# python-chess

***Release 0.11.1***

September 07, 2015

Contents

CHAPTER 1

# Introduction

python-chess is a pure Python chess library with move generation and validation and handling of common formats. This is the scholars mate in python-chess:

```
>>> import chess

>>> board = chess.Board()

>>> 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/

- Core
- PGN parsing and writing
- Polyglot opening book reading
- Syzygy endgame tablebase probing
- Gaviota endgame tablebase probing
- UCI engine communication
- Changelog

CHAPTER 3

# Features

- Supports Python 2.7 and Python 3.3+.

```
>>> # Python 2 compability for the following examples.
>>> from __future__ import print_function
```

- Supports standard chess and Chess960.

```
>>> board = chess.Board(chess960=True)
```

- Legal move generator and move validation. This includes all castling rules and en passant captures.

```
>>> board = chess.Board()
>>> board.legal_moves
<LegalMoveGenerator at 0x... (Na3, Nc3, Nf3, Nh3, a3, b3, c3, d3, e3, f3, g3, h3, a4, b4, c4, d4
>>> chess.Move.from_uci("a8a1") in board.legal_moves
False
```

- Make and unmake moves.

```
>>> Nf3 = chess.Move.from_uci("g1f3")
>>> board.push(Nf3) # Make the move

>>> board.pop() # Unmake the last move
Move.from_uci('g1f3')
```

- Show a simple ASCII board.

```
>>> board = chess.Board("r1bqkb1r/pppp1Qpp/2n2n2/4p3/2B1P3/8/PPPP1PPP/RNB1K1NR b KQkq - 0 4")
>>> print(board)
r . b q k b . r
p p p p . Q p p
. . n . . n . .
. . . . p . . .
. . B . P . . .
. . . . . . . .
P P P P . P P P
R N B . K . N R
```

- Detects checkmates, stalemates and draws by insufficient material.

```
>>> board.is_stalemate()
False
>>> board.is_insufficient_material()
False
>>> board.is_game_over()
```

```
True
>>> board.halfmove_clock
0
```

- Detects repetitions. Has a half move clock.

```
>>> board.can_claim_threefold_repetition()
False
>>> board.halfmove_clock
0
>>> board.can_claim_fifty_moves()
False
>>> board.can_claim_draw()
False
```

With the new rules from July 2014 a game ends drawn (even without a claim) once a fivefold repetition occurs or if there are 75 moves without a pawn push or capture. Other ways of ending a game take precedence.

```
>>> board.is_fivefold_repetition()
False
>>> board.is_seventyfive_moves()
False
```

- 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
>>> print(attackers)
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . 1 .
. . . . . 1 .
```

- Detects absolute pins and their directions.

```
>>> board.is_pinned(chess.BLACK, chess.E8)
True
>>> pin = board.pin(chess.BLACK, chess.E8)
>>> pin
SquareSet(0b1000000100000010000001000000000000000000000000000000000000000000)
>>> print(pin)
. . . . 1 . . .
. . . . . 1 . .
. . . . . . 1 .
. . . . . . . 1
. . . . . . . .
. . . . . . . .
```

```
. . . . . . . .
. . . . . . . .
```

- Parses and creates SAN representation of moves.

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

- Parses and creates FENs, extended FENs and Shredder FENs.

```
>>> board.fen()
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1'
>>> board.shredder_fen()
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w HAha - 0 1'
>>> board = chess.Board("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.Board()
>>> board.epd(bm=board.parse_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. Docs.

```
>>> import chess.polyglot

>>> book = chess.polyglot.open_reader("data/polyglot/performance.bin")

>>> board = chess.Board()
>>> main_entry = book.find(board)
>>> main_entry.move()
Move.from_uci('e2e4')
>>> main_entry.weight
1
>>> main_entry.learn
0

>>> book.close()
```

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

```
>>> import chess.pgn

>>> pgn = open("data/pgn/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'
```

- Probe Syzygy endgame tablebases (DTZ, WDL). Docs.

```
>>> import chess.syzygy

>>> tablebases = chess.syzygy.open_tablebases("data/syzygy")

>>> # Black to move is losing in 53 half moves (distance to zero) in this
>>> # KNBvK endgame.
>>> board = chess.Board("8/2K5/4B3/3N4/8/8/4k3/8 b - - 0 1")
>>> tablebases.probe_dtz(board)
-53

>>> tablebases.close()
```

- Probe Gaviota endgame tablebases (DTM, WDL) via a wrapper around libgtb. Docs.

```
>>> import chess.gaviota

>>> tablebases = chess.gaviota.open_tablebases("data/gaviota")

>>> # White to move mates in 31 half moves in this KRvK endgame.
>>> board = chess.Board("8/8/8/8/4k3/8/6R1/7K w - - 0 1")
>>> tablebases.probe_dtm(board)
31

>>> tablebases.close()
```

- Communicate with an UCI engine. Docs.

```
>>> import chess.uci
>>> import time

>>> engine = chess.uci.popen_engine("stockfish")
>>> engine.uci()
>>> engine.author
'Tord Romstad, Marco Costalba and Joona Kiiski'

>>> # Synchronous mode.
>>> board = chess.Board("1k1r4/pp1b1R2/3q2pp/4p3/2B5/4Q3/PPP2B2/2K5 b - - 0 1")
>>> engine.position(board)
```

```
>>> engine.go(movetime=2000) # Gets tuple of bestmove and ponder move.
BestMove(bestmove=Move.from_uci('d6d1'), ponder=Move.from_uci('c1d1'))

>>> # Asynchronous mode.
>>> def callback(command):
...     bestmove, ponder = command.result()
...     assert bestmove == chess.Move.from_uci('d6d1')
...
>>> command = engine.go(movetime=2000, async_callback=callback)
>>> command.done()
False
>>> command.result()
BestMove(bestmove=Move.from_uci('d6d1'), ponder=Move.from_uci('c1d1'))
>>> command.done()
True

>>> # Quit.
>>> engine.quit()
0
```

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.

You can install the *gmpy2* or *gmpy* (https://pypi.python.org/pypi/gmpy2) modules in order to get a slight performance boost on basic operations like bitscans and population counts.

python-chess only imports very basic general (non-chess-related) operations from native libraries. All logic is pure Python. There will always be pure Python fallbacks.

CHAPTER 5

# Installing

- With pip:

```
pip install futures # Backport for Python < 3.2
pip install python-chess
```

- From current source code:

