python-chess

Release 0.24.2

Jan 17, 2019

Contents

CHAPTER 1

Introduction

python-chess is a pure Python chess library with move generation, move validation and support for common formats. This is the Scholar's mate in python-chess:

```
>>> import chess

>>> board = chess.Board()

>>> board.legal_moves
<LegalMoveGenerator at ... (Nh3, Nf3, Nc3, Na3, h3, g3, f3, e3, d3, c3, ...)>
>>> chess.Move.from_uci("a8a1") in board.legal_moves
False

>>> 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

>>> board
Board('r1bqkb1r/pppp1Qpp/2n2n2/4p3/2B1P3/8/PPPP1PPP/RNB1K1NR b KQkq - 0 4')
```

CHAPTER 2

Documentation

CHAPTER 3

Features

- Supports Python 3.4+ and PyPy3.
- IPython/Jupyter Notebook integration. SVG rendering docs.

```
>>> board
```

- Chess variants: Standard, Chess960, Suicide, Giveaway, Atomic, King of the Hill, Racing Kings, Horde, Three-check, Crazyhouse. Variant docs.
- 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
```

- 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 as a draw (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(0x0000000000004040)
>>> chess.G2 in attackers
True
>>> print(attackers)
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . 1 .
. . . . . 1 .
```

- Parses and creates SAN representation of moves.

```
>>> board = chess.Board()
>>> board.san(chess.Move(chess.E2, chess.E4))
'e4'
>>> board.parse_san('Nf3')
Move.from_uci('g1f3')
```

(continues on next page)

(continued from previous page)

```
>>> board.variation_san([chess.Move.from_uci(m) for m in ["e2e4", "e7e5", "g1f3
↪"]])
'1. e4 e5 2. Nf3'
```

- 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
```

- Detects absolute pins and their directions.
- Reads 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()
```

- Reads and writes PGNs. Supports headers, comments, NAGs and a tree of variations. Docs.

```
>>> import chess.pgn

>>> with open("data/pgn/molinari-bordais-1979.pgn") as pgn:
...     first_game = chess.pgn.read_game(pgn)

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

>>> first_game.mainline()
```

(continues on next page)

(continued from previous page)

```
<Mainline at ... (1. e4 c5 2. c4 Nc6 3. Ne2 Nf6 4. Nbc3 Nb4 5. g3 Nd3#)>

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

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

```
>>> import chess.syzygy

>>> tablebase = chess.syzygy.open_tablebase("data/syzygy/regular")

>>> # 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")
>>> tablebase.probe_dtz(board)
-53

>>> tablebase.close()
```

- Communicate with UCI/XBoard engines. Based on `asyncio`. Docs.

```
>>> import chess.engine

>>> engine = chess.engine.SimpleEngine.popen_uci("stockfish")
>>> engine.id.get("name")
'Stockfish 10 64 POPCNT'

>>> board = chess.Board("1k1r4/pp1b1R2/3q2pp/4p3/2B5/4Q3/PPP2B2/2K5 b - - 0 1")
>>> limit = chess.engine.Limit(time=2.0)
>>> engine.play(board, limit)
<PlayResult at ... (move=d6d1, ponder=c1d1, info={...}, draw_offered=False)>

>>> engine.quit()
```

CHAPTER 4

Installing

Download and install the latest release:

```
pip install python-chess
```

CHAPTER 5

Selected use cases

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 GUI to play against UCI chess engines – http://johncheetham.com/projects/jcchess/
- deep learning for Crazyhouse – https://github.com/QueensGambit/CrazyAra
- a command-line PGN annotator – https://github.com/rpdelaney/python-chess-annotator
- a bot to play chess on Telegram – https://github.com/cxjdavin/tgchessbot
- an HTTP microservice to render board images – https://github.com/niklasf/web-boardimage
- a bridge between Lichess API and chess engines – https://github.com/careless25/lichess-bot
- a JIT compiled chess engine – https://github.com/SamRagusa/Batch-First

CHAPTER 6

Acknowledgements

Thanks to the Stockfish authors and thanks to Sam Tannous for publishing his approach to avoid rotated bitboards with direct lookup (PDF) alongside his GPL2+ engine Shatranj. Some move generation ideas are taken from these sources.

Thanks to Ronald de Man for his Syzygy endgame tablebases. The probing code in python-chess is very directly ported from his C probing code.

Thanks to Miguel A. Ballicora for his Gaviota tablebases. (I wish the generating code was free software.)

CHAPTER 7

License

python-chess is licensed under the GPL 3 (or any later version at your option). Check out LICENSE.txt for the full text.

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.`**`G8 = 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.**square**(*file_index*, *rank_index*)
Gets a square number by file and rank index.

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

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

chess.**square_distance**(*a*, *b*)
Gets the distance (i.e., the number of king steps) from square *a* to *b*.

chess.**square_mirror**(*square*)
Mirrors the square vertically.

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*, *drop=None*)
Represents a move from a square to a square and possibly the promotion piece type.

Drops and null moves are supported.

from_square
The source square.

to_square
The target square.

promotion
The promotion piece type or `one`.

drop
The drop piece type or `None`.

uci()
Gets an UCI string for the move.

For example, a move from a7 to a8 would be `a7a8` or `a7a8q` (if the latter is a promotion to a queen).

The UCI representation of a null move 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.

```
>>> import chess
>>>
>>> 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 for the standard chess starting position.

chess.**STARTING_BOARD_FEN = 'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR'**
The board part of the FEN for the standard chess starting position.

class chess.**Board**(*fen='rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1'*, ***, *chess960=False*)
A *BaseBoard* and additional information representing a chess 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 standard chess 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. Use *chess.Board.from_chess960_pos()* to create a board with one of the Chess960 starting positions.

It's safe to set *turn*, *castling_rights*, *ep_square*, *halfmove_clock* and *fullmove_number* directly.

turn
The side to move.

castling_rights
Bitmask of the rooks with castling rights.

To test for specific squares:

```
>>> import chess
>>>
>>> board = chess.Board()
>>> bool(board.castling_rights & chess.BB_H1)  # White can castle with the h1␣
↪rook
True
```

To add a specific square:

```
>>> board.castling_rights |= chess.BB_A1
```

Use *set_castling_fen()* to set multiple castling rights. Also see *has_castling_rights()*, *has_kingside_castling_rights()*, *has_queenside_castling_rights()*, *has_chess960_castling_rights()*, *clean_castling_rights()*.

ep_square
The potential en passant square on the third or sixth rank or `None`.

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.

promoted
A bitmask of pieces that have been promoted.

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

legal_moves = LegalMoveGenerator(self)
A dynamic list of legal moves.

```
>>> import chess
>>>
>>> board = chess.Board()
>>> board.legal_moves.count()
20
>>> bool(board.legal_moves)
True
>>> move = chess.Move.from_uci("g1f3")
>>> move in board.legal_moves
True
```

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

pseudo_legal_moves = PseudoLegalMoveGenerator(self)
A dynamic list of pseudo legal moves, much like the legal move list.

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.

Wraps `generate_pseudo_legal_moves()` and `is_pseudo_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 stack 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_board()
Clears the board.

clear_stack()
Clears the move stack.

root()
Returns a copy of the root position.

remove_piece_at(*square*)
Removes the piece from the given square. Returns the *Piece* or `None` if the square was already empty.

set_piece_at(*square*, *piece*, *promoted=False*)
Sets a piece at the given square.

An existing piece is replaced. Setting *piece* to `None` is equivalent to *remove_piece_at()*.

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_variant_end()
Checks if the game is over due to a special variant end condition.

Note, for example, that stalemate is not considered a variant-specific end condition (this method will return `False`), yet it can have a special **result** in suicide chess (any of *is_variant_loss()*, *is_variant_win()*, *is_variant_draw()* might return `True`).

is_variant_loss()
Checks if a special variant-specific loss condition is fulfilled.

is_variant_win()
Checks if a special variant-specific win condition is fulfilled.

is_variant_draw()
Checks if a special variant-specific drawing condition is fulfilled.

is_game_over(*, *claim_draw=False*)
Checks if the game is over due to *checkmate*, *stalemate*, *insufficient material*, the *seventyfive-move rule*, *fivefold repetition* or a *variant end condition*.

The game is not considered to be over by the *fifty-move rule* or *threefold repetition*, unless *claim_draw* is given. Note that checking the latter can be slow.

result(*, *claim_draw=False*)
Gets the game result.

`1-0`, `0-1` or `1/2-1/2` if the *game is over*. Otherwise, the result is undetermined: `*`.

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 1st 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 1st of July 2014 a game is automatically drawn (without a claim by one of the players) if a position occurs for the fifth time. Originally this had to occur on consecutive alternating moves, but this has since been revised.

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

Note that checking the latter can be slow.

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.

Note that checking this can be slow: In the worst case scenario every legal move has to be tested and the entire game has to be replayed because there is no incremental transposition table.

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

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

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

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

Warning Moves are not checked for legality.

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

Raises `IndexError` if the stack is empty.

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

Raises `IndexError` if the move stack is empty.

has_pseudo_legal_en_passant()
Checks if there is a pseudo-legal en passant capture.

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

fen(*, *shredder=False*, *en_passant='legal'*, *promoted=None*)
Gets a 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_rights*), the en passant square (*ep_square*), the *halfmove_clock* and the *fullmove_number*.

Parameters

- **shredder** – Use `castling_shredder_fen()` and encode castling rights by the file of the rook (like `HAha`) instead of the default `castling_xfen()` (like `KQkq`).
- **en_passant** – By default, only fully legal en passant squares are included (*has_legal_en_passant()*). Pass `fen` to strictly follow the FEN specification (always include the en passant square after a two-step pawn move) or `xfen` to follow the X-FEN specification (*has_pseudo_legal_en_passant()*).
- **promoted** – Mark promoted pieces like `Q~`. By default, this is only enabled in chess variants where this is relevant.

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

Raises `ValueError` if the FEN string is invalid.

set_castling_fen(*castling_fen*)
Sets castling rights from a string in FEN notation like `Qqk`.

Raises `ValueError` if the castling FEN is syntactically invalid.

set_board_fen(*fen*)
Parses a FEN and sets the board from it.

Raises `ValueError` if the FEN string is invalid.

set_piece_map(*pieces*)
Sets up the board from a dictionary of *pieces* by square index.

set_chess960_pos(*sharnagl*)
Sets up a Chess960 starting position given its index between 0 and 959. Also see *from_chess960_pos()*.

chess960_pos(*, *ignore_turn=False*, *ignore_castling=False*, *ignore_counters=True*)
Gets the Chess960 starting position index between 0 and 956 or `None` if the current position is not a Chess960 starting position.

By default white to move (**ignore_turn**) and full castling rights (**ignore_castling**) are required, but move counters (**ignore_counters**) are ignored.

epd(*, *shredder=False*, *en_passant='legal'*, *promoted=None*, ***operations*)
Gets an EPD representation of the current position.

See *fen()* for FEN formatting options (*shredder*, *ep_square* and *promoted*).

