python-chess

Release 1.9.3

unknown

CONTENTS

INTRODUCTION

python-chess is a chess library for Python, 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')
```

INSTALLING

Requires Python 3.7+. Download and install the latest release:

```
pip install chess
```

DOCUMENTATION

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

FEATURES

- Includes mypy typings.

- 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.outcome()
Outcome(termination=<Termination.CHECKMATE: 1>, winner=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
```

(continues on next page)

```
SquareSet(0x0000_0000_0000_4040)
>>> 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')
>>> 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

>>> 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()
<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")

>>> 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,
→resigned=False)>

>>> engine.quit()
```

SELECTED PROJECTS

If you like, share interesting things you are using python-chess for, for example:

	https://syzygy-tables.info/ A website to probe Syzygy endgame tablebases
	https://maiachess.com/ A human-like neural network chess engine
	clente/chess Oppinionated wrapper to use python-chess from the R programming language
	https://crazyara.org/ Deep learning for Crazyhouse
	http://johncheetham.com A GUI to play against UCI chess engines
	https://www.pettingzoo.ml A multi-agent reinforcement learning environment

- extensions to build engines (search and evaluation) – https://github.com/Mk-Chan/python-chess-engine-extensions

- a stand-alone chess computer based on DGT board – http://www.picochess.org/

- a bridge between Lichess API and chess engines – https://github.com/careless25/lichess-bot

- a command-line PGN annotator – https://github.com/rpdelaney/python-chess-annotator

- an HTTP microservice to render board images – https://github.com/niklasf/web-boardimage
- building a toy chess engine with alpha-beta pruning, piece-square tables, and move ordering – https://healeycodes.com/building-my-own-chess-engine/
- a JIT compiled chess engine – https://github.com/SamRagusa/Batch-First
- teaching Cognitive Science – https://jupyter.brynmawr.edu
- an Alexa skill to play blindfold chess – https://github.com/laynr/blindfold-chess
- a chessboard widget for PySide2 – https://github.com/H-a-y-k/hichesslib
- Django Rest Framework API for multiplayer chess – https://github.com/WorkShoft/capablanca-api
- a browser based PGN viewer written in PyScript – https://github.com/nmstoker/ChessMatchViewer

SIX

ACKNOWLEDGEMENTS

LICENSE

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

CONTENTS

8.1 Core

8.1.1 Colors

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

chess.**WHITE: chess.Color = True**

chess.**BLACK: chess.Color = False**

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

8.1.2 Piece types

chess.**PAWN: chess.PieceType = 1**

chess.**KNIGHT: chess.PieceType = 2**

chess.**BISHOP: chess.PieceType = 3**

chess.**ROOK: chess.PieceType = 4**

chess.**QUEEN: chess.PieceType = 5**

chess.**KING: chess.PieceType = 6**

chess.**piece_symbol**(*piece_type: chess.PieceType*) → str

chess.**piece_name**(*piece_type: chess.PieceType*) → str

8.1.3 Squares

chess.**A1: chess.Square = 0**

chess.**B1: chess.Square = 1**

and so on to

chess.**G8: chess.Square = 62**

chess.**H8: chess.Square = 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.`**`parse_square`**(*name: str*) → chess.Square
> Gets the square index for the given square *name* (e.g., `a1` returns `0`).
>
> > **Raises** `ValueError` if the square name is invalid.

`chess.`**`square_name`**(*square: chess.Square*) → str
> Gets the name of the square, like `a3`.

`chess.`**`square`**(*file_index: int*, *rank_index: int*) → chess.Square
> Gets a square number by file and rank index.

`chess.`**`square_file`**(*square: chess.Square*) → int
> Gets the file index of the square where `0` is the a-file.

`chess.`**`square_rank`**(*square: chess.Square*) → int
> Gets the rank index of the square where `0` is the first rank.

`chess.`**`square_distance`**(*a: chess.Square*, *b: chess.Square*) → int
> Gets the distance (i.e., the number of king steps) from square *a* to *b*.

`chess.`**`square_mirror`**(*square: chess.Square*) → chess.Square
> Mirrors the square vertically.

8.1.4 Pieces

class `chess.`**`Piece`**(*piece_type: chess.PieceType*, *color: chess.Color*)
> A piece with type and color.
>
> **`piece_type: chess.PieceType`**
> > The piece type.
>
> **`color: chess.Color`**
> > The piece color.
>
> **`symbol`**() → str
> > 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: bool = False*) → str
> > Gets the Unicode character for the piece.
>
> **classmethod** **`from_symbol`**(*symbol: str*) → *chess.Piece*
> > Creates a `Piece` instance from a piece symbol.
> >
> > > **Raises** `ValueError` if the symbol is invalid.

8.1.5 Moves

class `chess.`**`Move`**(*from_square: chess.Square*, *to_square: chess.Square*, *promotion: Optional[chess.PieceType] = None*, *drop: Optional[chess.PieceType] = None*)
Represents a move from a square to a square and possibly the promotion piece type.

Drops and null moves are supported.

> **`from_square: chess.Square`**
> > The source square.
>
> **`to_square: chess.Square`**
> > The target square.

promotion: Optional[chess.PieceType] = None
> The promotion piece type or `None`.

drop: Optional[chess.PieceType] = None
> The drop piece type or `None`.

uci() → str
> Gets a 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: str*) → *chess.Move*
> Parses a UCI string.
>
> > **Raises** `ValueError` if the UCI string is invalid.

classmethod null() → *chess.Move*
> 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: Optional[str] = 'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1', *, chess960: bool = False*)
> A *BaseBoard*, additional information representing a chess position, and a *move stack*.
>
> Provides *move generation*, validation, *parsing*, attack generation, `game end detection`, 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.