```
python setup.py sdist
python setup.py install
```

CHAPTER 6

# Featured projects

If you like, let me know if you are creating something intresting with python-chess, for example:

- a stand alone chess computer based on DGT board - http://www.picochess.org/
- a website to probe Syzygy endgame tablebases - https://syzygy-tables.info/
- a cross platform chess GUI - https://asdfjkl.github.io/jerry/
- extracting reasoning from chess engines - https://github.com/pcattori/deep-blue-talks

CHAPTER 7

# License

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

Thanks to Sam Tannous for publishing his approach to avoid rotated bitboards with direct lookup (pdf) alongside his GPL2+ engine Shatranj. Some of the bitboard move generation parts are ported from there.

Thanks to Ronald de Man for his Syzygy endgame tablebases (https://github.com/syzygy1/tb). The probing code in python-chess is very directly ported from his C probing code.

Thanks to Miguel A. Ballicora for his Gaviota tablebases (https://github.com/michiguel/Gaviota-Tablebases).

CHAPTER 8

# Contents

## 8.1 Core

### 8.1.1 Colors

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

chess.**WHITE = True**

chess.**BLACK = False**

You can get the opposite color using *not color*.

### 8.1.2 Piece types

chess.**PAWN = 1**

chess.**KNIGHT = 2**

chess.**BISHOP = 3**

chess.**ROOK = 4**

chess.**QUEEN = 5**

chess.**KING = 6**

### 8.1.3 Squares

chess.**A1 = 0**

chess.**B1 = 1**

and so on to

chess.**H7 = 62**

chess.**H8 = 63**

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

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

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

chess.**RANK_NAMES = ['1', '2', ..., '7', '8']**

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

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

chess.**square**(*file_index*, *rank_index*)
Gets a square number by file and rank index.

### 8.1.4 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.

**unicode_symbol**(*invert_color=False*)
Gets the unicode character for the piece.

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

Raises ValueError if the symbol is invalid.

### 8.1.5 Moves

**class** chess.**Move**(*from_square*, *to_square*, *promotion=None*)
Represents a move from a square to a square and possibly the promotion piece type.

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
```

### 8.1.6 Board

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

**class** `chess.`**Board**(*fen='rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1'*, *chess960=False*)
A board 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 board is initialized to the starting position, unless otherwise specified in the optional *fen* argument. If *fen* is `None` an empty board is created.

Optionally supports *chess960*. In Chess960 castling moves are encoded by a king move to the corresponding rook square.

**turn**
The side to move.

**castling_rights**
Bitmask of the rooks with castling rights.

```
>>> white_castle_kingside = board.castling_rights & chess.BB_H1
```

Also see *has_castling_rights()*, *has_kingside_castling_rights()*, *has_queenside_castling_rights()* and *has_chess960_castling_rights()*.

**ep_square**
The potential en passant square on the third or sixth rank or `0`.

Use *has_legal_en_passant()* to test if en passant capturing 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.

**chess960**
Whether the board is in Chess960 mode. In Chess960 castling moves are represented as king moves to the corresponding rook square.

**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
```

Wraps `generate_pseudo_legal_moves()` and `is_pseudo_legal()`.

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

Wraps `generate_legal_moves()` and `is_legal()`.

**move_stack**
The move stack. Use *Board.push()*, *Board.pop()*, *Board.peek()* and *Board.clear_stack()* for manipulation.

**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 rights are removed.

In order to be in a valid *status()* at least kings need to be put on the board.

**clear_stack**()
Clears the move stack and transposition table.

**pieces**(*piece_type*, *color*)
Gets pieces of the given type and color.

Returns a *set of squares*.

**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. Pawns that can be captured en passant are attacked.

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

Pinned pieces still count as attackers. Pawns that can be captured en passant are attacked.

Returns a *set of squares*.

**attacks**(*square*)
Gets a set of attacked squares from a given square.

There will be no attacks if the square is empty. Pinned pieces are still attacking other squares. Pawns will attack pawns they could capture en passant.

Returns a *set of squares*.

**pin**(*color*, *square*)
Detects pins of the given square to the king of the given color.

Returns a *set of squares* that mask the rank, file or diagonal of the pin. If there is no pin, then a mask of the entire board is returned.

**is_pinned**(*color*, *square*)
Detects if the given square is pinned to the king of the given color.

**is_check**()
Returns if the current side to move is in check.

**is_into_check**(*move*)
Checks if the given move would leave the king in check or put it into check. The move must be at least pseudo legal.

**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, insufficient mating material, the seventyfive-move rule or fivefold repetition.

**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.

**is_seventyfive_moves**()
Since the first of July 2014 a game is automatically drawn (without a claim by one of the players) if the half move clock since a capture or pawn move is equal to or grather than 150. Other means to end a game take precedence.

**is_fivefold_repetition**()
Since the first of July 2014 a game is automatically drawn (without a claim by one of the players) if a position occurs for the fifth time on consecutive alternating moves.

**can_claim_draw**()
Checks if the side to move can claim a draw by the fifty-move rule or by threefold repetition.

**can_claim_fifty_moves**()
Draw by the fifty-move rule can be claimed once the clock of halfmoves since the last capture or pawn move becomes equal or greater to 100 and the side to move still has a legal move they can make.

**can_claim_threefold_repetition**()
Draw by threefold repetition can be claimed if the position on the board occured for the third time or if such a repetition is reached with one of the possible legal moves.

**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.

**has_legal_en_passant**()
Checks if there is a legal en passant capture.

**fen**()
Gets the FEN representation of the position.

A FEN string (e.g. `rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1`) consists of the position part (`board_fen()`), the turn, the castling part (`castling_xfen()`), a relevant en passant square (`ep_square`, *has_legal_en_passant()*), the halfmove clock and the fullmove number.

**shredder_fen**()
Gets the Shredder FEN representation of the position.

Castling rights are encoded by the file of the rook. The starting castling rights in normal chess are HAha.

Use `castling_shredder_fen()` to get just the castling part.

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

Rasies `ValueError` if the FEN 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 and lists of moves and None. All other operands are converted to strings.

A list of moves for *pv* will be interpreted as a variation. All other move lists are interpreted as a set of moves in the current position.

*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_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.

**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.

**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 ambiguous.

**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.

**uci**(*move*, *chess960=None*)
Gets the UCI notation of the move.

*chess960* defaults to the mode of the board. Pass `True` to force *UCI_Chess960* mode.

**parse_uci**(*uci*)
Parses the given move in UCI notation.

Supports both UCI_Chess960 and standard UCI notation.

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

Raises `ValueError` if the move is invalid or illegal in the current position (but not a null move).

**push_uci**(*uci*)
Parses a move in UCI notation and puts it on the move stack.

Raises `ValueError` if the move is invalid or illegal in the current position (but not a null move).

Returns the move.

**is_en_passant**(*move*)
Checks if the given pseudo-legal move is an en passant capture.

**is_capture**(*move*)
Checks if the given pseudo-legal move is a capture.

**is_castling**(*move*)
Checks if the given pseudo-legal move is a castling move.

**is_kingside_castling**(*move*)
Checks if the given pseudo-legal move is a kingside castling move.

**is_queenside_castling**(*move*)
Checks if the given pseudo-legal move is a queenside castling move.

**has_castling_rights**(*color*)
Checks if the given side has castling rights.

**has_kingside_castling_rights**(*color*)
Checks if the given side has kingside (that is h-side in Chess960) castling rights.

**has_queenside_castling_rights**(*color*)
Checks if the given side has queenside (that is a-side in Chess960) castling rights.

**has_chess960_castling_rights**()
Checks if there are castling rights that are only possible in Chess960.

**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.

`STATUS_VALID` for a completely valid board.

Otherwise bitwise combinations of: `STATUS_NO_WHITE_KING`, `STATUS_NO_BLACK_KING`, `STATUS_TOO_MANY_KINGS`, `STATUS_TOO_MANY_WHITE_PAWNS`, `STATUS_TOO_MANY_BLACK_PAWNS`, `STATUS_PAWNS_ON_BACKRANK`, `STATUS_TOO_MANY_WHITE_PIECES`, `STATUS_TOO_MANY_BLACK_PIECES`, `STATUS_BAD_CASTLING_RIGHTS`, `STATUS_INVALID_EP_SQUARE`, `STATUS_OPPOSITE_CHECK`.

**is_valid**()
Checks if the board is valid.

Move making, generation and validation are only guaranteed to work on a completely valid board.

See *status()* for details.

**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.

**copy**()
Creates a copy of the board.

**classmethod empty**(*chess960=False*)
Creates a new empty board. Also see *clear()*.

**classmethod from_epd**(*epd*, *chess960=False*)
Creates a new board from an EPD string. See *set_epd()*.

Returns the board and the dictionary of parsed operations as a tuple.

### 8.1.7 Square sets

**class** chess.**SquareSet**(*mask=0*)
A set of squares.