EPD operations can be given as keyword arguments. Supported operands are strings, integers, floats, moves, 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:

```
>>> import chess
>>>
>>> board = chess.Board()
>>> 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, `hmvc` and `fmvn` are used to set the half-move clock and the full-move number. Otherwise, `0` and `1` are used.

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

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.

lan(*move*)
Gets the long algebraic notation of the given move in the context of the current position.

variation_san(*variation*)
Given a sequence of moves, returns a string representing the sequence in standard algebraic notation (e.g., `1. e4 e5 2. Nf3 Nc6` or `37...Bg6 38. fxg6`).

The board will not be modified as a result of calling this.

Raises `ValueError` if any moves in the sequence are illegal.

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

Returns the move.

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

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

chess960 defaults to the mode of the board. Pass `True` to force Chess960 mode.

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

Supports both 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.

Returns the move.

Raises `ValueError` if the move is invalid or illegal in the current position (but not a null 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_zeroing(*move*)
Checks if the given pseudo-legal move is a capture or pawn move.

is_irreversible(*move*)
Checks if the given pseudo-legal move is irreversible.

In standard chess, pawn moves, captures and moves that destroy castling rights are irreversible.

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.

clean_castling_rights()
Returns valid castling rights filtered from *castling_rights*.

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`, `STATUS_EMPTY`, `STATUS_RACE_CHECK`, `STATUS_RACE_OVER`, `STATUS_RACE_MATERIAL`.

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.

mirror()
Returns a mirrored copy of the board.

The board is mirrored vertically and piece colors are swapped, so that the position is equivalent modulo color.

copy(*, *stack=True*)
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.

classmethod from_chess960_pos(*sharnagl*)
Creates a new board, initialized with a Chess960 starting position.

```
>>> import chess
>>> import random
>>>
>>> board = chess.Board.from_chess960_pos(random.randint(0, 959))
```

class chess.**BaseBoard**(*board_fen='rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR'*)
A board representing the position of chess pieces. See *Board* for a full board with move generation.

The board is initialized with the standard chess starting position, unless otherwise specified in the optional *board_fen* argument. If *board_fen* is `None`, an empty board is created.

clear_board()
Clears the board.

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.

king(*color*)
Finds the king square of the given side. Returns `None` if there is no king of that color.

In variants with king promotions, only non-promoted kings are considered.

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.

Returns a *set of squares*.

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 **not** considered attacked.

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

Pinned pieces still count as attackers.

Returns a *set of squares*.

pin(*color*, *square*)
Detects an absolute pin (and its direction) of the given square to the king of the given color.

```
>>> import chess
>>>
>>> board = chess.Board("rnb1k2r/ppp2ppp/5n2/3q4/1b1P4/2N5/PP3PPP/R1BQKBNR w␣
↪KQkq - 3 7")
>>> board.is_pinned(chess.WHITE, chess.C3)
True
>>> direction = board.pin(chess.WHITE, chess.C3)
>>> direction
SquareSet(0x0000000102040810)
>>> print(direction)
. . . . . . . .
. . . . . . . .
. . . . . . . .
1 . . . . . . .
. 1 . . . . . .
. . 1 . . . . .
. . . 1 . . . .
. . . . 1 . . .
```

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.

remove_piece_at(*square*)
Removes the piece from the given square. Returns the *Piece* or `None` if the square was already empty.

set_piece_at(*square*, *piece*, *promoted=False*)
Sets a piece at the given square.

An existing piece is replaced. Setting *piece* to `None` is equivalent to *remove_piece_at()*.

board_fen(*, *promoted=False*)
Gets the board FEN.

set_board_fen(*fen*)
Parses a FEN and sets the board from it.

Raises `ValueError` if the FEN string is invalid.

piece_map()
Gets a dictionary of *pieces* by square index.

set_piece_map(*pieces*)
Sets up the board from a dictionary of *pieces* by square index.

set_chess960_pos(*sharnagl*)
Sets up a Chess960 starting position given its index between 0 and 959. Also see *from_chess960_pos()*.

chess960_pos()
Gets the Chess960 starting position index between 0 and 959 or `None`.

unicode(*, *invert_color=False*, *borders=False*)
Returns a string representation of the board with Unicode pieces. Useful for pretty-printing to a terminal.

Parameters

- **invert_color** – Invert color of the Unicode pieces.
- **borders** – Show borders and a coordinate margin.

mirror()
Returns a mirrored copy of the board.

The board is mirrored vertically and piece colors are swapped, so that the position is equivalent modulo color.

copy()
Creates a copy of the board.

classmethod empty()
Creates a new empty board. Also see *clear_board()*.

classmethod from_chess960_pos(*sharnagl*)
Creates a new board, initialized with a Chess960 starting position.

```
>>> import chess
>>> import random
>>>
>>> board = chess.Board.from_chess960_pos(random.randint(0, 959))
```

8.1.7 Square sets

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

```
>>> import chess
>>>
>>> squares = chess.SquareSet(chess.BB_A8 | chess.BB_RANK_1)
>>> squares
SquareSet(0x01000000000000ff)
```

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

```
>>> len(squares)
9
```

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

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

```
>>> for square in squares:
...     # 0 -- chess.A1
...     # 1 -- chess.B1
...     # 2 -- chess.C1
...     # 3 -- chess.D1
...     # 4 -- chess.E1
...     # 5 -- chess.F1
...     # 6 -- chess.G1
...     # 7 -- chess.H1
...     # 56 -- chess.A8
...     print(square)
...
0
1
2
3
4
5
6
7
56
```

```
>>> list(squares)
[0, 1, 2, 3, 4, 5, 6, 7, 56]
```

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)
72057594037928191
```

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()*.

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

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

isdisjoint(*other*)
Test if the square sets are disjoint.

issubset(*other*)
Test if this square set is a subset of another.

issuperset(*other*)
Test if this square set is a superset of another.

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

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

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

Raises `KeyError` on an empty set.

clear()
Remove all elements from this set.

carry_rippler()
Iterator over the subsets of this set.

mirror()
Returns a vertically mirrored copy of this square set.

tolist()
Convert the set to a list of 64 bools.

classmethod from_square(*square*)
Creates a *SquareSet* from a single square.

```
>>> import chess
>>>
>>> chess.SquareSet.from_square(chess.A1) == chess.BB_A1
True
```

Common integer masks are:

chess.**BB_EMPTY = 0**

chess.**BB_ALL = 0xFFFFFFFFFFFFFFFF**

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]**

Other masks:

chess.**BB_LIGHT_SQUARES = 0x55AA55AA55AA55AA**

chess.**BB_DARK_SQUARES = 0xAA55AA55AA55AA55**

chess.**BB_BACKRANKS = chess.BB_RANK_1 | chess.BB_RANK_8**

chess.**BB_CORNERS = chess.BB_A1 | chess.BB_H1 | chess.BB_A8 | chess.BB_H8**

chess.**BB_CENTER = chess.BB_D4 | chess.BB_E4 | chess.BB_D5 | chess.BB_E5**

8.2 PGN parsing and writing

8.2.1 Parsing

chess.pgn.**read_game**(*handle*, *, *Visitor=<class 'chess.pgn.GameCreator'>*)
Reads a game from a file opened in text mode.

```
>>> import chess.pgn
>>>
>>> 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'
>>>
>>> # Iterate through all moves and play them on a board.
>>> board = first_game.board()
>>> for move in first_game.mainline_moves():
...     board.push(move)
...
>>> board
Board('4r3/6P1/2p2P1k/1p6/pP2p1R1/P1B5/2P2K2/3r4 b - - 0 45')
```

By using text mode, the parser does not need to handle encodings. It is the caller's responsibility to open the file with the correct encoding. PGN files are usually ASCII or UTF-8 encoded. So, the following should cover most relevant cases (ASCII, UTF-8, UTF-8 with BOM).

```
>>> pgn = open("data/pgn/kasparov-deep-blue-1997.pgn", encoding="utf-8-sig")
```

Use `StringIO` to parse games from a string.

```
>>> import io
>>>
>>> pgn = io.StringIO("1. e4 e5 2. Nf3 *")
>>> 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 PGN 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. Any exceptions are logged and collected in *Game.errors*. This behavior can be *overriden*.

Returns the parsed game or `None` if the end of file is reached.

8.2.2 Writing

If you want to export your game with all headers, comments and variations, you can do it like this:

```
>>> import chess
>>> import chess.pgn
>>>
>>> game = chess.pgn.Game()
>>> game.headers["Event"] = "Example"
>>> node = game.add_variation(chess.Move.from_uci("e2e4"))
>>> node = node.add_variation(chess.Move.from_uci("e7e5"))
>>> node.comment = "Comment"
>>>
>>> print(game)
[Event "Example"]
```

(continues on next page)

(continued from previous page)

```
[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=open("/dev/null", "w"), end="\n\n")
```

Use the *StringExporter()* or *FileExporter()* visitors if you need more control.

8.2.3 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 the *GameNode*) also holds general information, such as game headers.

class chess.pgn.**Game**(*headers=None*)

The root node of a game with extra information such as headers and the starting position. Also has all the other properties and methods of *GameNode*.

headers

A mapping of headers. By default, the following 7 headers are provided:

```
>>> import chess.pgn
>>>
>>> game = chess.pgn.Game()
>>> game.headers
Headers(Event='?', Site='?', Date='????.??.??', Round='?', White='?', Black='?
↪', Result='*')
```

errors

A list of errors (such as illegal or ambiguous moves) encountered while parsing the game.

board(*, *_cache=False*)

Gets the starting position of the game.

Unless the `FEN` header tag is set, this is the default starting position (for the `Variant`).

setup(*board*)

Sets up a specific starting position. This sets (or resets) the `FEN`, `SetUp`, and `Variant` header tags.

accept(*visitor*)

Traverses the game in PGN order using the given *visitor*. Returns the *visitor* result.

classmethod from_board(*board*)

Creates a game from the move stack of a *Board()*.

classmethod without_tag_roster()

Creates an empty game without the default 7 tag roster.

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 will never have NAGs.

comment = ''
A comment that goes behind the move leading to this node. Comments that occur before any moves 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()* checks this) can have a starting comment. The root node can not have a starting comment.

variations
A list of child nodes.

board(*, *_cache=True*)
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. See *chess.Board.san()*.

Do not call this on the root node.

uci(*, *chess960=None*)
Gets the UCI notation of the move leading to this node. See *chess.Board.uci()*.

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.

is_end()
Checks if this node is the last node in the current variation.

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.

For example, in `1. e4 e5 (1... c5 2. Nf3) 2. Nf3`, the node holding 1... c5 starts a variation.

is_mainline()
Checks if the node is in the mainline 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 is also in the main variation.

variation(*move*)
Gets a child node by either the move or the variation 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 a variation one up in the list of variations.

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

remove_variation(*move*)
Removes a variation.

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.

mainline()
Returns an iterator over the mainline starting after this node.

mainline_moves()
Returns an iterator over the main moves after this node.

add_line(*moves*, *, *comment=''*, *starting_comment=''*, *nags=()*)
Creates a sequence of child nodes for the given list of moves. Adds *comment* and *nags* to the last node of the line and returns it.

accept(*visitor*, *, *_parent_board=None*)
Traverses game nodes in PGN order using the given *visitor*. Starts with the move leading to this node. Returns the *visitor* result.

accept_subgame(*visitor*)
Traverses headers and game nodes in PGN order, as if the game was starting after this node. Returns the *visitor* result.

8.2.4 Visitors

Visitors are an advanced concept for game tree traversal.

class chess.pgn.**BaseVisitor**
Base class for visitors.

Use with *chess.pgn.Game.accept()* or *chess.pgn.GameNode.accept()* or *chess.pgn.read_game()*.

The methods are called in PGN order.

begin_game()
Called at the start of a game.

begin_headers()
Called before visiting game headers.

visit_header(*tagname*, *tagvalue*)
Called for each game header.

end_headers()
Called after visiting game headers.

parse_san(*board*, *san*)
When the visitor is used by a parser, this is called to parse a move in standard algebraic notation.

You can override the default implementation to work around specific quirks of your input format.

visit_move(*board*, *move*)
Called for each move.

board is the board state before the move. The board state must be restored before the traversal continues.

visit_board(*board*)
Called for the starting position of the game and after each move.

The board state must be restored before the traversal continues.

visit_comment(*comment*)
Called for each comment.

visit_nag(*nag*)
Called for each NAG.

begin_variation()
Called at the start of a new variation. It is not called for the mainline of the game.

end_variation()
Concludes a variation.

visit_result(*result*)
Called at the end of a game with the value from the `Result` header.

end_game()
Called at the end of a game.

result()
Called to get the result of the visitor. Defaults to `True`.

handle_error(*error*)
Called for encountered errors. Defaults to raising an exception.

The following visitors are readily available.

class chess.pgn.**GameCreator**
Creates a game model. Default visitor for *read_game()*.

handle_error(*error*)
Populates *chess.pgn.Game.errors* with encountered errors and logs them.

result()
Returns the visited *Game()*.

class chess.pgn.**HeaderCreator**
Collects headers into a dictionary.

class chess.pgn.**BoardCreator**
Returns the final position of the game. The mainline of the game is on the move stack.

class chess.pgn.**SkipVisitor**
Skips a game.

class chess.pgn.**StringExporter**(*, *columns=80*, *headers=True*, *comments=True*, *variations=True*)
Allows exporting a game as a string.

```
>>> import chess.pgn
>>>
>>> game = chess.pgn.Game()
>>>
>>> exporter = chess.pgn.StringExporter(headers=True, variations=True,␣
↪comments=True)
>>> pgn_string = game.accept(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 newline characters at the end of the string.

class chess.pgn.**FileExporter**(*handle*, ***, *columns=80*, *headers=True*, *comments=True*, *variations=True*)

Acts like a *StringExporter*, but games are written directly into a text file.

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

```
>>> import chess.pgn
>>>
>>> game = chess.pgn.Game()
>>>
>>> new_pgn = open("/dev/null", "w", encoding="utf-8")
>>> exporter = chess.pgn.FileExporter(new_pgn)
>>> game.accept(exporter)
```

8.2.5 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.

8.2.6 Skimming

These functions allow for quickly skimming games without fully parsing them.

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

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

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

```
>>> import chess.pgn
>>>
>>> pgn = open("data/pgn/kasparov-deep-blue-1997.pgn")
>>>
>>> kasparov_offsets = []
>>>
>>> while True:
...     offset = pgn.tell()
...
...     headers = chess.pgn.read_headers(pgn)
...     if headers is None:
...         break
...
...     if "Kasparov" in headers.get("White", "?"):
...         kasparov_offsets.append(offset)
```

Then it can later be seeked an parsed.

```
>>> for offset in kasparov_offsets:
...     pgn.seek(offset)
...     chess.pgn.read_game(pgn)
0
<Game at ... ('Garry Kasparov' vs. 'Deep Blue (Computer)', 1997.??.??)>
1436
<Game at ... ('Garry Kasparov' vs. 'Deep Blue (Computer)', 1997.??.??)>
3067
<Game at ... ('Garry Kasparov' vs. 'Deep Blue (Computer)', 1997.??.??)>
```

chess.pgn.**skip_game**(*handle*)
Skip a game. Returns `True` if a game was found and skipped.

8.3 Polyglot opening book reading

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

The following example opens a book to find all entries for the start position:

```
>>> import chess
>>> import chess.polyglot
>>>
>>> board = chess.Board()
>>>
>>> with chess.polyglot.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.

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

find(*board*, *, *minimum_weight=1*, *exclude_moves=()*)
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. Use get() if you prefer to get None instead of an exception.

choice(*board*, *, *minimum_weight=1*, *exclude_moves=()*, *random=<module 'random' from '/home/docs/checkouts/readthedocs.org/user_builds/python-chess/envs/latest/lib/python3.5/random.py'>*)
Uniformly selects a random entry for the given position.

Raises IndexError if no entries are found.

weighted_choice(*board*, *, *exclude_moves=()*, *random=<module 'random' from '/home/docs/checkouts/readthedocs.org/user_builds/python-chess/envs/latest/lib/python3.5/random.py'>*)
Selects a random entry for the given position, distributed by the weights of the entries.

Raises IndexError if no entries are found.

close()
Closes the reader.

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

chess.polyglot.**zobrist_hash**(*board*, *, *_hasher=<chess.polyglot.ZobristHasher object>*)
Calculates the Polyglot Zobrist hash of the position.

A Zobrist hash is an XOR 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.

8.4 Gaviota endgame tablebase probing

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_tablebase**(*directory*, *, *libgtb=None*, *LibraryLoader=<ctypes.LibraryLoader object>*)

Opens a collection of tables for probing.

First native access via the shared library libgtb is tried. 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.

Second, pure Python probing code is tried.

class chess.gaviota.**PythonTablebase**

Provides access to Gaviota tablebases using pure Python code.

add_directory(*directory*)

Adds *.gtb.cp4* tables from a directory. The relevant files are lazily opened when the tablebase is actually probed.

probe_dtm(*board*)

Probes for depth to mate information.

The absolute value is the number of half-moves until forced mate (or `0` in drawn positions). 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:

```
>>> import chess
>>> import chess.gaviota
>>>
>>> with chess.gaviota.open_tablebase("data/gaviota") as tablebase:
...     board = chess.Board("8/8/8/8/8/8/8/K2kr3 w - - 0 1")
...     print(tablebase.probe_dtm(board))
...
-10
```

Raises `KeyError` (or specifically `chess.gaviota.MissingTableError`) if the probe fails. Use `get_dtm()` if you prefer to get `None` instead of an exception.

Note that probing a corrupted table file is undefined behavior.

probe_wdl(*board*)

Probes for win/draw/loss-information.

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

```
>>> import chess
>>> import chess.gaviota
>>>
>>> with chess.gaviota.open_tablebase("data/gaviota") as tablebase:
...     board = chess.Board("8/4k3/8/B7/8/8/8/4K3 w - - 0 1")
...     print(tablebase.probe_wdl(board))
...
0
```

Raises `KeyError` (or specifically `chess.gaviota.MissingTableError`) if the probe fails. Use `get_wdl()` if you prefer to get `None` instead of an exception.

Note that probing a corrupted table file is undefined behavior.

close()

Closes all loaded tables.

8.4.1 libgtb

For faster access you can build and install a shared library. Otherwise the pure Python probing code is used.

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

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

Opens a collection of tables for probing using libgtb.

In most cases *open_tablebase()* should be used. Use this function only if you do not want to downgrade to pure Python tablebase probing.

Raises RuntimeError or OSError when libgtb can not be used.

class chess.gaviota.**NativeTablebase**(*libgtb*)

Provides access to Gaviota tablebases via the shared library libgtb. Has the same interface as *PythonTablebase*.

8.5 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 (and experimentally 7) pieces. Positions with castling rights are not included.

chess.syzygy.**open_tablebase**(*directory*, *, *load_wdl=True*, *load_dtz=True*, *max_fds=128*, *VariantBoard=<class 'chess.Board'>*)

Opens a collection of tables for probing. See *Tablebase*.

Note: Generally probing requires tablebase files for the specific material composition, **as well as** tablebase files with less pieces. This is important because 6-piece and 5-piece files are often distributed seperately, but are both required for 6-piece positions. Use *add_directory()* to load tables from additional directories.

---

class chess.syzygy.**Tablebase**(*, *max_fds=128*, *VariantBoard=<class 'chess.Board'>*)

Manages a collection of tablebase files for probing.

If *max_fds* is not None, will at most use *max_fds* open file descriptors at any given time. The least recently used tables are closed, if nescessary.

add_directory(*directory*, *, *load_wdl=True*, *load_dtz=True*)

Adds tables from a directory.

By default all available tables with the correct file names (e.g. WDL files like KQvKN.rtbw and DTZ files like KRBvK.rtbz) are added.

The relevant files are lazily opened when the tablebase is actually probed.

Returns the number of table files that were found.

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 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.

```
>>> import chess
>>> import chess.syzygy
>>>
>>> with chess.syzygy.open_tablebase("data/syzygy/regular") as tablebase:
...     board = chess.Board("8/2K5/4B3/3N4/8/8/4k3/8 b - - 0 1")
...     print(tablebase.probe_wdl(board))
...
-2
```

Raises `KeyError` (or specifically `chess.syzygy.MissingTableError`) if the position could not be found in the tablebase. Use `get_wdl()` if you prefer to get `None` instead of an exception.

Note that probing corrupted table files is undefined behavior.

probe_dtz(*board*)

Probes DTZ tables for distance to zero information.

Both DTZ and WDL tables are required in order to probe for DTZ.

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. More precisely:

WDL	DTZ	
-2	-100 <= n <= -1	Unconditional loss (assuming 50-move counter is zero), where a zeroing move can be forced in -n plies.
-1	n < -100	Loss, but draw under the 50-move rule. A zeroing move can be forced in -n plies or -n - 100 plies (if a later phase is responsible for the blessed loss).
0	0	Draw.
1	100 < n	Win, but draw under the 50-move rule. A zeroing move can be forced in n plies or n - 100 plies (if a later phase is responsible for the cursed win).
2	1 <= n <= 100	Unconditional win (assuming 50-move counter is zero), where a zeroing move can be forced in n plies.

The return value can be off by one: a return value -n can mean a losing zeroing move in in n + 1 plies and a return value +n can mean a winning zeroing move in n + 1 plies. This is guaranteed not to happen for positions exactly on the edge of the 50-move rule, so that (with some care) this never impacts the result of practical play.

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.