> **Warning:** It is possible to set up and work with invalid positions. In this case, *Board* implements a kind of "pseudo-chess" (useful to gracefully handle errors or to implement chess variants). Use *is_valid()* to detect invalid positions.

turn: chess.Color
> The side to move (`chess.WHITE` or `chess.BLACK`).

castling_rights: chess.Bitboard
> 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()*.

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

halfmove_clock: int
> The number of half-moves since the last capture or pawn move.

promoted: chess.Bitboard
> A bitmask of pieces that have been promoted.

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

ep_square: Optional[chess.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.

move_stack: List[*chess.Move*]
> The move stack. Use *Board.push()*, *Board.pop()*, *Board.peek()* and *Board.clear_stack()* for manipulation.

property legal_moves
> 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()`.

property pseudo_legal_moves
>   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()`.

reset() → None
>   Restores the starting position.

clear() → None
>   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() → None
>   Clears the board.
>
>   *Board* also clears the move stack.

clear_stack() → None
>   Clears the move stack.

root() → BoardT
>   Returns a copy of the root position.

ply() → int
>   Returns the number of half-moves since the start of the game, as indicated by *fullmove_number* and *turn*.
>
>   If moves have been pushed from the beginning, this is usually equal to `len(board.move_stack)`. But note that a board can be set up with arbitrary starting positions, and the stack can be cleared.

checkers() → *chess.SquareSet*
>   Gets the pieces currently giving check.
>
>   Returns a *set of squares*.

is_check() → bool
>   Tests if the current side to move is in check.

gives_check(*move:* chess.Move) → bool
>   Probes if the given move would put the opponent in check. The move must be at least pseudo-legal.

is_variant_end() → bool
>   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() → bool
>   Checks if the current side to move lost due to a variant-specific condition.

is_variant_win() → bool
>   Checks if the current side to move won due to a variant-specific condition.

is_variant_draw() → bool
>   Checks if a variant-specific drawing condition is fulfilled.

outcome(*, *claim_draw: bool* = *False*) → Optional[*chess.Outcome*]

   Checks if the game is over due to *checkmate*, *stalemate*, *insufficient material*, the *seventyfive-move rule*, *fivefold repetition*, or a *variant end condition*. Returns the *chess.Outcome* if the game has ended, otherwise None.

   Alternatively, use is_game_over() if you are not interested in who won the game and why.

   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.

is_checkmate() → bool

   Checks if the current position is a checkmate.

is_stalemate() → bool

   Checks if the current position is a stalemate.

is_insufficient_material() → bool

   Checks if neither side has sufficient winning material (*has_insufficient_material()*).

has_insufficient_material(*color: chess.Color*) → bool

   Checks if *color* has insufficient winning material.

   This is guaranteed to return False if *color* can still win the game.

   The converse does not necessarily hold: The implementation only looks at the material, including the colors of bishops, but not considering piece positions. So fortress positions or positions with forced lines may return False, even though there is no possible winning line.

is_seventyfive_moves() → bool

   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 greater than 150. Other means to end a game take precedence.

is_fivefold_repetition() → bool

   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() → bool

   Checks if the player to move can claim a draw by the fifty-move rule or by threefold repetition.

   Note that checking the latter can be slow.

is_fifty_moves() → bool

   Checks that the clock of halfmoves since the last capture or pawn move is greater or equal to 100, and that no other means of ending the game (like checkmate) take precedence.

can_claim_fifty_moves() → bool

   Checks if the player to move can claim a draw by the fifty-move rule.

   In addition to *is_fifty_moves()*, the fifty-move rule can also be claimed if there is a legal move that achieves this condition.

can_claim_threefold_repetition() → bool

   Checks if the player to move can claim a draw by threefold repetition.

   Draw by threefold repetition can be claimed if the position on the board occurred 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.

is_repetition(*count: int = 3*) → bool
> Checks if the current position has repeated 3 (or a given number of) times.
>
> Unlike *can_claim_threefold_repetition()*, this does not consider a repetition that can be played on the next move.
>
> Note that checking this can be slow: In the worst case, the entire game has to be replayed because there is no incremental transposition table.

push(*move:* chess.Move) → None
> 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. It is the caller's responsibility to ensure that the move is at least pseudo-legal or a null move.

pop() → *chess.Move*
> Restores the previous position and returns the last move from the stack.
>
> > **Raises** IndexError if the move stack is empty.

peek() → *chess.Move*
> Gets the last move from the move stack.
>
> > **Raises** IndexError if the move stack is empty.

find_move(*from_square: chess.Square, to_square: chess.Square, promotion: Optional[chess.PieceType] = None*) → *chess.Move*
> Finds a matching legal move for an origin square, a target square, and an optional promotion piece type.
>
> For pawn moves to the backrank, the promotion piece type defaults to *chess.QUEEN*, unless otherwise specified.
>
> Castling moves are normalized to king moves by two steps, except in Chess960.
>
> > **Raises** ValueError if no matching legal move is found.

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

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

fen(**, shredder: bool = False, en_passant: Literal[legal,* fen*, xfen] = 'legal', promoted: Optional[bool] = None*) → str
> 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 board 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: str*) → None
Parses a FEN and sets the position from it.

> **Raises** `ValueError` if syntactically invalid. Use *`is_valid()`* to detect invalid positions.

set_castling_fen(*castling_fen: str*) → None
Sets castling rights from a string in FEN notation like `Qqk`.

Also clears the move stack.

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

chess960_pos(**, ignore_turn: bool = False, ignore_castling: bool = False, ignore_counters: bool = True*) → Optional[int]
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: bool = False, en_passant: Literal[legal, fen, xfen] = 'legal', promoted: Optional[bool] = None, **operations: Union[None, str, int, float, chess.Move, Iterable[chess.Move]]*) → str
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, finite floats, legal moves and `None`. Additionally, the operation `pv` accepts a legal variation as a list of moves. The operations `am` and `bm` accept a list of legal moves in the current position.

