_position`(*position*)
Seeks a specific position and yields all entries.

`chess.`**`POLYGLOT_RANDOM_ARRAY`** **= [0x9D39247E33776D41, ..., 0xF8D626AAAF278509]**
Array of 781 polyglot compatible pseudo random values for Zobrist hashing.

CHAPTER 8

Indices and tables

- *genindex*
- *search*