```
>>> import chess
>>> import chess.syzygy
>>>
>>> with chess.syzygy.open_tablebase("data/syzygy/regular") as tablebase:
...     board = chess.Board("8/2K5/4B3/3N4/8/8/4k3/8 b - - 0 1")
...     print(tablebase.probe_dtz(board))
...
-53
```

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

Raises `KeyError` (or specifically `chess.syzygy.MissingTableError`) if the position could not be found in the tablebase. Use `get_dtz()` if you prefer to get `None` instead of an exception.

Note that probing corrupted table files is undefined behavior.

close()
Closes all loaded tables.

8.6 UCI/XBoard engine communication (experimental)

UCI and XBoard are protocols for communicating with chess engines. This module implements an abstraction for playing moves and analysing positions with both kinds of engines.

warning This is an experimental module that may change in semver incompatible ways. Please weigh in on the design if the provided APIs do not cover your use case.

The intention is to eventually replace `chess.uci` and `chess.xboard`, but not before things have settled down and there has been a transition period.

The preferred way to use the API is with an asyncio event loop. The examples also show a synchronous wrapper *SimpleEngine* that automatically spawns an event loop in the background.

8.6.1 Playing

Example: Let Stockfish play against itself, 100 milliseconds per move.

```
import chess
import chess.engine

engine = chess.engine.SimpleEngine.popen_uci("stockfish")

board = chess.Board()
while not board.is_game_over():
    result = engine.play(board, chess.engine.Limit(time=0.100))
    board.push(result.move)

engine.quit()
```

```
import asyncio
import chess
import chess.engine

async def main():
    transport, engine = await chess.engine.popen_uci("stockfish")

    board = chess.Board()
    while not board.is_game_over():
        result = await engine.play(board, chess.engine.Limit(time=0.100))
        board.push(result.move)

    await engine.quit()
```

(continues on next page)

(continued from previous page)

```
asyncio.set_event_loop_policy(chess.engine.EventLoopPolicy())
asyncio.run(main())
```

class chess.engine.**EngineProtocol**

Protocol for communicating with a chess engine process.

coroutine play(*board*, *limit*, *, *game=None*, *info=<Info.NONE: 0>*, *ponder=False*, *root_moves=None*, *options={}*)

Play a position.

Parameters

- **board** – The position. The entire move stack will be sent to the engine.
- **limit** – An instance of *chess.engine.Limit* that determines when to stop thinking.
- **game** – Optional. An arbitrary object that identifies the game. Will automatically inform the engine if the object is not equal to the previous game (e.g. `ucinewgame`, `new`).
- **info** – Selects which additional information to retrieve from the engine. `INFO_NONE`, `INFO_BASE` (basic information that is trivial to obtain), `INFO_SCORE`, `INFO_PV`, `INFO_REFUTATION`, `INFO_CURRLINE`, `INFO_ALL` or any bitwise combination. Some overhead is associated with parsing extra information.
- **ponder** – Whether the engine should keep analysing in the background even after the result has been returned.
- **root_moves** – Optional. Consider only root moves from this list.
- **options** – Optional. A dictionary of engine options for the analysis. The previous configuration will be restored after the analysis is complete. You can permanently apply a configuration with *configure()*.

class chess.engine.**Limit**(*, *time=None*, *depth=None*, *nodes=None*, *mate=None*, *white_clock=None*, *black_clock=None*, *white_inc=None*, *black_inc=None*, *remaining_moves=None*)

Search termination condition.

time
Search exactly *time* seconds.

depth
Search *depth* ply only.

nodes
Search only a limited number of *nodes*.

mate
Search for a mate in *mate* moves.

white_clock
Time in seconds remaining for White.

black_clock
Time in seconds remaining for Black.

white_inc
Fisher increment for White, in seconds.

black_inc
Fisher increment for Black, in seconds.

remaining_moves
Number of moves to the next time control. If this is not set, but *white_clock* and *black_clock* are, then it is sudden death.

class chess.engine.**PlayResult**(*move*, *ponder*, *info=None*, *draw_offered=False*)
Returned by *chess.engine.EngineProtocol.play()*.

move
The best move accordig to the engine.

ponder
The response that the engine expects after *move*, or `None`.

info
A dictionary of extra information sent by the engine. Commonly used keys are: `score` (a *PovScore*), `pv` (a list of *Move* objects), `depth`, `seldepth`, `time` (in seconds), `nodes`, `nps`, `tbhits`, `multipv`.

Others: `currmove`, `currmovenumber`, `hashfull`, `cpuload`, `refutation`, `currline`, `ebf` and `string`.

draw_offered
Whether the engine offered a draw before moving.

8.6.2 Analysing and evaluating a position

Example:

```
import chess
import chess.engine

engine = chess.engine.SimpleEngine.popen_uci("stockfish")

board = chess.Board()
info = engine.analyse(board, chess.engine.Limit(time=0.100))
print("Score:", info["score"])
# Score: +20

board = chess.Board("r1bqkbnr/p1pp1ppp/1pn5/4p3/2B1P3/5Q2/PPPP1PPP/RNB1K1NR w KQkq -␣
↪2 4")
info = engine.analyse(board, chess.engine.Limit(depth=20))
print("Score:", info["score"])
# Score: #1

engine.quit()
```

```
import asyncio
import chess
import chess.engine

async def main():
    transport, engine = await chess.engine.popen_uci("stockfish")

    board = chess.Board()
    info = await engine.analyse(board, chess.engine.Limit(time=0.100))
    print(info["score"])
    # Score: +20

    board = chess.Board("r1bqkbnr/p1pp1ppp/1pn5/4p3/2B1P3/5Q2/PPPP1PPP/RNB1K1NR w␣
↪KQkq - 2 4")
```

(continues on next page)

(continued from previous page)

```
    info = await engine.analyse(board, chess.engine.Limit(depth=20))
    print(info["score"])
    # Score: #1

    await engine.quit()

asyncio.set_event_loop_policy(chess.engine.EventLoopPolicy())
asyncio.run(main())
```

class chess.engine.**EngineProtocol**

Protocol for communicating with a chess engine process.

coroutine analyse(*board*, *limit*, ***, *multipv=None*, *game=None*, *info=<Info.ALL: 31>*, *root_moves=None*, *options={}*)

Analyses a position and returns a dictionary of *information*.

Parameters

- **board** – The position to analyse. The entire move stack will be sent to the engine.
- **limit** – An instance of *chess.engine.Limit* that determines when to stop the analysis.
- **multipv** – Optional. Analyse multiple root moves. Will return a list of at most *multipv* dictionaries rather than just a single info dictionary.
- **game** – Optional. An arbitrary object that identifies the game. Will automatically inform the engine if the object is not equal to the previous game (e.g. `ucinewgame`, `new`).
- **info** – Selects which information to retrieve from the engine. `INFO_NONE`, `INFO_BASE` (basic information that is trivial to obtain), `INFO_SCORE`, `INFO_PV`, `INFO_REFUTATION`, `INFO_CURRLINE`, `INFO_ALL` or any bitwise combination. Some overhead is associated with parsing extra information.
- **root_moves** – Optional. Limit analysis to a list of root moves.
- **options** – Optional. A dictionary of engine options for the analysis. The previous configuration will be restored after the analysis is complete. You can permanently apply a configuration with *configure()*.

class chess.engine.**PovScore**(*relative*, *turn*)

A relative *Score* and the point of view.

relative

The relative *Score*.

turn

The point of view (`chess.WHITE` or `chess.BLACK`).

white()

Get the score from White's point of view.

black()

Get the score from Black's point of view.

pov(*color*)

Get the score from the point of view of the given *color*.

is_mate()

Tests if this is a mate score.

class chess.engine.**Score**

Evaluation of a position.

The score can be `Cp` (centi-pawns), `Mate` or `MateGiven`. A positive value indicates an advantage.

There is a total order defined on centi-pawn and mate scores.

```
>>> from chess.engine import Cp, Mate, MateGiven
>>>
>>> Mate(-0) < Mate(-1) < Cp(-50) < Cp(200) < Mate(4) < Mate(1) < MateGiven
True
```

Scores can be negated to change the point of view:

```
>>> -Cp(20)
Cp(-20)
```

```
>>> -Mate(-4)
Mate(+4)
```

```
>>> -Mate(0)
MateGiven
```

score(*, *mate_score=None*)

Returns the centi-pawn score as an integer or `None`.

You can optionally pass a large value to convert mate scores to centi-pawn scores.

```
>>> Cp(-300).score()
-300
>>> Mate(5).score() is None
True
>>> Mate(5).score(mate_score=100000)
99995
```

mate()

Returns the number of plies to mate, negative if we are getting mated, or `None`.

Warning This conflates `Mate(0)` (we lost) and `MateGiven` (we won) to `0`.

is_mate()

Tests if this is a mate score.

8.6.3 Indefinite or infinite analysis

Example: Stream information from the engine and stop on an arbitrary condition.

```
import chess
import chess.engine

engine = chess.engine.SimpleEngine.popen_uci("stockfish")

with engine.analysis(chess.Board()) as analysis:
    for info in analysis:
        print(info.get("score"), info.get("pv"))

        # Unusual stop condition.
```

(continues on next page)

(continued from previous page)

```
        if info.get("hashfull", 0) > 900:
            break

engine.quit()
```

```
import asyncio
import chess
import chess.engine

async def main():
    transport, engine = await chess.engine.popen_uci("stockfish")

    with await engine.analysis(chess.Board()) as analysis:
        async for info in analysis:
            print(info.get("score"), info.get("pv"))

            # Unusual stop condition.
            if info.get("hashfull", 0) > 900:
                break

    await engine.quit()

asyncio.set_event_loop_policy(chess.engine.EventLoopPolicy())
asyncio.run(main())
```

class chess.engine.**EngineProtocol**

Protocol for communicating with a chess engine process.

coroutine analysis(*board*, *limit=None*, ***, *multipv=None*, *game=None*, *info=<Info.ALL: 31>*, *root_moves=None*, *options={}*)

Starts analysing a position.

Parameters

- **board** – The position to analyse. The entire move stack will be sent to the engine.
- **limit** – Optional. An instance of *chess.engine.Limit* that determines when to stop the analysis. Analysis is infinite by default.
- **multipv** – Optional. Analyse multiple root moves.
- **game** – Optional. An arbitrary object that identifies the game. Will automatically inform the engine if the object is not equal to the previous game (e.g. `ucinewgame`, `new`).
- **info** – Selects which information to retrieve from the engine. `INFO_NONE`, `INFO_BASE` (basic information that is trivial to obtain), `INFO_SCORE`, `INFO_PV`, `INFO_REFUTATION`, `INFO_CURRLINE`, `INFO_ALL` or any bitwise combination. Some overhead is associated with parsing extra information.
- **root_moves** – Optional. Limit analysis to a list of root moves.
- **options** – Optional. A dictionary of engine options for the analysis. The previous configuration will be restored after the analysis is complete. You can permanently apply a configuration with *configure()*.

Returns *AnalysisResult*, a handle that allows asynchronously iterating over the information sent by the engine and stopping the the analysis at any time.

class chess.engine.**AnalysisResult**(*stop=None*)

Handle to ongoing engine analysis. Returned by *chess.engine.EngineProtocol.analysis()*.

Can be used to asynchronously iterate over information sent by the engine.

Automatically stops the analysis when used as a context manager.

info
A dictionary of aggregated information sent by the engine. This is actually an alias for `multipv[0]`.

multipv
A list of dictionaries with aggregated information sent by the engine. One item for each root move.

stop()
Stops the analysis as soon as possible.

coroutine wait()
Waits until the analysis is complete (or stopped).

coroutine next()
Waits for the next dictionary of information from the engine and returns it. Returns `None` if the analysis has been stopped and all information has been consumed.

It might be more convenient to use `async for info in analysis` (requires at least Python 3.5).

8.6.4 Options

`configure()`, `play()`, `analyse()` and `analysis()` accept a dictionary of options.