```
>>> squares = chess.SquareSet(chess.BB_B1 | chess.BB_G1)
>>> squares
SquareSet(0b1000010)
```

```
>>> print(squares)
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. 1 . . . . 1 .
```

```
>>> len(squares)
2
```

```
>>> bool(squares)
True
```

```
>>> chess.B1 in squares
True
```

```
>>> for square in squares:
...     # 1 -- chess.B1
...     # 6 -- chess.G1
...     print(square)
...
1
6
```

```
>>> list(squares)
[1, 6]
```

Square sets are internally represented by 64 bit integer masks of the included squares. Bitwise operations can be used to compute unions, intersections and shifts.

```
>>> int(squares)
66
```

Also supports common set operations like `issubset()`, `issuperset()`, `union()`, `intersection()`, `difference()`, `symmetric_difference()` and `copy()` as well as `update()`, `intersection_update()`, `difference_update()`, `symmetric_difference_update()` and `clear()`.

**Warning** Square sets can be used as dictionary keys, but do not modify them when doing this.

**add**(*square*)
Add a square to the set.

**remove**(*square*)
Remove a square from the set.

Raises `KeyError` if the given square was not in the set.

**discard**(*square*)
Discards a square from the set.

**pop**()
Removes a square from the set and returns it.

Raises `KeyError` on an empty set.

Common integer masks are:

`chess.`**BB_VOID = 0**

`chess.`**BB_ALL**

`chess.`**BB_LIGHT_SQUARES**

`chess.`**BB_DARK_SQUARES**

Single squares:

`chess.`**BB_SQUARES = [chess.BB_A1, chess.BB_B1, ..., chess.BB_G8, chess.BB_H8]**

Ranks and files:

`chess.`**BB_RANKS = [chess.BB_RANK_1, ..., chess.BB_RANK_8]**

chess.**BB_FILES = [chess.BB_FILE_A, ..., chess.BB_FILE_H]**

## 8.2 PGN parsing and writing

### 8.2.1 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.

**errors**

A list of illegal or ambiguous move errors encountered while parsing the game.

**board**()

Gets the starting position of the game.

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*, *FEN* and *Variant* 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. Comments that occur before any move are assigned to the root node.

**starting_comment = ''**
A comment for the start of a variation. Only nodes that actually start a variation (*starts_variation()*) can have a starting comment. The root node can not have a starting comment.

**variations**
A list of child nodes.

**board**()
Gets a board with the position of the node.

It's a copy, so modifying the board will not alter the game.

**san**()
Gets the standard algebraic notation of the move leading to this node.

Do not call this on the root node.

**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 root node does not start a variation and can have no starting comment.

**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=()*)
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.

### 8.2.2 Parsing

chess.pgn.**read_game**(*handle*, *error_handler=<function _raise>*)

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/pgn/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.

The parser is relatively forgiving when it comes to errors. It skips over tokens it can not parse. However it is difficult to handle illegal or ambiguous moves. If such a move is encountered the default behaviour is to stop right in the middle of the game and raise `ValueError`. If you pass `None` for *error_handler* all errors are silently ignored, instead. If you pass a function this function will be called with the error as an argument.

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

chess.pgn.**scan_headers**(*handle*)

Scan a PGN file opened in text mode for game offsets and headers.

Yields a tuple for each game. The first element is the offset. The second element is an ordered dictionary of game headers.

Since actually parsing many games from a big file is relatively expensive, this is a better way to look only for specific games and seek and parse them later.

This example scans for the first game with Kasparov as the white player.

```
>>> pgn = open("mega.pgn")
>>> for offset, headers in chess.pgn.scan_headers(pgn):
...     if "Kasparov" in headers["White"]:
...         kasparov_offset = offset
...         break
```

Then it can later be seeked an parsed.

```
>>> pgn.seek(kasparov_offset)
>>> game = chess.pgn.read_game(pgn)
```

This also works nicely with generators, scanning lazily only when the next offset is required.

```
>>> white_win_offsets = (offset for offset, headers in chess.pgn.scan_headers(pgn)
...                                if headers["Result"] == "1-0")
>>> first_white_win = next(white_win_offsets)
>>> second_white_win = next(white_win_offsets)
```

**Warning** Be careful when seeking a game in the file while more offsets are being generated.

chess.pgn.**scan_offsets**(*handle*)

Scan a PGN file opened in text mode for game offsets.

Yields the starting offsets of all the games, so that they can be seeked later. This is just like *scan_headers()* but more efficient if you do not actually need the header information.

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

### 8.2.3 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.

`chess.pgn.Game.export()` 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)
```

### 8.2.4 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.

`chess.pgn.`**NAG_GOOD_MOVE = 1**
A good move. Can also be indicated by `!` in PGN notation.

`chess.pgn.`**NAG_MISTAKE = 2**
A mistake. Can also be indicated by `?` in PGN notation.