The name of the field cannot be a lone dash and cannot contain spaces, newlines, carriage returns or tabs.

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

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

set_epd(*epd: str*) → Dict[str, Union[None, str, int, float, *chess.Move*, List[*chess.Move*]]]
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:* chess.Move) → str
> Gets the standard algebraic notation of the given move in the context of the current position.

lan(*move:* chess.Move) → str
> Gets the long algebraic notation of the given move in the context of the current position.

variation_san(*variation: Iterable[*chess.Move*]*) → str
> 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: str*) → *chess.Move*
> Uses the current position as the context to parse a move in standard algebraic notation and returns the corresponding move object.
>
> Ambiguous moves are rejected. Overspecified moves (including long algebraic notation) are accepted.
>
> The returned move is guaranteed to be either legal or a null move.
>
> > **Raises** `ValueError` if the SAN is invalid, illegal or ambiguous.

push_san(*san: str*) → *chess.Move*
> Parses a move in standard algebraic notation, makes the move and puts it onto the move stack.
>
> Returns the move.
>
> > **Raises** `ValueError` if neither legal nor a null move.

uci(*move:* chess.Move, *, *chess960: Optional[*bool*] = None*) → str
> Gets the UCI notation of the move.
>
> *chess960* defaults to the mode of the board. Pass `True` to force Chess960 mode.

parse_uci(*uci: str*) → *chess.Move*
> 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: str*) → *chess.Move*
> 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).

push_xboard(*san: str*) → *chess.Move*
> Parses a move in standard algebraic notation, makes the move and puts it onto the move stack.
>
> Returns the move.
>
> > **Raises** `ValueError` if neither legal nor a null move.

is_en_passant(*move:* chess.Move) → bool
> Checks if the given pseudo-legal move is an en passant capture.

is_capture(*move:* chess.Move) → bool
> Checks if the given pseudo-legal move is a capture.

is_zeroing(*move:* chess.Move) → bool
> Checks if the given pseudo-legal move is a capture or pawn move.

is_irreversible(*move:* chess.Move) → bool
> Checks if the given pseudo-legal move is irreversible.
>
> In standard chess, pawn moves, captures, moves that destroy castling rights and moves that cede en passant are irreversible.
>
> This method has false-negatives with forced lines. For example, a check that will force the king to lose castling rights is not considered irreversible. Only the actual king move is.

is_castling(*move:* chess.Move) → bool
> Checks if the given pseudo-legal move is a castling move.

is_kingside_castling(*move:* chess.Move) → bool
> Checks if the given pseudo-legal move is a kingside castling move.

is_queenside_castling(*move:* chess.Move) → bool
> Checks if the given pseudo-legal move is a queenside castling move.

clean_castling_rights() → chess.Bitboard
> Returns valid castling rights filtered from *castling_rights*.

has_castling_rights(*color: chess.Color*) → bool
> Checks if the given side has castling rights.

has_kingside_castling_rights(*color: chess.Color*) → bool
> Checks if the given side has kingside (that is h-side in Chess960) castling rights.

has_queenside_castling_rights(*color: chess.Color*) → bool
> Checks if the given side has queenside (that is a-side in Chess960) castling rights.

has_chess960_castling_rights() → bool
> Checks if there are castling rights that are only possible in Chess960.

status() → chess.Status
> Gets a bitmask of possible problems with the position.
>
> STATUS_VALID if all basic validity requirements are met. This does not imply that the position is actually reachable with a series of legal moves from the starting position.
>
> 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, STATUS_TOO_MANY_CHECKERS, STATUS_IMPOSSIBLE_CHECK.

is_valid() → bool
> Checks some basic validity requirements.
>
> See *status()* for details.

transform(*f: Callable[[chess.Bitboard], chess.Bitboard]*) → BoardT
> Returns a transformed copy of the board (without move stack) by applying a bitboard transformation function.
>
> Available transformations include chess.flip_vertical(), chess.flip_horizontal(), chess.flip_diagonal(), chess.flip_anti_diagonal(), chess.shift_down(), chess.shift_up(), chess.shift_left(), and chess.shift_right().

Alternatively, `apply_transform()` can be used to apply the transformation on the board.

mirror() → BoardT

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. Also swap the "en passant" square, castling rights and turn.

Alternatively, `apply_mirror()` can be used to mirror the board.

copy(*, *stack: Union[bool, int] = True*) → BoardT

Creates a copy of the board.

Defaults to copying the entire move stack. Alternatively, *stack* can be `False`, or an integer to copy a limited number of moves.

classmethod empty(*, *chess960: bool = False*) → BoardT

Creates a new empty board. Also see *clear()*.

classmethod from_epd(*epd: str*, *, *chess960: bool = False*) → Tuple[BoardT, Dict[str, Union[None, str, int, float, *chess.Move*, List[*chess.Move*]]]]

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(*scharnagl: int*) → BoardT

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: Optional[str] = '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.

reset_board() → None

Resets pieces to the starting position.

Board also resets the move stack, but not turn, castling rights and move counters. Use *chess.Board.reset()* to fully restore the starting position.

clear_board() → None

Clears the board.

Board also clears the move stack.

pieces(*piece_type: chess.PieceType*, *color: chess.Color*) → *chess.SquareSet*

Gets pieces of the given type and color.