```
>>> import chess.engine
>>>
>>> engine = chess.engine.SimpleEngine.popen_uci("stockfish")
>>>
>>> # Check available options.
>>> engine.options["Hash"]
Option(name='Hash', type='spin', default=16, min=1, max=131072, var=[])
>>>
>>> # Set an option.
>>> engine.configure({"Hash": 32})
>>>
>>> # [...]
```

```
import asyncio
import chess.engine

async def main():
    transport, protocol = await chess.engine.popen_uci("stockfish")

    # Check available options.
    print(engine.options["Hash"])
    # Option(name='Hash', type='spin', default=16, min=1, max=131072, var=[])

    # Set an option.
    await engine.configure({"Hash": 32})

    # [...]

asyncio.set_event_loop_policy(chess.engine.EventLoopPolicy())
asyncio.run(main())
```

class chess.engine.**EngineProtocol**
Protocol for communicating with a chess engine process.

options
Dictionary of available options.

coroutine configure(*options*)
Configures global engine options.

Parameters **options** – A dictionary of engine options, where the keys are names of *options*. Do not set options that are managed automatically (*chess.engine.Option.is_managed()*).

class chess.engine.**Option**
Information about an available engine option.

name
The name of the option.

type
The type of the option.

type	UCI	CECP	value
check	X	X	`True` or `False`
button	X	X	`None`
reset		X	`None`
save		X	`None`
string	X	X	string without line breaks
file		X	string, interpreted as the path to a file
path		X	string, interpreted as the path to a directory

default
The default value of the option.

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

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

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

is_managed()
Some options are managed automatically: `UCI_Chess960`, `UCI_Variant`, `UCI_AnalyseMode`, `MultiPV`, `Ponder`.

8.6.5 Logging

Communication is logged with debug level on a logger named `chess.engine`. Debug logs are useful while troubleshooting. Please also provide them when submitting bug reports.

```
import logging

# Enable debug logging.
logging.basicConfig(level=logging.DEBUG)
```

8.6.6 Reference

class chess.engine.**EngineError**
Runtime error caused by a misbehaving engine or incorrect usage.

class chess.engine.**EngineTerminatedError**
The engine process exited unexpectedly.

coroutine chess.engine.**popen_uci**(*command*, *, *setpgrp=False*, ***popen_args*)
Spawns and initializes an UCI engine.

Parameters

- **command** – Path of the engine executable, or a list including the path and arguments.
- **setpgrp** – Open the engine process in a new process group. This will stop signals (such as keyboard interrupts) from propagating from the parent process. Defaults to `False`.
- **popen_args** – Additional arguments for popen. Do not set `stdin`, `stdout`, `bufsize` or `universal_newlines`.

Returns a subprocess transport and engine protocol pair.

coroutine chess.engine.**popen_xboard**(*command*, *, *setpgrp=False*, ***popen_args*)
Spawns and initializes an XBoard engine.

Parameters

- **command** – Path of the engine executable, or a list including the path and arguments.
- **setpgrp** – Open the engine process in a new process group. This will stop signals (such as keyboard interrupts) from propagating from the parent process. Defaults to `False`.
- **popen_args** – Additional arguments for popen. Do not set `stdin`, `stdout`, `bufsize` or `universal_newlines`.

Returns a subprocess transport and engine protocol pair.

class chess.engine.**EngineProtocol**
Protocol for communicating with a chess engine process.

returncode
Future: Exit code of the process.

id
Dictionary of information about the engine. Common keys are `name` and `author`.

coroutine ping()
Pings the engine and waits for a response. Used to ensure the engine is still alive and idle.

coroutine quit()
Asks the engine to shut down.

class chess.engine.**UciProtocol**
An implementation of the Universal Chess Interface protocol.

class chess.engine.**XBoardProtocol**
An implementation of the XBoard protocol (CECP).

class chess.engine.**SimpleEngine**(*transport*, *protocol*, *, *timeout=10.0*)
Synchronous wrapper around a transport and engine protocol pair. Provides the same methods and attributes as *EngineProtocol*, with blocking functions instead of coroutines.

You may not concurrently modify objects passed to any of the methods. Other than that *SimpleEngine* is thread-safe. When sending a new command to the engine, any previous running command will be cancelled as soon as possible.

Methods will raise `asyncio.TimeoutError` if an operation takes *timeout* seconds longer than expected (unless *timeout* is `None`).

Automatically closes the transport when used as a context manager.

close()
Closes the transport and the background event loop as soon as possible.

classmethod popen_uci(*command*, *, *timeout=10.0*, *debug=False*, *setpgrp=False*, ***popen_args*)
Spawns and initializes an UCI engine. Returns a *SimpleEngine* instance.

classmethod popen_xboard(*command*, *, *timeout=10.0*, *debug=False*, *setpgrp=False*, ***popen_args*)
Spawns and initializes an XBoard engine. Returns a *SimpleEngine* instance.

class chess.engine.**SimpleAnalysisResult**(*simple_engine*, *inner*)
Synchronous wrapper around *AnalysisResult*. Returned by `chess.engine.SimpleEngine.analysis()`.

chess.engine.**EventLoopPolicy**()
An event loop policy that ensures the event loop is capable of spawning and watching subprocesses, even when not running in the main thread.

Windows: Creates a `ProactorEventLoop`.

Unix: Creates a `SelectorEventLoop`. Child watchers are thread local. When not running on the main thread, the default child watchers use relatively slow polling to detect process termination. This does not affect communication.

8.7 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'>*, *setpgrp=False*, ***kwargs*)
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/bin/stockfish")
>>> engine.uci()
>>> engine.name
'Stockfish 8 64 POPCNT'
>>> engine.author
'T. Romstad, M. Costalba, J. Kiiski, G. Linscott'
```

Parameters

- **command** –
- **engine_cls** –
- **setpgrp** – Open the engine process in a new process group. This will stop signals (such as keyboard interrupts) from propagating from the parent process. Defaults to `False`.

chess.uci.**spur_spawn_engine**(*shell*, *command*, *, *engine_cls=<class 'chess.uci.Engine'>*)

Spawns a remote engine using a Spur shell.

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

class chess.uci.**Engine**(*, *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 of the engine. Conforming engines should send this as *id author* after the initial *uci* command.

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, like 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, like 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.

8.7.1 UCI commands

class chess.uci.**Engine**(*, *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 information to the GUI to help with the debugging. Usually, this mode is 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 engine's 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.

Rather than sending just the final FEN, the initial FEN and all moves leading up to the position will be sent. This will allow the engine to use the move history (for example to 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.

8.7.2 Asynchronous communication

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

```
>>> import chess.uci
>>>
>>> engine = chess.uci.popen_engine("stockfish")
>>>
>>> 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)
```

8.7.3 Info handler

class chess.uci.**Score**

A *cp* (centipawns) or *mate* score sent by an UCI engine.

cp

Evaluation in centipawns or `None`.

mate

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

class chess.uci.**InfoHandler**

Chess engines may send information about their calculations with the *info* command. An *InfoHandler* instance can be used to aggregate or react to this information.

```
>>> import chess.uci
>>>
>>> engine = chess.uci.popen_engine("stockfish")
>>>
>>> # Register a standard info handler.
>>> info_handler = chess.uci.InfoHandler()
>>> engine.info_handlers.append(info_handler)
>>>
>>> # Start a search.
>>> engine.position(chess.Board())
>>> engine.go(movetime=1000)
BestMove(bestmove=Move.from_uci('e2e4'), ponder=Move.from_uci('e7e6'))
>>>
>>> # Retrieve the score of the mainline (PV 1) after search is completed.
>>> # Note that the score is relative to the side to move.
>>> info_handler.info["score"][1]
Score(cp=34, mate=None)
```

See *info* for a way to access this dictionary in a thread-safe way during search.

If you want to be notified whenever new information is available, you would usually subclass the *InfoHandler* class:

```
>>> class MyHandler(chess.uci.InfoHandler):
...     def post_info(self):
...         # Called whenever a complete info line has been processed.
...         print(self.info)
...         super().post_info()  # Release the lock
```

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()`.

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

```
>>> # Wait a moment, then access a consistent snapshot.
>>> time.sleep(3)
>>> with info_handler:
...     if 1 in info_handler.info["score"]:
...         print("Score: ", info_handler.info["score"][1].cp)
...         print("Mate: ", info_handler.info["score"][1].mate)
Score: 34
Mate: None
```

depth (*x*)
Receives the search depth in plies.

seldepth (*x*)
Receives the selective search depth in plies.

time (*x*)
Receives a new time searched in milliseconds.

nodes (*x*)
Receives the number of nodes searched.

pv (*moves*)
Receives 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*)
Receives 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*)
Receives a new evaluation in *cp* (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 number means the engine thinks it will get mated.

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

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

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

The move comes directly from the engine, so the castling move representation depends on the *UCI_Chess960* option of the engine.

currmovenumber(*x*)
Receives a new current move number.

hashfull(*x*)
Receives new information about the hash table.

The hash table is *x* permill full.

nps(*x*)
Receives a new nodes per second (nps) statistic.

tbhits(*x*)
Receives a new information about the number of tablebase hits.

cpuload(*x*)
Receives a new *cpuload* information in permill.

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

refutation(*move*, *refuted_by*)
Receives 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*)
Receives a new snapshot of a line that a specific CPU is calculating.

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

ebf(*ebf*)
Receives the effective branching factor.

pre_info(*line*)
Receives new info lines before they are processed.

When subclassing, remember to call this method on the parent class to keep the locking intact.

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

When subclassing, remember to call this method on the parent class to keep the locking intact.

on_bestmove(*bestmove*, *ponder*)
Receives a new *bestmove* and a new *ponder* move.

on_go()
Notified when a *go* command is beeing sent.

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

8.7.4 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 text field.

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.

8.8 SVG rendering

The `chess.svg` module renders SVG Tiny images (mostly for IPython/Jupyter Notebook integration). The piece images by Colin M.L. Burnett are triple licensed under the GFDL, BSD and GPL.

`chess.svg.`**`piece`**(*piece*, *size=None*)
Renders the given *chess.Piece* as an SVG image.

```
>>> import chess
>>> import chess.svg
>>>
>>> chess.svg.piece(chess.Piece.from_symbol("R"))
```

`chess.svg.`**`board`**(*board=None*, ***, *squares=None*, *flipped=False*, *coordinates=True*, *lastmove=None*, *check=None*, *arrows=()*, *size=None*, *style=None*)
Renders a board with pieces and/or selected squares as an SVG image.