`chess.pgn.`**NAG_BRILLIANT_MOVE = 3**
A brilliant move. Can also be indicated by `!!` in PGN notation.

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

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

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

## 8.3 Polyglot opening book reading

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

```
>>> with open_reader("data/polyglot/performance.bin") as reader:
...     for entry in reader.find_all(board):
...         print(entry.move(), entry.weight, entry.learn)
e2e4 1 0
d2d4 1 0
c2c4 1 0
```

**class** `chess.polyglot.`**Entry**
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**(*chess960=False*)
Gets the move (as a *Move* object).

**class** chess.polyglot.**MemoryMappedReader**(*filename*)
Maps a polyglot opening book to memory.

**close**()
Closes the reader.

**find**(*board*, *minimum_weight=1*)
Finds the main entry for the given position or zobrist hash.

The main entry is the first entry with the highest weight.

By default entries with weight 0 are excluded. This is a common way to delete entries from an opening book without compacting it. Pass *minimum_weight* 0 to select all entries.

Raises IndexError if no entries are found.

**find_all**(*board*, *minimum_weight=1*)
Seeks a specific position and yields corresponding entries.

**choice**(*board*, *minimum_weight=1*, *random=<module 'random' from '/home/docs/checkouts/readthedocs.org/user_builds/python-chess/envs/v0.11.1/lib/python3.4/random.py'>*)
Uniformly selects a random entry for the given position.

Raises IndexError if no entries are found.

**weighted_choice**(*board*, *random=<module 'random' from '/home/docs/checkouts/readthedocs.org/user_builds/python-chess/envs/v0.11.1/lib/python3.4/random.py'>*)
Selects a random entry for the given position, distributed by the weights of the entries.

Raises IndexError if no entries are found.

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

## 8.4 Syzygy endgame tablebase probing

Syzygy tablebases provide **WDL** (win/draw/loss) and **DTZ** (distance to zero) information for all endgame positions with up to 6 pieces. Positions with castling rights are not included.

chess.syzygy.**open_tablebases**(*directory=None*, *load_wdl=True*, *load_dtz=True*)
Opens a collection of tablebases for probing. See *Tablebases*.

**class** chess.syzygy.**Tablebases**(*directory=None*, *load_wdl=True*, *load_dtz=True*)
Manages a collection of tablebase files for probing.

Syzygy tables come in files like *KQvKN.rtbw* or *KRBvK.rtbz*, one WDL (*.rtbw*) and DTZ (*.rtbz*) file for each material composition.

Directly loads tables from *directory*. See *open_directory()*.

**open_directory**(*directory*, *load_wdl=True*, *load_dtz=True*)
Loads tables from a directory.

By default all available tables with the correct file names (e.g. *KQvKN.rtbw* or *KRBvK.rtbz*) are loaded.

Returns the number of successfully openened and loaded tablebase files.

**probe_wdl**(*board*)
Probes WDL tables for win/draw/loss-information.

Probing is thread-safe when done with different *board* objects and if *board* objects are not modified during probing.

Returns `None` if the position was not found in any of the loaded tables.

Returns `2` if the side to move is winning, `0` if the position is a draw and `-2` if the side to move is losing.

Returns `1` in case of a cursed win and `-1` in case of a blessed loss. Mate can be forced but the position can be drawn due to the fifty-move rule.

```
>>> with chess.syzygy.open_tablebases("data/syzygy") as tablebases:
...     tablebases.probe_wdl(chess.Board("8/2K5/4B3/3N4/8/8/4k3/8 b - - 0 1"))
...
-2
```

**probe_dtz**(*board*)

Probes DTZ tables for distance to zero information.

Probing is thread-safe when done with different *board* objects and if *board* objects are not modified during probing.

Return `None` if the position was not found in any of the loaded tables. Both DTZ and WDL tables are required in order to probe for DTZ values.

Returns a positive value if the side to move is winning, `0` if the position is a draw and a negative value if the side to move is losing.

A non-zero distance to zero means the number of halfmoves until the next pawn move or capture can be forced, keeping a won position. Minmaxing the DTZ values guarantees winning a won position (and drawing a drawn position), because it makes progress keeping the win in hand. However the lines are not always the most straightforward ways to win. Engines like Stockfish calculate themselves, checking with DTZ, but only play according to DTZ if they can not manage on their own.

```
>>> with chess.syzygy.open_tablebases("data/syzygy") as tablebases:
...     tablebases.probe_dtz(chess.Board("8/2K5/4B3/3N4/8/8/4k3/8 b - - 0 1"))
...
-53
```

**close**()

Closes all loaded tables.

## 8.5 Gaviota endgame tablebase probing

This module is experimental and does not yet come with a pure Python fallback. Instead you have to build and install a shared library:

```
git clone https://github.com/michiguel/Gaviota-Tablebases
cd Gaviota-Tablebases
make
sudo make install
```

Gaviota tablebases provide **WDL** (win/draw/loss) and **DTM** (depth to mate) information for all endgame positions with up to 5 pieces. Positions with castling rights are not included.

chess.gaviota.**open_tablebases**(*directory=None*, *libgtb=None*, *LibraryLoader=<ctypes.LibraryLoader object>*)

Opens a collection of tablebases for probing.

Currently the only access method is via the shared library libgtb. You can optionally provide a specific library name or a library loader. The shared library has global state and caches, so only one instance can be open at a time.

**class** chess.gaviota.**NativeTablebases**(*directory*, *libgtb*)
Provides access to Gaviota tablebases via the shared library libgtb.

**open_directory**(*directory*)
Loads *.gtb.cp4* tables from a directory.

**probe_dtm**(*board*)
Probes for depth to mate information.

Returns `None` if the position was not found in any of the tables.

Otherwise the absolute value is the number of half moves until forced mate. The value is positive if the side to move is winning, otherwise it is negative.

In the example position white to move will get mated in 10 half moves:

```
>>> with chess.gaviota.open_tablebases("data/gaviota") as tablebases:
...     tablebases.probe_dtm(chess.Board("8/8/8/8/8/8/8/K2kr3 w - - 0 1"))
...
-10
```

**probe_wdl**(*board*)
Probes for win/draw/loss-information.

Returns `None` if the position was not found in any of the tables.

Returns `1` if the side to move is winning, `0` if it is a draw, and `-1` if the side to move is losing.

```
>>> with chess.gaviota.open_tablebases("data/gaviota") as tablebases:
...     tablebases.probe_wdl(chess.Board("4k3/8/B7/8/8/8/4K3 w - - 0 1"))
...
0
```

**close**()
Closes all loaded tables and clears all caches.

## 8.6 UCI engine communication

The Universal Chess Interface is a protocol for communicating with engines.

chess.uci.**popen_engine**(*command*, *engine_cls=<class 'chess.uci.Engine'>*)
Opens a local chess engine process.