Returns a *set of squares*.

piece_at(*square: chess.Square*) → Optional[*chess.Piece*]

Gets the *piece* at the given square.

piece_type_at(*square: chess.Square*) → Optional[chess.PieceType]

Gets the piece type at the given square.

color_at(*square: chess.Square*) → Optional[chess.Color]

Gets the color of the piece at the given square.

king (*color: chess.Color*) → Optional[chess.Square]
> 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: chess.Square*) → *chess.SquareSet*
> Gets the set of attacked squares from the 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: chess.Color*, *square: chess.Square*) → bool
> 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: chess.Color*, *square: chess.Square*) → *chess.SquareSet*
> Gets the set of attackers of the given color for the given square.
>
> Pinned pieces still count as attackers.
>
> Returns a *set of squares*.

pin (*color: chess.Color*, *square: chess.Square*) → *chess.SquareSet*
> 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(0x0000_0001_0204_0810)
>>> 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: chess.Color*, *square: chess.Square*) → bool
> Detects if the given square is pinned to the king of the given color.

remove_piece_at (*square: chess.Square*) → Optional[*chess.Piece*]
> Removes the piece from the given square. Returns the `Piece` or `None` if the square was already empty.
>
> `Board` also clears the move stack.

set_piece_at (*square: chess.Square*, *piece: Optional[chess.Piece]*, *promoted: bool = False*) → None
> Sets a piece at the given square.
>
> An existing piece is replaced. Setting *piece* to `None` is equivalent to `remove_piece_at()`.
>
> `Board` also clears the move stack.

board_fen(*, *promoted: Optional[bool] = False*) → str
> Gets the board FEN (e.g., rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR).

set_board_fen(*fen: str*) → None
> Parses *fen* and sets up the board, where *fen* is the board part of a FEN.
>
> *Board* also clears the move stack.
>
> > **Raises** ValueError if syntactically invalid.

piece_map(*, *mask: chess.Bitboard = 18446744073709551615*) → Dict[chess.Square, *chess.Piece*]
> Gets a dictionary of *pieces* by square index.

set_piece_map(*pieces: Mapping[chess.Square, chess.Piece]*) → None
> Sets up the board from a dictionary of *pieces* by square index.
>
> *Board* also clears the move stack.

set_chess960_pos(*scharnagl: int*) → None
> Sets up a Chess960 starting position given its index between 0 and 959. Also see *from_chess960_pos()*.

chess960_pos() → Optional[int]
> Gets the Chess960 starting position index between 0 and 959, or None.

unicode(*, *invert_color: bool = False*, *borders: bool = False*, *empty_square: str = ''*, *orientation: chess.Color = True*) → str
> 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.

transform(*f: Callable[[chess.Bitboard], chess.Bitboard]*) → BaseBoardT
> Returns a transformed copy of the board (without move stack) by applying a bitboard transformation function.
>
> Available transformations include chess.flip_vertical(), chess.flip_horizontal(), chess.flip_diagonal(), chess.flip_anti_diagonal(), chess.shift_down(), chess.shift_up(), chess.shift_left(), and chess.shift_right().
>
> Alternatively, apply_transform() can be used to apply the transformation on the board.

mirror() → BaseBoardT
> Returns a mirrored copy of the board (without move stack).
>
> The board is mirrored vertically and piece colors are swapped, so that the position is equivalent modulo color.
>
> Alternatively, apply_mirror() can be used to mirror the board.

copy() → BaseBoardT
> Creates a copy of the board.

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

classmethod from_chess960_pos(*scharnagl: int*) → BaseBoardT
> 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 Outcome

class chess.**Outcome**(*termination:* chess.Termination, *winner: Optional[chess.Color]*)

Information about the outcome of an ended game, usually obtained from *chess.Board.outcome()*.

termination: *chess.Termination*

The reason for the game to have ended.

winner: **Optional[chess.Color]**

The winning color or None if drawn.

result() → str

Returns 1-0, 0-1 or 1/2-1/2.

class chess.**Termination**(*value*)

Enum with reasons for a game to be over.

CHECKMATE = 1

See *chess.Board.is_checkmate()*.

STALEMATE = 2

See *chess.Board.is_stalemate()*.

INSUFFICIENT_MATERIAL = 3

See *chess.Board.is_insufficient_material()*.

SEVENTYFIVE_MOVES = 4

See *chess.Board.is_seventyfive_moves()*.

FIVEFOLD_REPETITION = 5

See *chess.Board.is_fivefold_repetition()*.

FIFTY_MOVES = 6

See *chess.Board.can_claim_fifty_moves()*.

THREEFOLD_REPETITION = 7

See *chess.Board.can_claim_threefold_repetition()*.

VARIANT_WIN = 8

See *chess.Board.is_variant_win()*.

VARIANT_LOSS = 9

See *chess.Board.is_variant_loss()*.

VARIANT_DRAW = 10

See *chess.Board.is_variant_draw()*.

8.1.8 Square sets

class chess.**SquareSet** (*squares: chess.IntoSquareSet = 0*)

A set of squares.

```
>>> import chess
>>>
>>> squares = chess.SquareSet([chess.A8, chess.A1])
>>> squares
SquareSet(0x0100_0000_0000_0001)
```

```
>>> squares = chess.SquareSet(chess.BB_A8 | chess.BB_RANK_1)
>>> squares
SquareSet(0x0100_0000_0000_00ff)
```

```
>>> 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: chess.Square*) → None
  Adds a square to the set.

discard(*square: chess.Square*) → None
  Discards a square from the set.

isdisjoint(*other: chess.IntoSquareSet*) → bool
  Tests if the square sets are disjoint.

issubset(*other: chess.IntoSquareSet*) → bool
  Tests if this square set is a subset of another.

issuperset(*other: chess.IntoSquareSet*) → bool
  Tests if this square set is a superset of another.

remove(*square: chess.Square*) → None
  Removes a square from the set.

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

pop() → chess.Square
  Removes and returns a square from the set.

  Raises KeyError if the set is empty.

clear() → None
  Removes all elements from this set.

carry_rippler() → Iterator[chess.Bitboard]
  Iterator over the subsets of this set.

mirror() → *chess.SquareSet*
  Returns a vertically mirrored copy of this square set.

tolist() → List[bool]
  Converts the set to a list of 64 bools.

classmethod ray(*a: chess.Square*, *b: chess.Square*) → *chess.SquareSet*
  All squares on the rank, file or diagonal with the two squares, if they are aligned.