Parameters

- **board** – A *chess.BaseBoard* for a chessboard with pieces or `None` (the default) for a chessboard without pieces.
- **squares** – A *chess.SquareSet* with selected squares.
- **flipped** – Pass `True` to flip the board.
- **coordinates** – Pass `False` to disable coordinates in the margin.
- **lastmove** – A *chess.Move* to be highlighted.
- **check** – A square to be marked as check.
- **arrows** – A list of *Arrow* objects like `[chess.svg.Arrow(chess.E2, chess.E4)]` or a list of tuples like `[(chess.E2, chess.E4)]`. An arrow from a square pointing to the same square is drawn as a circle, like `[(chess.E2, chess.E2)]`.
- **size** – The size of the image in pixels (e.g., `400` for a 400 by 400 board) or `None` (the default) for no size limit.

- **style** – A CSS stylesheet to include in the SVG image.

```
>>> import chess
>>> import chess.svg
>>>
>>> board = chess.Board("8/8/8/8/4N3/8/8/8 w - - 0 1")
>>> squares = board.attacks(chess.E4)
>>> chess.svg.board(board=board, squares=squares)
```

class chess.svg.**Arrow**(*tail*, *head*, *, *color='#888'*)

Details of an arrow to be drawn.

tail

Start square of the arrow.

head

End square of the arrow.

color = "#888"

Arrow color.

8.9 Variants

python-chess supports several chess variants.

```
>>> import chess.variant
>>>
>>> board = chess.variant.GiveawayBoard()
```

```
>>> # General information about the variants
>>> type(board).uci_variant
'giveaway'
>>> type(board).starting_fen
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w - - 0 1'
```

See *chess.Board.is_variant_end()*, *is_variant_win()* *is_variant_draw()* *is_variant_loss()* for special variant end conditions and results.

Variant	Board class	UCI	Syzygy
Standard	*chess.Board*	chess	.rtbw, .rtbz
Suicide	chess.variant.SuicideBoard	suicide	.stbw, .stbz
Giveaway	chess.variant.GiveawayBoard	giveaway	.gtbw, .gtbz
Atomic	chess.variant.AtomicBoard	atomic	.atbw, .atbz
King of the Hill	chess.variant.KingOfTheHillBoard	kingofthehill	
Racing Kings	chess.variant.RacingKingsBoard	racingkings	
Horde	chess.variant.HordeBoard	horde	
Three-check	chess.variant.ThreeCheckBoard	3check	
Crazyhouse	chess.variant.CrazyhouseBoard	crazyhouse	

chess.variant.**find_variant**(*name*)

Looks for a variant board class by variant name.

8.9.1 Chess960

Chess960 is orthogonal to all other variants.

```
>>> chess.Board(chess960=True)
Board('rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1', chess960=True)
```

See *chess.BaseBoard.set_chess960_pos()*, *chess960_pos()*, and *from_chess960_pos()* for dealing with Chess960 starting positions.

8.9.2 UCI

Multi-Variant Stockfish and other engines have an `UCI_Variant` option. This is automatically managed.

```
>>> import chess.uci
>>> import chess.variant
>>>
>>> engine = chess.uci.popen_engine("stockfish")
>>> engine.uci()
>>>
>>> board = chess.variant.RacingKingsBoard()
>>> engine.position(board)
```

8.9.3 Syzygy

Syzygy tablebases are available for suicide, giveaway and atomic chess.

```
>>> import chess.syzygy
>>> import chess.variant
>>>
>>> tables = chess.syzygy.open_tablebase("data/syzygy", VariantBoard=chess.variant.
↪AtomicBoard)
```

8.10 Changelog for python-chess

8.10.1 New in v0.24.2

Bugfixes:

- *CrazyhouseBoard.root()* and *ThreeCheckBoard.root()* were not returning the correct pockets and number of remaining checks, respectively. Thanks @gbtami.
- *chess.pgn.skip_game()* now correctly skips PGN comments that contain line-breaks and PGN header tag notation.

Changes:

- Renamed *chess.pgn.GameModelCreator* to *GameCreator*. Alias kept in place and will be removed in a future release.
- Renamed *chess.engine* to *chess._engine*. Use re-exports from *chess.uci* or *chess.xboard*.
- Renamed *Board.stack* to *Board._stack*. Do not use this directly.

- Improved memory usage: *Board.legal_moves* and *Board.pseudo_legal_moves* no longer create reference cycles. PGN visitors can manage headers themselves.
- Removed previously deprecated items.

Features:

- Added *chess.pgn.BaseVisitor.visit_board()* and *chess.pgn.BoardCreator*.

8.10.2 New in v0.24.1, v0.23.11

Bugfixes:

- Fix *chess.Board.set_epd()* and *chess.Board.from_epd()* with semicolon in string operand. Thanks @jdart1.
- *chess.pgn.GameNode.uci()* was always raising an exception. Also included in v0.24.0.

8.10.3 New in v0.24.0

This release **drops support for Python 2**. The *0.23.x* branch will be maintained for one more month.

Changes:

- **Require Python 3.4.** Thanks @hugovk.
- No longer using extra pip features: *pip install python-chess[engine,gaviota]* is now *pip install python-chess*.
- Various keyword arguments can now be used as **keyword arguments only**.
- *chess.pgn.GameNode.accept()* now **also visits the move leading to that node**.
- *chess.pgn.GameModelCreator* now requires that *begin_game()* be called.
- *chess.pgn.scan_headers()* and *chess.pgn.scan_offsets()* have been removed. Instead the new functions *chess.pgn.read_headers()* and *chess.pgn.skip_game()* can be used for a similar purpose.
- *chess.syzygy*: Invalid magic headers now raise *IOError*. Previously they were only checked in an assertion. *type(board).{tbw_magic,tbz_magic,pawnless_tbw_magic,pawnless_tbz_magic}* are now byte literals.
- *board.status()* constants (*STATUS_*) are now typed using *enum.IntFlag*. Values remain unchanged.
- *chess.svg.Arrow* is no longer a *namedtuple*.
- *chess.PIECE_SYMBOLS[0]* and *chess.PIECE_NAMES[0]* are now *None* instead of empty strings.
- Performance optimizations:
  - *chess.pgn.Game.from_board()*,
  - *chess.square_name()*
  - Replace *collections.deque* with lists almost everywhere.
- Renamed symbols (aliases will be removed in the next release):
  - *chess.BB_VOID -> BB_EMPTY*
  - *chess.bswap() -> flip_vertical()*
  - *chess.pgn.GameNode.main_line() -> mainline_moves()*
  - *chess.pgn.GameNode.is_main_line() -> is_mainline()*
  - *chess.variant.BB_HILL -> chess.BB_CENTER*
  - *chess.syzygy.open_tablebases() -> open_tablebase()*

  - *chess.syzygy.Tablebases -> Tablebase*
  - *chess.syzygy.Tablebase.open_directory() -> add_directory()*
  - *chess.gaviota.open_tablebases() -> open_tablebase()*
  - *chess.gaviota.open_tablebases_native() -> open_tablebase_native()*
  - *chess.gaviota.NativeTablebases -> NativeTablebase*
  - *chess.gaviota.PythonTablebases -> PythonTablebase*
  - *chess.gaviota.NativeTablebase.open_directory() -> add_directory()*
  - *chess.gaviota.PythonTablebase.open_directory() -> add_directory()*

Bugfixes:

- The PGN parser now gives the visitor a chance to handle unknown chess variants and continue parsing.
- *chess.pgn.GameNode.uci()* was always raising an exception.

New features:

- *chess.SquareSet* now extends *collections.abc.MutableSet* and can be initialized from iterables.
- *board.apply_transform(f)* and *board.transform(f)* can apply bitboard transformations to a position. Examples: *chess.flip_{vertical,horizontal,diagonal,anti_diagonal}*.
- *chess.pgn.GameNode.mainline()* iterates over nodes of the mainline. Can also be used with *reversed()*. Reversal is now also supported for *chess.pgn.GameNode.mainline_moves()*.
- *chess.svg.Arrow(tail, head, color="#888")* gained an optional *color* argument.
- *chess.pgn.BaseVisitor.parse_san(board, san)* is used by parsers and can be overwritten to deal with non-standard input formats.
- *chess.pgn*: Visitors can advise the parser to skip games or variations by returning the special value *chess.pgn.SKIP* from *begin_game()*, *end_headers()* or *begin_variation()*. This is only a hint. The corresponding *end_game()* or *end_variation()* will still be called.
- Added *chess.svg.MARGIN*.

8.10.4 New in v0.23.10

Bugfixes:

- *chess.SquareSet* now correctly handles negative masks. Thanks @hasnul.
- *chess.pgn* now accepts *[Variant "chess 960"]* (with the space).

8.10.5 New in v0.23.9

Changes:

- Updated *Board.is_fivefold_repetition()*. FIDE rules have changed and the repetition no longer needs to occur on consecutive alternating moves. Thanks @LegionMammal978.

8.10.6 New in v0.23.8

Bugfixes:

- *chess.syzygy*: Correctly initialize wide DTZ map for experimental 7 piece table KRBBPvKQ.

8.10.7 New in v0.23.7

Bugfixes:

- Fixed *ThreeCheckBoard.mirror()* and *CrazyhouseBoard.mirror()*, which were previously resetting remaining checks and pockets respectively. Thanks @QueensGambit.

Changes:

- *Board.move_stack* is now guaranteed to be UCI compatible with respect to the representation of castling moves and *board.chess960*.
- Drop support for Python 3.3, which is long past end of life.
- *chess.uci*: The *position* command now manages *UCI_Chess960* and *UCI_Variant* automatically.
- *chess.uci*: The *position* command will now always send the entire history of moves from the root position.
- Various coding style fixes and improvements. Thanks @hugovk.

New features:

- Added *Board.root()*.

8.10.8 New in v0.23.6

Bugfixes:

- Gaviota: Fix Python based Gaviota tablebase probing when there are multiple en passant captures. Thanks @bjoernholzhauer.
- Syzygy: Fix DTZ for some mate in 1 positions. Similarly to the fix from v0.23.1 this is mostly cosmetic.
- Syzygy: Fix DTZ off-by-one in some 6 piece antichess positions with moves that threaten to force a capture. This is mostly cosmetic.

Changes:

- Let *uci.Engine.position()* send history of at least 8 moves if available. Previously it sent only moves that were relevant for repetition detection. This is mostly useful for Lc0. Once performance issues are solved, a future version will always send the entire history. Thanks @SashaMN and @Mk-Chan.
- Various documentation fixes and improvements.

New features:

- Added *polyglot.MemoryMappedReader.get(board, default=None)*.

8.10.9 New in v0.23.5

Bugfixes:

- Atomic chess: KNvKN is not insufficient material.

- Crazyhouse: Detect insufficient material. This can not happen unless the game was started with insufficient material.

Changes:

- Better error messages when parsing info from UCI engine fails.
- Better error message for *b.set_board_fen(b.fen())*.

8.10.10 New in v0.23.4

New features:

- XBoard: Support pondering. Thanks Manik Charan.
- UCI: Support unofficial *info ebf*.

Bugfixes:

- Implement 16 bit DTZ mapping, which is required for some of the longest 7 piece endgames.

8.10.11 New in v0.23.3

New features:

- XBoard: Support *variant*. Thanks gbtami.