No initialization commands are sent, so do not forget to send the mandatory *uci* command.

```
>>> engine = chess.uci.popen_engine("/usr/games/stockfish")
>>> engine.uci()
>>> engine.name
'Stockfish 230814 64'
>>> engine.author
'Tord Romstad, Marco Costalba and Joona Kiiski'
```

The input and input streams will be linebuffered and able both Windows and Unix newlines.

chess.uci.**spur_spawn_engine**(*shell*, *command*, *engine_cls=<class 'chess.uci.Engine'>*)
Spwans a remote engine using a Spur shell.

```
>>> import spur
>>> shell = spur.SshShell(hostname="localhost", username="username", password="pw")
>>> engine = chess.uci.spur_spwan_engine(shell, ["/usr/games/stockfish"])
>>> engine.uci()
```

**class** chess.uci.**Engine**(*process*, *Executor=<class 'concurrent.futures.thread.ThreadPoolExecutor'>*)

**process**
The underlying operating system process.

**name**
The name of the engine. Conforming engines should send this as *id name* when they receive the initial *uci* command.

**author**
The author, as sent via *id author*. Just like the name.

**options**
A case insensitive dictionary of *Options*. The engine should send available options when it receives the initial *uci* command.

**uciok**
`threading.Event()` that will be set as soon as *uciok* was received. By then `name`, `author` and `options` should be available.

**return_code**
The return code of the operating system process.

**terminated**
`threading.Event()` that will be set as soon as the underyling operating system process is terminated and the `return_code` is available.

**terminate**(*async_callback=None*)
Terminate the engine.

This is not an UCI command. It instead tries to terminate the engine on operating system level, for example by sending SIGTERM on Unix systems. If possible, first try the *quit* command.

**Returns** The return code of the engine process (or a Future).

**kill**(*async_callback=None*)
Kill the engine.

Forcefully kill the engine process, for example by sending SIGKILL.

**Returns** The return code of the engine process (or a Future).

**is_alive**()
Poll the engine process to check if it is alive.

### 8.6.1 UCI commands

**class** chess.uci.**Engine**(*process*, *Executor=<class 'concurrent.futures.thread.ThreadPoolExecutor'>*)

**uci**(*async_callback=None*)
Tells the engine to use the UCI interface.

This is mandatory before any other command. A conforming engine will send its name, authors and available options.

**Returns** Nothing

**debug**(*on*, *async_callback=None*)
Switch the debug mode on or off.

In debug mode the engine should send additional infos to the GUI to help debugging. This mode should be switched off by default.

**Parameters** **on** – bool

**Returns** Nothing

**isready**(*async_callback=None*)
Command used to synchronize with the engine.

The engine will respond as soon as it has handled all other queued commands.

**Returns** Nothing

**setoption**(*options*, *async_callback=None*)
Set values for the engines available options.

**Parameters** **options** – A dictionary with option names as keys.

**Returns** Nothing

**ucinewgame**(*async_callback=None*)
Tell the engine that the next search will be from a different game.

This can be a new game the engine should play or if the engine should analyse a position from a different game. Using this command is recommended but not required.

**Returns** Nothing

**position**(*board*, *async_callback=None*)
Set up a given position.

Instead of just the final FEN, the initial FEN and all moves leading up to the position will be sent, so that the engine can detect repetitions.

If the position is from a new game it is recommended to use the *ucinewgame* command before the *position* command.

**Parameters** **board** – A *chess.Board*.

**Returns** Nothing

**Raises** `EngineStateException` if the engine is still calculating.

**go**(*searchmoves=None*, *ponder=False*, *wtime=None*, *btime=None*, *winc=None*, *binc=None*, *movestogo=None*, *depth=None*, *nodes=None*, *mate=None*, *movetime=None*, *infinite=False*, *async_callback=None*)
Start calculating on the current position.

All parameters are optional, but there should be at least one of *depth*, *nodes*, *mate*, *infinite* or some time control settings, so that the engine knows how long to calculate.

Note that when using *infinite* or *ponder* the engine will not stop until it is told to.

**Parameters**

- **searchmoves** – Restrict search to moves in this list.
- **ponder** – Bool to enable pondering mode. The engine will not stop pondering in the background until a *stop* command is received.
- **wtime** – Integer of milliseconds white has left on the clock.
- **btime** – Integer of milliseconds black has left on the clock.
- **winc** – Integer of white Fisher increment.
- **binc** – Integer of black Fisher increment.

- **movestogo** – Number of moves to the next time control. If this is not set, but wtime or btime are, then it is sudden death.
- **depth** – Search *depth* ply only.
- **nodes** – Search so many *nodes* only.
- **mate** – Search for a mate in *mate* moves.
- **movetime** – Integer. Search exactly *movetime* milliseconds.
- **infinite** – Search in the background until a *stop* command is received.

**Returns** A tuple of two elements. The first is the best move according to the engine. The second is the ponder move. This is the reply as sent by the engine. Either of the elements may be *None*.

**Raises** `EngineStateException` if the engine is already calculating.

**stop**(*async_callback=None*)
Stop calculating as soon as possible.

**Returns** Nothing.

**ponderhit**(*async_callback=None*)
May be sent if the expected ponder move has been played.

The engine should continue searching but should switch from pondering to normal search.

**Returns** Nothing.

**Raises** `EngineStateException` if the engine is not currently searching in ponder mode.

**quit**(*async_callback=None*)
Quit the engine as soon as possible.

**Returns** The return code of the engine process.

`EngineTerminatedException` is raised if the engine process is no longer alive.

### 8.6.2 Asynchronous communication

By default all operations are executed synchronously and their result is returned. For example

```
>>> engine.go(movetime=2000)
BestMove(bestmove=Move.from_uci('e2e4'), ponder=None)
```

will take about 2000 milliseconds. All UCI commands have an optional *async_callback* argument. They will then immediately return a Future and continue.

```
>>> command = engine.go(movetime=2000, async_callback=True)
>>> command.done()
False
>>> command.result() # Synchronously wait for the command to finish
BestMove(bestmove=Move.from_uci('e2e4'), ponder=None)
>>> command.done()
True
```

Instead of just passing *async_callback=True* a callback function may be passed. It will be invoked **possibly on a different thread** as soon as the command is completed. It takes the command future as a single argument.

```
>>> def on_go_finished(command):
...     # Will likely be executed on a different thread.
...     bestmove, ponder = command.result()
...
>>> command = engine.go(movetime=2000, async_callback=on_go_finished)
```

### 8.6.3 Note about castling moves

There are different ways castling moves may be encoded. The normal way to do it is `e1g1` for short castling. The same move would be `e1h1` in *UCI_Chess960* mode.