```
>>> import chess
>>>
>>> print(chess.SquareSet.ray(chess.E2, chess.B5))
. . . . . . . .
. . . . . . . .
1 . . . . . . .
. 1 . . . . . .
```

(continues on next page)

```
. . 1 . . . . .
. . . 1 . . . .
. . . . 1 . . .
. . . . . 1 . .
```

classmethod between(*a: chess.Square*, *b: chess.Square*) → *chess.SquareSet*

All squares on the rank, file or diagonal between the two squares (bounds not included), if they are aligned.

```python
>>> import chess
>>>
>>> print(chess.SquareSet.between(chess.E2, chess.B5))
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . 1 . . . . .
. . . 1 . . . .
. . . . . . . .
. . . . . . . .
```

classmethod from_square(*square: chess.Square*) → *chess.SquareSet*

Creates a *SquareSet* from a single square.

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

Common integer masks are:

chess.**BB_EMPTY: chess.Bitboard = 0**

chess.**BB_ALL: chess.Bitboard = 0xFFFF_FFFF_FFFF_FFFF**

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: chess.Bitboard = 0x55AA_55AA_55AA_55AA**

chess.**BB_DARK_SQUARES: chess.Bitboard = 0xAA55_AA55_AA55_AA55**

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: TextIO*) → Optional[*chess.pgn.Game*]

chess.pgn.**read_game**(*handle: TextIO*, *, *Visitor: Callable[], chess.pgn.BaseVisitor[ResultT]]*) → Optional[ResultT]

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, sometimes with BOM (which this parser automatically ignores). See `open()` for options to deal with encoding errors.

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

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 seven 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. By default, any exceptions are logged and collected in `Game.errors`. This behavior can be *overridden*.

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"]
[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. Conceptually each node represents a position of the game. The tree consists of one root node (*Game*, also holding game headers) and many child nodes (*ChildNode*). Both extend *GameNode*.

class chess.pgn.**GameNode**(**, comment: str = ''*)

    parent: Optional[*chess.pgn.GameNode*]
        The parent node or None if this is the root node of the game.

    move: Optional[*chess.Move*]
        The move leading to this node or None if this is the root node of the game.

    variations: List[*chess.pgn.ChildNode*]
        A list of child nodes.

    comment: str
        A comment that goes behind the move leading to this node. Comments that occur before any moves are assigned to the root node.

    abstract board() → *chess.Board*
        Gets a board with the position of the node.

        For the root node, this is the default starting position (for the Variant) unless the FEN header tag is set.

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

abstract ply() → int
> Returns the number of half-moves up to this node, as indicated by fullmove number and turn of the position. See *chess.Board.ply()*.
>
> Usually this is equal to the number of parent nodes, but it may be more if the game was started from a custom position.

turn() → chess.Color
> Gets the color to move at this node. See *chess.Board.turn*.

game() → *chess.pgn.Game*
> Gets the root node, i.e., the game.

end() → *chess.pgn.GameNode*
> Follows the main variation to the end and returns the last node.

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

starts_variation() → bool
> 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() → bool
> Checks if the node is in the mainline of the game.

is_main_variation() → bool
> 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: Union[int, chess.Move, chess.pgn.GameNode]*) → *chess.pgn.ChildNode*
> Gets a child node by either the move or the variation index.

has_variation(*move: Union[int, chess.Move, chess.pgn.GameNode]*) → bool
> Checks if this node has the given variation.

promote_to_main(*move: Union[int, chess.Move, chess.pgn.GameNode]*) → None
> Promotes the given *move* to the main variation.

promote(*move: Union[int, chess.Move, chess.pgn.GameNode]*) → None
> Moves a variation one up in the list of variations.

demote(*move: Union[int, chess.Move, chess.pgn.GameNode]*) → None
> Moves a variation one down in the list of variations.

remove_variation(*move: Union[int, chess.Move, chess.pgn.GameNode]*) → None
> Removes a variation.

add_variation(*move:* chess.Move, *, comment: str = '', starting_comment: str = '', nags: Iterable[int] = []*) → *chess.pgn.ChildNode*
> Creates a child node with the given attributes.

add_main_variation(*move:* chess.Move, *, comment: str = '', nags: Iterable[int] = []*) → *chess.pgn.ChildNode*
> Creates a child node with the given attributes and promotes it to the main variation.

next() → Optional[*chess.pgn.ChildNode*]
> Returns the first node of the mainline after this node, or None if this node does not have any children.

mainline() → chess.pgn.Mainline[*chess.pgn.ChildNode*]
> Returns an iterable over the mainline starting after this node.

---

mainline_moves() → chess.pgn.Mainline[*chess.Move*]
 Returns an iterable over the main moves after this node.

add_line(*moves: Iterable[*chess.Move*], *, comment: *str* = '', starting_comment: *str* = '', nags: Iterable[*int*] = []*) → *chess.pgn.GameNode*
 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.

eval() → Optional[*chess.engine.PovScore*]
 Parses the first valid [%eval ...] annotation in the comment of this node, if any.

eval_depth() → Optional[int]
 Parses the first valid [%eval ...] annotation in the comment of this node and returns the corresponding depth, if any.

set_eval(*score: Optional[*chess.engine.PovScore*], depth: Optional[*int*] = None*) → None
 Replaces the first valid [%eval ...] annotation in the comment of this node or adds a new one.

arrows() → List[*chess.svg.Arrow*]
 Parses all [%csl ...] and [%cal ...] annotations in the comment of this node.

 Returns a list of *arrows*.

set_arrows(*arrows: Iterable[Union[*chess.svg.Arrow*, Tuple[chess.Square, chess.Square]]]*) → None
 Replaces all valid [%csl ...] and [%cal ...] annotations in the comment of this node or adds new ones.

clock() → Optional[float]
 Parses the first valid [%clk ...] annotation in the comment of this node, if any.

 Returns the player's remaining time to the next time control after this move, in seconds.

set_clock(*seconds: Optional[*float*]*) → None
 Replaces the first valid [%clk ...] annotation in the comment of this node or adds a new one.

emt() → Optional[float]
 Parses the first valid [%emt ...] annotation in the comment of this node, if any.

 Returns the player's elapsed move time use for the comment of this move, in seconds.

set_emt(*seconds: Optional[*float*]*) → None
 Replaces the first valid [%emt ...] annotation in the comment of this node or adds a new one.

abstract accept(*visitor: chess.pgn.BaseVisitor[*ResultT*]*) → ResultT
 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: chess.pgn.BaseVisitor[*ResultT*]*) → ResultT
 Traverses headers and game nodes in PGN order, as if the game was starting after this node. Returns the *visitor* result.

class chess.pgn.**Game**(*headers: Optional[Union[Mapping[*str*, *str*], Iterable[Tuple[*str*, *str*]]]] = None*)
 The root node of a game with extra information such as headers and the starting position. Extends *GameNode*.

headers: chess.pgn.Headers
 A mapping of headers. By default, the following 7 headers are provided (Seven Tag Roster):