8.10.12 New in v0.23.2

Bugfixes:

- XBoard: Handle multiple features and features with spaces. Thanks gbtami.
- XBoard: Ignore debug output prefixed with #. Thanks Dan Ravensloft and Manik Charan.

8.10.13 New in v0.23.1

Bugfixes:

- Fix DTZ in case of mate in 1. This is a cosmetic fix, as the previous behavior was only off by one (which is allowed by design).

8.10.14 New in v0.23.0

New features:

- Experimental support for 7 piece Syzygy tablebases.

Changes:

- *chess.syzygy.filenames()* was renamed to *tablenames()* and gained an optional *piece_count=6* argument.
- *chess.syzygy.normalize_filename()* was renamed to *normalize_tablename()*.
- The undocumented constructors of *chess.syzygy.WdlTable* and *chess.syzygy.DtzTable* have been changed.

8.10.15 New in v0.22.2

Bugfixes:

- In standard chess promoted pieces were incorrectly considered as distinguishable from normal pieces with regard to position equality and threefold repetition. Thanks to kn-sq-tb for reporting.

Changes:

- The PGN *game.headers* are now a custom mutable mapping that validates the validity of tag names.
- Basic attack and pin methods moved to *BaseBoard*.
- Documentation fixes and improvements.

New features:

- Added *Board.lan()* for long algebraic notation.

8.10.16 New in v0.22.1

New features:

- Added *Board.mirror()*, *SquareSet.mirror()* and *bswap()*.
- Added *chess.pgn.GameNode.accept_subgame()*.
- XBoard: Added *resign*, *analyze*, *exit*, *name*, *rating*, *computer*, *egtpath*, *pause*, *resume*. Completed option parsing.

Changes:

- *chess.pgn*: Accept FICS wilds without warning.
- XBoard: Inform engine about game results.

Bugfixes:

- *chess.pgn*: Allow games without movetext.
- XBoard: Fixed draw handling.

8.10.17 New in v0.22.0

Changes:

- *len(board.legal_moves)* **replaced by** *board.legal_moves.count()*. Previously *list(board.legal_moves)* was generating moves twice, resulting in a considerable slowdown. Thanks to Martin C. Doege for reporting.
- **Dropped Python 2.6 support.**
- XBoard: *offer_draw* renamed to *draw*.

New features:

- XBoard: Added *DrawHandler*.

8.10.18 New in v0.21.2

Changes:

- *chess.svg* is now fully SVG Tiny 1.2 compatible. Removed *chess.svg.DEFAULT_STYLE* which would from now on be always empty.

8.10.19 New in v0.21.1

Bugfixes:

- *Board.set_piece_at()* no longer shadows optional *promoted* argument from *BaseBoard*.
- Fixed *ThreeCheckBoard.is_irreversible()* and *ThreeCheckBoard._transposition_key()*.

New features:

- Added *Game.without_tag_roster()*. *chess.pgn.StringExporter()* can now handle games without any headers.
- XBoard: *white*, *black*, *random*, *nps*, *otim*, *undo*, *remove*. Thanks to Manik Charan.

Changes:

- Documentation fixes and tweaks by Boštjan Mejak.
- Changed unicode character for empty squares in *Board.unicode()*.

8.10.20 New in v0.21.0

Release yanked.

8.10.21 New in v0.20.1

Bugfixes:

- Fix arrow positioning on SVG boards.
- Documentation fixes and improvements, making most doctests runnable.

8.10.22 New in v0.20.0

Bugfixes:

- Some XBoard commands were not returning futures.
- Support semicolon comments in PGNs.

Changes:

- Changed FEN and EPD formatting options. It is now possible to include en passant squares in FEN and X-FEN style, or to include only strictly relevant en passant squares.
- Relax en passant square validation in *Board.set_fen()*.
- Ensure *is_en_passant()*, *is_capture()*, *is_zeroing()* and *is_irreversible()* strictly return bools.
- Accept *Z0* as a null move in PGNs.

New features:

- XBoard: Add *memory*, *core*, *stop* and *movenow* commands. Abstract *post*/*nopost*. Initial *FeatureMap* support. Support *usermove*.
- Added *Board.has_pseudo_legal_en_passant()*.
- Added *Board.piece_map()*.
- Added *SquareSet.carry_rippler()*.
- Factored out some (unstable) low level APIs: *BB_CORNERS*, *_carry_rippler()*, *_edges()*.

8.10.23 New in v0.19.0

New features:

- **Experimental XBoard engine support.** Thanks to Manik Charan and Cash Costello. Expect breaking changes in future releases.
- Added an undocumented *chess.polyglot.ZobristHasher* to make Zobrist hashing easier to extend.

Bugfixes:

- Merely pseudo-legal en passant does no longer count for repetitions.
- Fixed repetition detection in Three-Check and Crazyhouse. (Previously check counters and pockets were ignored.)
- Checking moves in Three-Check are now considered as irreversible by *ThreeCheckBoard.is_irreversible()*.
- *chess.Move.from_uci("")* was raising *IndexError* instead of *ValueError*. Thanks Jonny Balls.

Changes:

- *chess.syzygy.Tablebases* constructor no longer supports directly opening a directory. Use *chess.syzygy.open_tablebases()*.
- *chess.gaviota.PythonTablebases* and *NativeTablebases* constructors no longer support directly opening a directory. Use *chess.gaviota.open_tablebases()*.
- *chess.Board* instances are now compared by the position they represent, not by exact match of the internal data structures (or even move history).
- Relaxed castling right validation in Chess960: Kings/rooks of opposing sites are no longer required to be on the same file.
- Removed misnamed *Piece.__unicode__()* and *BaseBoard.__unicode__()*. Use *Piece.unicode_symbol()* and *BaseBoard.unicode()* instead.
- Changed *chess.SquareSet.__repr__()*.
- Support *[Variant "normal"]* in PGNs.
- *pip install python-chess[engine]* instead of *python-chess[uci]* (since the extra dependencies are required for both UCI and XBoard engines).
- Mixed documentation fixes and improvements.

8.10.24 New in v0.18.4

Changes:

- Support *[Variant "fischerandom"]* in PGNs for Cutechess compability. Thanks to Steve Maughan for reporting.

8.10.25 New in v0.18.3

Bugfixes:

- *chess.gaviota.NativeTablebases.get_dtm()* and *get_wdl()* were missing.

8.10.26 New in v0.18.2

Bugfixes:

- Fixed castling in atomic chess when there is a rank attack.
- The halfmove clock in Crazyhouse is no longer incremented unconditionally. *CrazyhouseBoard.is_zeroing(move)* now considers pawn moves and captures as zeroing. Added *Board.is_irreversible(move)* that can be used instead.
- Fixed an inconsistency where the *chess.pgn* tokenizer accepts long algebraic notation but *Board.parse_san()* did not.

Changes:

- Added more NAG constants in *chess.pgn*.

8.10.27 New in v0.18.1

Bugfixes:

- Crazyhouse drops were accepted as pseudo legal (and legal) even if the respective piece was not in the pocket.
- *CrazyhouseBoard.pop()* was failing to undo en passant moves.
- *CrazyhouseBoard.pop()* was always returning *None*.
- *Move.__copy__()* was failing to copy Crazyhouse drops.
- Fix ~ order (marker for promoted pieces) in FENs.
- Promoted pieces in Crazyhouse were not communicated with UCI engines.

Changes:

- *ThreeCheckBoard.uci_variant* changed from *threecheck* to *3check*.

8.10.28 New in v0.18.0

Bugfixes:

- Fixed *Board.parse_uci()* for crazyhouse drops. Thanks to Ryan Delaney.
- Fixed *AtomicBoard.is_insufficient_material()*.
- Fixed signature of *SuicideBoard.was_into_check()*.
- Explicitly close input and output streams when a *chess.uci.PopenProcess* terminates.
- The documentation of *Board.attackers()* was wrongly stating that en passant capturable pawns are considered attacked.

Changes:

- *chess.SquareSet* is no longer hashable (since it is mutable).

- Removed functions and constants deprecated in v0.17.0.
- Dropped *gmpy2* and *gmpy* as optional dependencies. They were no longer improving performance.
- Various tweaks and optimizations for 5% improvement in PGN parsing and perft speed. (Signature of *_is_safe* and *_ep_skewered* changed).
- Rewritten *chess.svg.board()* using *xml.etree*. No longer supports *pre* and *post*. Use an XML parser if you need to modify the SVG. Now only inserts actually used piece defintions.
- Untangled UCI process and engine instanciation, changing signatures of constructors and allowing arbitrary arguments to *subprocess.Popen*.
- Coding style and documentation improvements.

New features:

- *chess.svg.board()* now supports arrows. Thanks to @rheber for implementing this feature.
- Let *chess.uci.PopenEngine* consistently handle Ctrl+C across platforms and Python versions. *chess.uci.popen_engine()* now supports a *setpgrp* keyword argument to start the engine process in a new process group. Thanks to @dubiousjim.
- Added *board.king(color)* to find the (royal) king of a given side.
- SVGs now have *viewBox* and *chess.svg.board(size=None)* supports and defaults to *None* (i.e. scaling to the size of the container).

8.10.29 New in v0.17.0

Changes:

- Rewritten move generator, various performance tweaks, code simplications (500 lines removed) amounting to **doubled PGN parsing and perft speed**.
- Removed *board.generate_evasions()* and *board.generate_non_evasions()*.
- Removed *board.transpositions*. Transpositions are now counted on demand.
- *file_index()*, *rank_index()*, and *pop_count()* have been renamed to *square_file()*, *square_rank()* and *popcount()* respectively. Aliases will be removed in some future release.
- *STATUS_ILLEGAL_CHECK* has been renamed to *STATUS_RACE_CHECK*. The alias will be removed in a future release.
- Removed *DIAG_ATTACKS_NE*, *DIAG_ATTACKS_NW*, *RANK_ATTACKS* and *FILE_ATTACKS* as well as the corresponding masks. New attack tables *BB_DIAG_ATTACKS* (combined both diagonal tables), *BB_RANK_ATTACKS* and *BB_FILE_ATTACKS* are indexed by square instead of mask.
- *board.push()* no longer requires pseudo-legality.
- Documentation improvements.

Bugfixes:

- **Positions in variant end are now guaranteed to have no legal moves.** *board.is_variant_end()* has been added to test for special variant end conditions. Thanks to salvador-dali.
- *chess.svg*: Fixed a typo in the class names of black queens. Fixed fill color for black rooks and queens. Added SVG Tiny support. These combined changes fix display in a number of applications, including Jupyter Qt Console. Thanks to Alexander Meshcheryakov.
- *board.ep_square* was not consistently *None* instead of *0*.
- Detect invalid racing kings positions: *STATUS_RACE_OVER*, *STATUS_RACE_MATERIAL*.

- *SAN_REGEX*, *FEN_CASTLING_REGEX* and *TAG_REGEX* now try to match the entire string and no longer accept newlines.
- Fixed *Move.__hash__()* for drops.