This is abstracted away by the UCI module, but if the engine supports it, it is recommended to enable enable *UCI_Chess960* mode.

```
>>> engine.setoption({"UCI_Chess960": True})
```

### 8.6.4 Info handler

Chess engines may send information about their calculations with the *info* command. You can register info handlers to be asynchronously notified whenever the engine sends more information. You would usually subclass the *InfoHandler* class.

**class** `chess.uci.`**Score**

A centipawns or mate score sent by an UCI engine.

**cp**

Evaluation in centipawns or `None`.

**mate**

Mate in x or `None`. Negative if the engine thinks it is going to be mated.

**lowerbound**

If the score is not exact but only a lowerbound.

**upperbound**

If the score is only an upperbound.

**class** `chess.uci.`**InfoHandler**

**info**

The default implementation stores all received information in this dictionary. To get a consistent snapshot use the object as if it were a `threading.Lock()`.

```
>>> # Register the handler.
>>> handler = chess.uci.InfoHandler()
>>> engine.info_handlers.append(handler)
```

```
>>> # Start thinking.
>>> engine.go(infinite=True)
```

```
>>> # Wait a moment, then access a consistent snapshot.
>>> time.sleep(3)
>>> with handler:
...     if 1 in handler.info["score"]:
...         print("Score: ", handler.info["score"][1].cp)
...         print("Mate: ", handler.info["score"][1].mate)
```

```
    Score: 34
    Mate: None
```

**depth** (*x*)
Received search depth in plies.

**seldepth** (*x*)
Received selective search depth in plies.

**time** (*x*)
Received new time searched in milliseconds.

**nodes** (*x*)
Received number of nodes searched.

**pv** (*moves*)
Received the principal variation as a list of moves.

In MultiPV mode this is related to the most recent *multipv* number sent by the engine.

**multipv** (*num*)
Received a new multipv number, starting at 1.

If multipv occurs in an info line, this is guaranteed to be called before *score* or *pv*.

**score** (*cp*, *mate*, *lowerbound*, *upperbound*)
Received a new evaluation in centipawns or a mate score.

*cp* may be *None* if no score in centipawns is available.

*mate* may be *None* if no forced mate has been found. A negative numbers means the engine thinks it will get mated.

lowerbound and upperbound are usually *False*. If *True*, the sent score are just a lowerbound or upperbound.

In MultiPV mode this is related to the most recent *multipv* number sent by the engine.

**currmove** (*move*)
Received a move the engine is currently thinking about.

These moves come directly from the engine. So the castling move representation depends on the UCI_Chess960 option of the engine.

**currmovenumber** (*x*)
Received a new currmovenumber.

**hashfull** (*x*)
Received new information about the hashtable.

The hashtable is x permill full.

**nps** (*x*)
Received new nodes per second statistic.

**tbhits** (*x*)
Received new information about the number of table base hits.

**cpuload** (*x*)
Received new cpuload information in permill.

**string** (*string*)
Received a string the engine wants to display.

**refutation**(*move*, *refuted_by*)
Received a new refutation of a move.

*refuted_by* may be a list of moves representing the mainline of the refutation or *None* if no refutation has been found.

Engines should only send refutations if the *UCI_ShowRefutations* option has been enabled.

**currline**(*cpunr*, *moves*)
Received a new snapshot of a line a specific CPU is calculating.

*cpunr* is an integer representing a specific CPU. *moves* is a list of moves.

**pre_info**(*line*)
Received a new info line about to be processed.

When subclassing remember to call this method of the parent class in order to keep the locking in tact.

**post_info**()
Processing of a new info line has been finished.

When subclassing remember to call this method of the parent class in order to keep the locking in tact.

**on_bestmove**(*bestmove*, *ponder*)
A new bestmove and pondermove have been received.

**on_go**()
A go command is being sent.

Since information about the previous search is invalidated the dictionary with the current information will be cleared.

### 8.6.5 Options

**class** chess.uci.**Option**
Information about an available option for an UCI engine.

**name**
The name of the option.

**type**
The type of the option.

Officially documented types are `check` for a boolean value, `spin` for an integer value between a minimum and a maximum, `combo` for an enumeration of predefined string values (one of which can be selected), `button` for an action and `string` for a textfield.

**default**
The default value of the option.

There is no need to send a *setoption* command with the defaut value.

**min**
The minimum integer value of a *spin* option.

**max**
The maximum integer value of a *spin* option.

**var**
A list of allows string values for a *combo* option.

## 8.7 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.

### 8.7.1 New in v0.11.1

Bugfixes:

- *syzygy.Tablebases.probe_dtz()* has was giving wrong results for some positions with possible en passant capturing. This was found and fixed upstream: https://github.com/official-stockfish/Stockfish/issues/394.
- Ignore extra spaces in UCI *info* lines, as for example sent by the Hakkapeliitta engine. Thanks to Jürgen Précour for reporting

### 8.7.2 New in v0.11.0

Changes:

- **Chess960** support and the **representation of castling moves** has been changed.

  The constructor of board has a new *chess960* argument, defaulting to *False*: *Board(fen=STARTING_FEN, chess960=False)*. That property is available as *Board.chess960*.

  In Chess960 mode the behaviour is as in the previous release. Castling moves are represented as a king move to the corresponding rook square.

  In the default standard chess mode castling moves are represented with the standard UCI notation, e.g. *e1g1* for king-side castling.

  *Board.uci(move, chess960=None)* creates UCI representations for moves. Unlike *Move.uci()* it can convert them in the context of the current position.

  *Board.has_chess960_castling_rights()* has been added to test for castling rights that are impossible in standard chess.

  The modules *chess.polyglot*, *chess.pgn* and *chess.uci* will transparently handle both modes.

- In a previous release *Board.fen()* has been changed to only display an en passant square if a legal en passant move is indeed possible. This has now also been adapted for *Board.shredder_fen()* and *Board.epd()*.

New features:

- Get individual FEN components: *Board.board_fen()*, *Board.castling_xfen()*, *Board.castling_shredder_fen()*.
- Use *Board.has_legal_en_passant()* to test if a position has a legal en passant move.
- Make *repr(board.legal_moves)* human readable.

### 8.7.3 New in v0.10.1

Bugfixes:

- Fix use-after-free in Gaviota tablebase initialization.

### 8.7.4 New in v0.10.0

New dependencies:

- If you are using Python < 3.2 you have to install *futures* in order to use the *chess.uci* module.

Changes:

- There are big changes in the UCI module. Most notably in async mode multiple commands can be executed at the same time (e.g. *go infinite* and then *stop* or *go ponder* and then *ponderhit*).

  *go infinite* and *go ponder* will now wait for a result, i.e. you may have to call *stop* or *ponderhit* from a different thread or run the commands asynchronously.