```
>>> import chess.pgn
>>>
>>> game = chess.pgn.Game()
```

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

errors: **List[Exception]**
    A list of errors (such as illegal or ambiguous moves) encountered while parsing the game.

setup(*board: Union[chess.Board, str]*) → None
    Sets up a specific starting position. This sets (or resets) the FEN, SetUp, and Variant header tags.

accept(*visitor:* chess.pgn.BaseVisitor*[ResultT]*) → ResultT
    Traverses the game in PGN order using the given *visitor*. Returns the *visitor* result.

classmethod from_board(*board:* chess.Board) → GameT
    Creates a game from the move stack of a *Board()*.

classmethod without_tag_roster() → GameT
    Creates an empty game without the default Seven Tag Roster.

class chess.pgn.**ChildNode**(*parent:* chess.pgn.GameNode, *move:* chess.Move, **, comment: str = '',
                                        starting_comment: str = '', nags: Iterable[int] = []*)
A child node of a game, with the move leading to it. Extends *GameNode*.

nags: **Set[int]**
    A set of NAGs as integers. NAGs always go behind a move, so the root node of the game will never have NAGs.

parent: *chess.pgn.GameNode*
    The parent node.

move: *chess.Move*
    The move leading to this node.

starting_comment: **str**
    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.

san() → str
    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: Optional[bool] = None*) → str
    Gets the UCI notation of the move leading to this node. See *chess.Board.uci()*.

    Do not call this on the root node.

end() → *chess.pgn.ChildNode*
    Follows the main variation to the end and returns the last node.

8.2.4 Visitors

Visitors are an advanced concept for game tree traversal.

class chess.pgn.**BaseVisitor**(**args*, ***kwds*)

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() → Optional[chess.pgn.SkipType]

Called at the start of a game.

begin_headers() → Optional[chess.pgn.Headers]

Called before visiting game headers.

visit_header(*tagname: str*, *tagvalue: str*) → None

Called for each game header.

end_headers() → Optional[chess.pgn.SkipType]

Called after visiting game headers.

parse_san(*board:* chess.Board, *san: str*) → *chess.Move*

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.

Deprecated since version 1.1: This method is very limited, because it is only called on moves that the parser recognizes in the first place. Instead of adding workarounds here, please report common quirks so that they can be handled for everyone.

visit_move(*board:* chess.Board, *move:* chess.Move) → None

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:* chess.Board) → None

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: str*) → None

Called for each comment.

visit_nag(*nag: int*) → None

Called for each NAG.

begin_variation() → Optional[chess.pgn.SkipType]

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

end_variation() → None

Concludes a variation.

visit_result(*result: str*) → None

Called at the end of a game with the value from the Result header.

end_game() → None

Called at the end of a game.

abstract result() → ResultT

Called to get the result of the visitor.

handle_error(*error: Exception*) → None
    Called for encountered errors. Defaults to raising an exception.

The following visitors are readily available.

class chess.pgn.**GameBuilder**
class chess.pgn.**GameBuilder**(**, Game: Type[GameT]*)
    Creates a game model. Default visitor for `read_game()`.

    handle_error(*error: Exception*) → None
        Populates `chess.pgn.Game.errors` with encountered errors and logs them.

        You can silence the log and handle errors yourself after parsing:

```
>>> import chess.pgn
>>> import logging
>>>
>>> logging.getLogger("chess.pgn").setLevel(logging.CRITICAL)
>>>
>>> pgn = open("data/pgn/kasparov-deep-blue-1997.pgn")
>>>
>>> game = chess.pgn.read_game(pgn)
>>> game.errors  # List of exceptions
[]
```

        You can also override this method to hook into error handling:

```
>>> import chess.pgn
>>>
>>> class MyGameBuilder(chess.pgn.GameBuilder):
>>>     def handle_error(self, error: Exception) -> None:
>>>         pass  # Ignore error
>>>
>>> pgn = open("data/pgn/kasparov-deep-blue-1997.pgn")
>>>
>>> game = chess.pgn.read_game(pgn, Visitor=MyGameBuilder)
```

    result() → GameT
        Returns the visited `Game()`.

class chess.pgn.**HeadersBuilder**
class chess.pgn.**HeadersBuilder**(**, Headers: Type[chess.pgn.Headers]*)
    Collects headers into a dictionary.

class chess.pgn.**BoardBuilder**(**args, **kwds*)
    Returns the final position of the game. The mainline of the game is on the move stack.

class chess.pgn.**SkipVisitor**(**args, **kwds*)
    Skips a game.

class chess.pgn.**StringExporter**(**, columns: Optional[int] = 80, headers: bool = True, comments: bool = True, variations: bool = 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: TextIO*, ***, *columns: Optional[*int*] = 80*, *headers:* bool *=*
True, *comments:* bool *= True*, *variations:* bool *= 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: TextIO*) → Optional[chess.pgn.Headers]
Reads game headers from a PGN file opened in text mode. Skips the rest of the game.

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 and 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: TextIO*) → bool
> Skips a game. Returns `True` if a game was found and skipped.

8.3 Polyglot opening book reading

chess.polyglot.**open_reader**(*path: Union[str,   bytes,   os.PathLike]*)   →
> *chess.polyglot.MemoryMappedReader*
> 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**(*key: int*, *raw_move: int*, *weight: int*, *learn: int*, *move:* chess.Move)
> An entry from a Polyglot opening book.

> **key: int**
> > The Zobrist hash of the position.

> **raw_move: int**
> > The raw binary representation of the move. Use *move* instead.

weight: **int**
> An integer value that can be used as the weight for this entry.

learn: **int**
> Another integer value that can be used for extra information.

move: *chess.Move*
> The *Move*.

class chess.polyglot.**MemoryMappedReader**(*filename: Union[str, bytes, os.PathLike]*)
> Maps a Polyglot opening book to memory.

> **find_all**(*board: Union[chess.Board, int]*, *, *minimum_weight: int = 1, exclude_moves: Container[chess.Move] = []*) → Iterator[*chess.polyglot.Entry*]
> > Seeks a specific position and yields corresponding entries.

> **find**(*board: Union[chess.Board, int]*, *, *minimum_weight: int = 1, exclude_moves: Container[chess.Move] = []*) → *chess.polyglot.Entry*
> > 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: Union[chess.Board, int]*, *, *minimum_weight: int = 1, exclude_moves: Container[chess.Move] = [], random: Optional[random.Random] = None*) → *chess.polyglot.Entry*
> > Uniformly selects a random entry for the given position.

> > > **Raises** IndexError if no entries are found.

> **weighted_choice**(*board: Union[chess.Board, int]*, *, *exclude_moves: Container[chess.Move] = [], random: Optional[random.Random] = None*) → *chess.polyglot.Entry*
> > Selects a random entry for the given position, distributed by the weights of the entries.

> > > **Raises** IndexError if no entries are found.

> **close**() → None
> > 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: chess.Board, *, _hasher: Callable[[chess.Board], int] = <chess.polyglot.ZobristHasher object>*) → int
> 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.

> **Warning:** Ensure tablebase files match the known checksums. Maliciously crafted tablebase files may cause denial of service with `PythonTablebase` and memory unsafety with `NativeTablebase`.

chess.gaviota.**open_tablebase**(*directory: str*, ***, *libgtb: Optional[str] = None*, *LibraryLoader: ctypes.LibraryLoader[ctypes.CDLL] = <ctypes.LibraryLoader object>*) → Union[NativeTablebase, PythonTablebase]
　　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: str*) → None
　　　　Adds *.gtb.cp4* tables from a directory. The relevant files are lazily opened when the tablebase is actually probed.

　　probe_dtm(*board:* chess.Board) → int
　　　　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:* chess.Board) → int
　　　　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")
```

(continues on next page)

```
...          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**() → None
>   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: str*, ***, *libgtb: Optional[str] = None*,
                                      *LibraryLoader: ctypes.LibraryLoader[ctypes.CDLL] =
                                      <ctypes.LibraryLoader object>*) → NativeTablebase
>   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: ctypes.CDLL*)
>   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$_{50}$ (win/draw/loss under the 50-move rule) and DTZ$_{50}$" (distance to zeroing) information with rounding for all endgame positions with up to 7 pieces. Positions with castling rights are not included.

> **Warning:**   Ensure tablebase files match the known checksums. Maliciously crafted tablebase files may cause
> denial of service.

chess.syzygy.**open_tablebase**(*directory: str*, ***, *load_wdl: bool = True*, *load_dtz: bool = True*,
                              max_fds: Optional[int] = 128, *VariantBoard: Type[chess.Board] =
                              <class 'chess.Board'>*) → *chess.syzygy.Tablebase*
>   Opens a collection of tables for probing. See *Tablebase*.

> **Note:**   Generally probing requires tablebase files for the specific material composition, **as well as** material
> compositions transitively reachable by captures and promotions. This is important because 6-piece and 5-

piece (let alone 7-piece) files are often distributed separately, but are both required for 6-piece positions. Use `add_directory()` to load tables from additional directories.

---

class chess.syzygy.**Tablebase**(*, *max_fds: Optional[int] = 128*, *VariantBoard: Type[chess.Board]*
= *<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 necessary.

    add_directory(*directory: str*, *, *load_wdl: bool = True*, *load_dtz: bool = True*) → int
        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:* chess.Board) → int
        Probes WDL tables for win/draw/loss information under the 50-move rule, assuming the position has been reached directly after a capture or pawn move.

        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:* chess.Board) → int
        Probes DTZ tables for DTZ50" information with rounding.

        Minmaxing the DTZ50" 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.

        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 implies some primary tablebase lines may waste up to 1 ply. Rounding is never used for endgame phases where it would change the game theoretical outcome.

This means users need to be careful in positions that are nearly drawn under the 50-move rule! Carelessly wasting 1 more ply by not following the tablebase recommendation, for a total of 2 wasted plies, may change the outcome of the game.

```
>>> 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.

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

> **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() → None
Closes all loaded tables.

8.6 UCI/XBoard engine communication

The Universal chess interface (UCI) and XBoard protocol are standards for communicating with chess engines. This module implements an abstraction for playing moves and analysing positions with both kinds of engines.