New features:

- *board.remove_piece_at()* now returns the removed piece.
- Added *square_distance()* and *square_mirror()*.
- Added *msb()*, *lsb()*, *scan_reversed()* and *scan_forward()*.
- Added *BB_RAYS* and *BB_BETWEEN*.

8.10.30 New in v0.16.2

Changes:

- *board.move_stack* now contains the exact move objects added with *Board.push()* (instead of normalized copies for castling moves). This ensures they can be used with *Board.variation_san()* amongst others.
- *board.ep_square* is now *None* instead of *0* for no en passant square.
- *chess.svg*: Better vector graphics for knights. Thanks to ProgramFox.
- Documentation improvements.

8.10.31 New in v0.16.1

Bugfixes:

- Explosions in atomic chess were not destroying castling rights. Thanks to ProgramFOX for finding this issue.

8.10.32 New in v0.16.0

Bugfixes:

- *pin_mask()*, *pin()* and *is_pinned()* make more sense when already in check. Thanks to Ferdinand Mosca.

New features:

- **Variant support: Suicide, Giveaway, Atomic, King of the Hill, Racing Kings, Horde, Three-check, Crazyhouse.** *chess.Move* now supports drops.
- More fine grained dependencies. Use *pip install python-chess[uci,gaviota]* to install dependencies for the full feature set.
- Added *chess.STATUS_EMPTY* and *chess.STATUS_ILLEGAL_CHECK*.
- The *board.promoted* mask keeps track of promoted pieces.
- Optionally copy boards without the move stack: *board.copy(stack=False)*.
- *examples/bratko_kopec* now supports avoid move (am), variants and displays fractional scores immidiately. Thanks to Daniel Dugovic.
- *perft.py* rewritten with multi-threading support and moved to *examples/perft*.
- *chess.syzygy.dependencies()*, *chess.syzygy.all_dependencies()* to generate Syzygy tablebase dependencies.

Changes:

- **Endgame tablebase probing (Syzygy, Gaviota):** *probe_wdl()* **,** *probe_dtz()* **and** *probe_dtm()* **now raise** *KeyError* **or** *MissingTableError* **instead of returning** *None*. If you prefer getting *None* in case of an error use *get_wdl()*, *get_dtz()* and *get_dtm()*.
- *chess.pgn.BaseVisitor.result()* returns *True* by default and is no longer used by *chess.pgn.read_game()* if no game was found.
- Non-fast-forward update of the Git repository to reduce size (old binary test assets removed).
- *board.pop()* now uses a boardstate stack to undo moves.
- *uci.engine.position()* will send the move history only until the latest zeroing move.
- Optimize *board.clean_castling_rights()* and micro-optimizations improving PGN parser performance by around 20%.
- Syzygy tables now directly use the endgame name as hash keys.
- Improve test performance (especially on Travis CI).
- Documentation updates and improvements.

8.10.33 New in v0.15.4

New features:

- Highlight last move and checks when rendering board SVGs.

8.10.34 New in v0.15.3

Bugfixes:

- *pgn.Game.errors* was not populated as documented. Thanks to Ryan Delaney for reporting.

New features:

- Added *pgn.GameNode.add_line()* and *pgn.GameNode.main_line()* which make it easier to work with lists of moves as variations.

8.10.35 New in v0.15.2

Bugfixes:

- Fix a bug where *shift_right()* and *shift_2_right()* were producing integers larger than 64bit when shifting squares off the board. This is very similar to the bug fixed in v0.15.1. Thanks to piccoloprogrammatore for reporting.

8.10.36 New in v0.15.1

Bugfixes:

- Fix a bug where *shift_up_right()* and *shift_up_left()* were producing integers larger than 64bit when shifting squares off the board.

New features:

- Replaced __html__ with experimental SVG rendering for IPython.

8.10.37 New in v0.15.0

Changes:

- *chess.uci.Score* **no longer has** *upperbound* **and** *lowerbound* **attributes**. Previously these were always *False*.
- Significant improvements of move generation speed, around **2.3x faster PGN parsing**. Removed the following internal attributes and methods of the *Board* class: *attacks_valid*, *attacks_to*, *attacks_from*, *_pinned()*, *attacks_valid_stack*, *attacks_from_stack*, *attacks_to_stack*, *generate_attacks()*.
- UCI: Do not send *isready* directly after *go*. Though allowed by the UCI protocol specification it is just not nescessary and many engines were having trouble with this.
- Polyglot: Use less memory for uniform random choices from big opening books (reservoir sampling).
- Documentation improvements.

Bugfixes:

- Allow underscores in PGN header tags. Found and fixed by Bajusz Tamás.

New features:

- Added *Board.chess960_pos()* to identify the Chess960 starting position number of positions.
- Added *chess.BB_BACKRANKS* and *chess.BB_PAWN_ATTACKS*.

8.10.38 New in v0.14.1

Bugfixes:

- Backport Bugfix for Syzygy DTZ related to en-passant. See official-stockfish/Stockfish@6e2ca97d93812b2.

Changes:

- Added optional argument *max_fds=128* to *chess.syzygy.open_tablebases()*. An LRU cache is used to keep at most *max_fds* files open. This allows using many tables without running out of file descriptors. Previously all tables were opened at once.
- Syzygy and Gaviota now store absolute tablebase paths, in case you change the working directory of the process.
- The default implementation of *chess.uci.InfoHandler.score()* will no longer store score bounds in *info["score"]*, only real scores.
- Added *Board.set_chess960_pos()*.
- Documentation improvements.

8.10.39 New in v0.14.0

Changes:

- *Board.attacker_mask()* **has been renamed to** *Board.attackers_mask()* for consistency.
- **The signature of** *Board.generate_legal_moves()* **and** *Board.generate_pseudo_legal_moves()* **has been changed.** Previously it was possible to select piece types for selective move generation:

  Board.generate_legal_moves(castling=True, pawns=True, knights=True, bishops=True, rooks=True, queens=True, king=True)

  Now it is possible to select arbitrary sets of origin and target squares. *to_mask* uses the corresponding rook squares for castling moves.

Board.generate_legal_moves(from_mask=BB_ALL, to_mask=BB)

To generate all knight and queen moves do:

board.generate_legal_moves(board.knights | board.queens)

To generate only castling moves use:

Board.generate_castling_moves(from_mask=BB_ALL, to_mask=BB_ALL)

- Additional hardening has been added on top of the bugfix from v0.13.3. Diagonal skewers on the last double pawn move are now handled correctly, even though such positions can not be reached with a sequence of legal moves.
- *chess.syzygy* now uses the more efficient selective move generation.

New features:

- The following move generation methods have been added: *Board.generate_pseudo_legal_ep(from_mask=BB_ALL, to_mask=BB_ALL)*, *Board.generate_legal_ep(from_mask=BB_ALL, to_mask=BB_ALL)*, *Board.generate_pseudo_legal_captures(from_mask=BB_ALL, to_mask=BB_ALL)*, *Board.generate_legal_captures(from_mask=BB_ALL, to_mask=BB_ALL)*.

8.10.40 New in v0.13.3

This is a bugfix release for a move generation bug. Other than the bugfix itself there are only minimal fully backwardscompatible changes. You should update immediately.

Bugfixes:

- When capturing en passant, both the capturer and the captured pawn disappear from the fourth or fifth rank. If those pawns were covering a horizontal attack on the king, then capturing en passant should not have been legal.

  Board.generate_legal_moves() and *Board.is_into_check()* have been fixed.

  The same principle applies for diagonal skewers, but nothing has been done in this release: If the last double pawn move covers a diagonal attack, then the king would have already been in check.

  v0.14.0 adds additional hardening for all cases. It is recommended you upgrade to v0.14.0 as soon as you can deal with the non-backwards compatible changes.

Changes:

- *chess.uci* now uses *subprocess32* if applicable (and available). Additionally a lock is used to work around a race condition in Python 2, that can occur when spawning engines from multiple threads at the same time.
- Consistently handle tabs in UCI engine output.

8.10.41 New in v0.13.2

Changes:

- *chess.syzygy.open_tablebases()* now raises if the given directory does not exist.
- Allow visitors to handle invalid *FEN* tags in PGNs.
- Gaviota tablebase probing fails faster for piece counts > 5.

Minor new features:

- Added *chess.pgn.Game.from_board()*.

8.10.42 New in v0.13.1

Changes:

- Missing *SetUp* tags in PGNs are ignored.
- Incompatible comparisons on *chess.Piece*, *chess.Move*, *chess.Board* and *chess.SquareSet* now return *NotImplemented* instead of *False*.

Minor new features:

- Factored out basic board operations to *chess.BaseBoard*. This is inherited by *chess.Board* and extended with the usual move generation features.
- Added optional *claim_draw* argument to *chess.Base.is_game_over()*.
- Added *chess.Board.result(claim_draw=False)*.
- Allow *chess.Board.set_piece_at(square, None)*.
- Added *chess.SquareSet.from_square(square)*.

8.10.43 New in v0.13.0

- *chess.pgn.Game.export()* and *chess.pgn.GameNode.export()* have been removed and replaced with a new visitor concept.
- *chess.pgn.read_game()* no longer takes an *error_handler* argument. Errors are now logged. Use the new visitor concept to change this behaviour.

8.10.44 New in v0.12.5

Bugfixes:

- Context manager support for pure Python Gaviota probing code. Various documentation fixes for Gaviota probing. Thanks to Jürgen Précour for reporting.
- PGN variation start comments for variations on the very first move were assigned to the game. Thanks to Norbert Räcke for reporting.

8.10.45 New in v0.12.4

Bugfixes:

- Another en passant related Bugfix for pure Python Gaviota tablebase probing.

New features:

- Added *pgn.GameNode.is_end()*.

Changes:

- Big speedup for *pgn* module. Boards are cached less agressively. Board move stacks are copied faster.
- Added tox.ini to specify test suite and flake8 options.

8.10.46 New in v0.12.3

Bugfixes:

- Some invalid castling rights were silently ignored by *Board.set_fen()*. Now it is ensured information is stored for retrieval using *Board.status()*.

8.10.47 New in v0.12.2

Bugfixes:

- Some Gaviota probe results were incorrect for positions where black could capture en passant.

8.10.48 New in v0.12.1

Changes:

- Robust handling of invalid castling rights. You can also use the new method *Board.clean_castling_rights()* to get the subset of strictly valid castling rights.

8.10.49 New in v0.12.0

New features:

- Python 2.6 support. Patch by vdbergh.
- Pure Python Gaviota tablebase probing. Thanks to Jean-Noël Avila.

8.10.50 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.

8.10.51 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.

8.10.52 New in v0.10.1

Bugfixes:

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

8.10.53 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()*.

8.10.54 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.

8.10.55 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.

8.10.56 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__()*.

8.10.57 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.

8.10.58 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.

8.10.59 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.

8.10.60 New in v0.7.0

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

8.10.61 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()*.

8.10.62 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.

8.10.63 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.

8.10.64 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.

8.10.65 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.

8.10.66 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.

8.10.67 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.

8.10.68 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.

8.10.69 New in v0.1.0

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

8.10.70 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.

8.10.71 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

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z