  *stop* and *ponderhit* no longer have a result.

- The values of the color constants *chess.WHITE* and *chess.BLACK* have been changed. Previously *WHITE* was *0*, *BLACK* was *1*. Now *WHITE* is *True*, *BLACK* is *False*. The recommended way to invert *color* is using *not color*.

- The pseudo piece type *chess.NONE* has been removed in favor of just using *None*.

- Changed the *Board(fen)* constructor. If the optional *fen* argument is not given behavior did not change. However if *None* is passed explicitly an empty board is created. Previously the starting position would have been set up.

- *Board.fen()* will now only show completely legal en passant squares.

- *Board.set_piece_at()* and *Board.remove_piece_at()* will now clear the move stack, because the old moves may not be valid in the changed position.

- *Board.parse_uci()* and *Board.push_uci()* will now accept null moves.

- Changed shebangs from *#!/usr/bin/python* to *#!/usr/bin/env python* for better virtualenv support.

- Removed unused game data files from repository.

Bugfixes:

- PGN: Prefer the game result from the game termination marker over * in the header. These should be identical in standard compliant PGNs. Thanks to Skyler Dawson for reporting this.

- Polyglot: *minimum_weight* for *find()*, *find_all()* and *choice()* was not respected.

- Polyglot: Negative indexing of opening books was raising *IndexError*.

- Various documentation fixes and improvements.

New features:

- Experimental probing of Gaviota tablebases via libgtb.

- New methods to construct boards:

```
>>> chess.Board.empty()
Board('8/8/8/8/8/8/8/8 w - - 0 1')

>>> board, ops = chess.Board.from_epd("4k3/8/8/8/8/8/8/4K3 b - - fmvn 17; hmvc 13")
>>> board
Board('4k3/8/8/8/8/8/8/4K3 b - - 13 17')
>>> ops
{'fmvn': 17, 'hmvc': 13}
```

- Added *Board.copy()* and hooks to let the copy module to the right thing.

- Added *Board.has_castling_rights(color)*, *Board.has_kingside_castling_rights(color)* and *Board.has_queenside_castling_rights(color)*.

- Added *Board.clear_stack()*.
- Support common set operations on *chess.SquareSet()*.

### 8.7.5 New in v0.9.1

Bugfixes:

- UCI module could not handle castling ponder moves. Thanks to Marco Belli for reporting.
- The initial move number in PGNs was missing, if black was to move in the starting position. Thanks to Jürgen Précour for reporting.
- Detect more impossible en passant squares in *Board.status()*. There already was a requirement for a pawn on the fifth rank. Now the sixth and seventh rank must be empty, additionally. We do not do further retrograde analysis, because these are the only cases affecting move generation.

### 8.7.6 New in v0.8.3

Bugfixes:

- The initial move number in PGNs was missing, if black was to move in the starting position. Thanks to Jürgen Précour for reporting.
- Detect more impossible en passant squares in *Board.status()*. There already was a requirement for a pawn on the fifth rank. Now the sixth and seventh rank must be empty, additionally. We do not do further retrograde analysis, because these are the only cases affecting move generation.

### 8.7.7 New in v0.9.0

**This is a big update with quite a few breaking changes. Carefully review the changes before upgrading. It's no problem if you can not update right now. The 0.8.x branch still gets bugfixes.**

Incompatible changes:

- Removed castling right constants. Castling rights are now represented as a bitmask of the rook square. For example:

```
>>> board = chess.Board()

>>> # Standard castling rights.
>>> board.castling_rights == chess.BB_A1 | chess.BB_H1 | chess.BB_A8 | chess.BB_H8
True

>>> # Check for the presence of a specific castling right.
>>> can_white_castle_queenside = chess.BB_A1 & board.castling_rights
```

  Castling moves were previously encoded as the corresponding king movement in UCI, e.g. *e1f1* for white kingside castling. **Now castling moves are encoded as a move to the corresponding rook square** (*UCI_Chess960*-style), e.g. *e1a1*.

  You may use the new methods *Board.uci(move, chess960=True)*, *Board.parse_uci(uci)* and *Board.push_uci(uci)* to handle this transparently.

  The *uci* module takes care of converting moves when communicating with an engine that is not in *UCI_Chess960* mode.

- The *get_entries_for_position(board)* method of polyglot opening book readers has been changed to *find_all(board, minimum_weight=1)*. By default entries with weight 0 are excluded.
- The *Board.pieces* lookup list has been removed.
- In 0.8.1 the spelling of repetition (was repitition) was fixed. *can_claim_threefold_repetition()* and *is_fivefold_repetition()* are the affected method names. Aliases are now removed.
- *Board.set_epd()* will now interpret *bm*, *am* as a list of moves for the current position and *pv* as a variation (represented by a list of moves). Thanks to Jordan Bray for reporting this.
- Removed *uci.InfoHandler.pre_bestmove()* and *uci.InfoHandler.post_bestmove()*.
- *uci.InfoHandler().info["score"]* is now relative to multipv. Use

```
>>> with info_handler as info:
...     if 1 in info["score"]:
...         cp = info["score"][1].cp
```

  where you were previously using

```
>>> with info_handler as info:
...     if "score" in info:
...         cp = info["score"].cp
```

- Clear *uci.InfoHandler()* dictionary at the start of new searches (new *on_go()*), not at the end of searches.
- Renamed *PseudoLegalMoveGenerator.bitboard* and *LegalMoveGenerator.bitboard* to *PseudoLegalMoveGenerator.board* and *LegalMoveGenerator.board*, respectively.
- Scripts removed.
- Python 3.2 compability dropped. Use Python 3.3 or higher. Python 2.7 support is not affected.

New features:

- **Introduced Chess960 support.** *Board(fen)* and *Board.set_fen(fen)* now support X-FENs. Added *Board.shredder_fen()*. *Board.status(allow_chess960=True)* has an optional argument allowing to insist on standard chess castling rules. Added *Board.is_valid(allow_chess960=True)*.
- **Improved move generation using** Shatranj-style direct lookup. **Removed rotated bitboards. Perft speed has been more than doubled.**
- Added *choice(board)* and *weighted_choice(board)* for polyglot opening book readers.
- Added *Board.attacks(square)* to determine attacks *from* a given square. There already was *Board.attackers(color, square)* returning attacks *to* a square.
- Added *Board.is_en_passant(move)*, *Board.is_capture(move)* and *Board.is_castling(move)*.
- Added *Board.pin(color, square)* and *Board.is_pinned(color, square)*.
- There is a new method *Board.pieces(piece_type, color)* to get a set of squares with the specified pieces.
- Do expensive Syzygy table initialization on demand.
- Allow promotions like *e8Q* (usually *e8=Q*) in *Board.parse_san()* and PGN files.
- Patch by Richard C. Gerkin: Added *Board.__unicode__()* just like *Board.__str__()* but with unicode pieces.
- Patch by Richard C. Gerkin: Added *Board.__html__()*.

### 8.7.8 New in v0.8.2

Bugfixes:

- *pgn.Game.setup()* with the standard starting position was failing when the standard starting position was already set. Thanks to Jordan Bray for reporting this.

Optimizations:

- Remove *bswap()* from Syzygy decompression hot path. Directly read integers with the correct endianness.

### 8.7.9 New in v0.8.1

- Fixed pondering mode in uci module. For example *ponderhit()* was blocking indefinitely. Thanks to Valeriy Huz for reporting this.
- Patch by Richard C. Gerkin: Moved searchmoves to the end of the UCI go command, where it will not cause other command parameters to be ignored.
- Added missing check or checkmate suffix to castling SANs, e.g. *O-O-O#*.
- Fixed off-by-one error in polyglot opening book binary search. This would not have caused problems for real opening books.
- Fixed Python 3 support for reverse polyglot opening book iteration.
- Bestmoves may be literally *(none)* in UCI protocol, for example in checkmate positions. Fix parser and return *None* as the bestmove in this case.
- Fixed spelling of repetition (was repitition). *can_claim_threefold_repetition()* and *is_fivefold_repetition()* are the affected method names. Aliases are there for now, but will be removed in the next release. Thanks to Jimmy Patrick for reporting this.
- Added *SquareSet.__reversed__()*.
- Use containerized tests on Travis CI, test against Stockfish 6, improved test coverage amd various minor clean-ups.

### 8.7.10 New in v0.8.0

- **Implement Syzygy endgame tablebase probing.** https://syzygy-tables.info is an example project that provides a public API using the new features.
- The interface for aynchronous UCI command has changed to mimic *concurrent.futures*. *is_done()* is now just *done()*. Callbacks will receive the command object as a single argument instead of the result. The *result* property and *wait()* have been removed in favor of a synchronously waiting *result()* method.
- The result of the *stop* and *go* UCI commands are now named tuples (instead of just normal tuples).
- Add alias *Board* for *Bitboard*.
- Fixed race condition during UCI engine startup. Lines received during engine startup sometimes needed to be processed before the Engine object was fully initialized.

### 8.7.11 New in v0.7.0

- **Implement UCI engine communication.**
- Patch by Matthew Lai: *Add caching for gameNode.board()*.

### 8.7.12 New in v0.6.0

- If there are comments in a game before the first move, these are now assigned to *Game.comment* instead of *Game.starting_comment*. *Game.starting_comment* is ignored from now on. *Game.starts_variation()* is no longer true. The first child node of a game can no longer have a starting comment. It is possible to have a game with *Game.comment* set, that is otherwise completely empty.
- Fix export of games with variations. Previously the moves were exported in an unusual (i.e. wrong) order.
- Install *gmpy2* or *gmpy* if you want to use slightly faster binary operations.
- Ignore superfluous variation opening brackets in PGN files.
- Add *GameNode.san()*.
- Remove *sparse_pop_count()*. Just use *pop_count()*.
- Remove *next_bit()*. Now use *bit_scan()*.

### 8.7.13 New in v0.5.0

- PGN parsing is now more robust: *read_game()* ignores invalid tokens. Still exceptions are going to be thrown on illegal or ambiguous moves, but this behaviour can be changed by passing an *error_handler* argument.

```
>>> # Raises ValueError:
>>> game = chess.pgn.read_game(file_with_illegal_moves)
```

```
>>> # Silently ignores errors and continues parsing:
>>> game = chess.pgn.read_game(file_with_illegal_moves, None)
```

```
>>> # Logs the error, continues parsing:
>>> game = chess.pgn.read_game(file_with_illegal_moves, logger.exception)
```

  If there are too many closing brackets this is now ignored.

  Castling moves like 0-0 (with zeros) are now accepted in PGNs. The *Bitboard.parse_san()* method remains strict as always, though.

  Previously the parser was strictly following the PGN spefification in that empty lines terminate a game. So a game like

```
[Event "?"]

{ Starting comment block }

1. e4 e5 2. Nf3 Nf6 *
```

  would have ended directly after the starting comment. To avoid this, the parser will now look ahead until it finds at least one move or a termination marker like *, *1-0*, *1/2-1/2* or *0-1*.

- Introduce a new function *scan_headers()* to quickly scan a PGN file for headers without having to parse the full games.
- Minor testcoverage improvements.

### 8.7.14 New in v0.4.2

- Fix bug where *pawn_moves_from()* and consequently *is_legal()* weren't handling en passant correctly. Thanks to Norbert Naskov for reporting.

### 8.7.15 New in v0.4.1

- Fix *is_fivefold_repitition()*: The new fivefold repetition rule requires the repetitions to occur on *alternating consecutive* moves.
- Minor testing related improvements: Close PGN files, allow running via setuptools.
- Add recently introduced features to README.

### 8.7.16 New in v0.4.0

- Introduce *can_claim_draw()*, *can_claim_fifty_moves()* and *can_claim_threefold_repitition()*.
- Since the first of July 2014 a game is also over (even without claim by one of the players) if there were 75 moves without a pawn move or capture or a fivefold repetition. Let *is_game_over()* respect that. Introduce *is_seventyfive_moves()* and *is_fivefold_repitition()*. Other means of ending a game take precedence.
- Threefold repetition checking requires efficient hashing of positions to build the table. So performance improvements were needed there. The default polyglot compatible zobrist hashes are now built incrementally.
- Fix low level rotation operations *l90()*, *l45()* and *r45()*. There was no problem in core because correct versions of the functions were inlined.
- Fix equality and inequality operators for *Bitboard*, *Move* and *Piece*. Also make them robust against comparisons with incompatible types.
- Provide equality and inequality operators for *SquareSet* and *polyglot.Entry*.
- Fix return values of incremental arithmetical operations for *SquareSet*.
- Make *polyglot.Entry* a *collections.namedtuple*.
- Determine and improve test coverage.
- Minor coding style fixes.

### 8.7.17 New in v0.3.1

- *Bitboard.status()* now correctly detects *STATUS_INVALID_EP_SQUARE*, instead of errors or false reports.
- Polyglot opening book reader now correctly handles knight underpromotions.
- Minor coding style fixes, including removal of unused imports.

### 8.7.18 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.

### 8.7.19 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.

### 8.7.20 New in v0.1.0

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

### 8.7.21 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.

### 8.7.22 Pre v0.0.4

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

CHAPTER 9

# Indices and tables

- genindex
- search

## A



## B



## C



## D



## E



## F

## Q



## R



## S



## T



## U



## V



## W



## Z