> **Warning:** Many popular chess engines make no guarantees, not even memory safety, when parameters and positions are not completely *valid*. This module tries to deal with benign misbehaving engines, but ultimately they are executables running on your system.

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.

```python
import chess
import chess.engine

engine = chess.engine.SimpleEngine.popen_uci(r"C:\Users\xxxxx\Downloads\stockfish_14_
↪win_x64\stockfish_14_win_x64_avx2.exe")

board = chess.Board()
while not board.is_game_over():
    result = engine.play(board, chess.engine.Limit(time=0.1))
    board.push(result.move)

engine.quit()
```

```python
import asyncio
import chess
import chess.engine


async def main() -> None:
    transport, engine = await chess.engine.popen_uci(r"C:\Users\xxxxx\Downloads\
↪stockfish_14_win_x64\stockfish_14_win_x64_avx2.exe")

    board = chess.Board()
    while not board.is_game_over():
        result = await engine.play(board, chess.engine.Limit(time=0.1))
        board.push(result.move)

    await engine.quit()

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

class chess.engine.**Protocol**

Protocol for communicating with a chess engine process.

> **abstract async play**(*board: chess.Board*, *limit: chess.engine.Limit*, ***, *game: Optional[object] = None*, *info: chess.engine.Info = <Info.NONE: 0>*, *ponder: bool = False*, *draw_offered: bool = False*, *root_moves: Optional[Iterable[chess.Move]] = None*, *options: Mapping[str, Optional[Union[str, int, bool]]] = {}*) → *chess.engine.PlayResult*
>
> Plays 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.

- **draw_offered** – Whether the engine's opponent has offered a draw. Ignored by UCI engines.

- **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: Optional[float] = None, depth: Optional[int] = None, nodes: Optional[int] = None, mate: Optional[int] = None, white_clock: Optional[float] = None, black_clock: Optional[float] = None, white_inc: Optional[float] = None, black_inc: Optional[float] = None, remaining_moves: Optional[int] = None*)

Search-termination condition.

time: Optional[float] = None
    Search exactly *time* seconds.

depth: Optional[int] = None
    Search *depth* ply only.

nodes: Optional[int] = None
    Search only a limited number of *nodes*.

mate: Optional[int] = None
    Search for a mate in *mate* moves.

white_clock: Optional[float] = None
    Time in seconds remaining for White.

black_clock: Optional[float] = None
    Time in seconds remaining for Black.

white_inc: Optional[float] = None
    Fisher increment for White, in seconds.

black_inc: Optional[float] = None
    Fisher increment for Black, in seconds.

remaining_moves: Optional[int] = None
    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: Optional[chess.Move], ponder: Optional[chess.Move], info: Optional[chess.engine.InfoDict] = None, *, draw_offered: bool = False, resigned: bool = False*)

Returned by `chess.engine.Protocol.play()`.

move: Optional[*chess.Move*]
    The best move according to the engine, or None.

ponder: Optional[*chess.Move*]
    The response that the engine expects after *move*, or None.

info: *chess.engine.InfoDict*
    A dictionary of extra *information* sent by the engine, if selected with the *info* argument of `play()`.

draw_offered: **bool**
  Whether the engine offered a draw before moving.

resigned: **bool**
  Whether the engine resigned.

8.6.2 Analysing and evaluating a position

Example:

```python
import chess
import chess.engine

engine = chess.engine.SimpleEngine.popen_uci("/usr/bin/stockfish")

board = chess.Board()
info = engine.analyse(board, chess.engine.Limit(time=0.1))
print("Score:", info["score"])
# Score: PovScore(Cp(+20), WHITE)

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: PovScore(Mate(+1), WHITE)

engine.quit()
```

```python
import asyncio
import chess
import chess.engine

async def main() -> None:
    transport, engine = await chess.engine.popen_uci("/usr/bin/stockfish")

    board = chess.Board()
    info = await engine.analyse(board, chess.engine.Limit(time=0.1))
    print(info["score"])
    # Score: PovScore(Cp(+20), WHITE)

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

    await engine.quit()

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

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

  async analyse(*board:* chess.Board, *limit:* chess.engine.Limit, *, *game:* *object* = *'None'*, *info: chess.engine.Info = 'INFO_ALL'*, *root_moves: Optional[Iterable[*chess.Move*]]* *= 'None'*, *options: Mapping[*str*, Optional[Union[*str*, *int*, *bool*]]] = '{}'*) → chess.engine.InfoDict

async analyse(*board:* chess.Board, *limit:* chess.engine.Limit, *, *multipv:* *int*, *game:* *object* = *'None'*, *info: chess.engine.Info = 'INFO_ALL', root_moves: Optional[Iterable[*chess.Move*]] = 'None', options: Mapping[*str*, Optional[Union[*str*, *int*, *bool*]]] = '{}'*)* → List[*chess.engine.InfoDict*]

async analyse(*board:* chess.Board, *limit:* chess.engine.Limit, *, *multipv: Optional[int] = 'None'*, *game:* *object* = *'None'*, *info: chess.engine.Info = 'INFO_ALL', root_moves: Optional[Iterable[*chess.Move*]] = 'None', options: Mapping[*str*, Optional[Union[*str*, *int*, *bool*]]] = '{}'*)* → Union[List[*chess.engine.InfoDict*], *chess.engine.InfoDict*]

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.**InfoDict**(**args*, ***kwargs*)

Dictionary of aggregated 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`, `multipv` (1 for the mainline).

Others: `tbhits`, `currmove`, `currmovenumber`, `hashfull`, `cpuload`, `refutation`, `currline`, `ebf` (effective branching factor), `wdl` (a `PovWdl`), and `string`.

class chess.engine.**PovScore**(*relative:* chess.engine.Score, *turn: chess.Color*)

A relative `Score` and the point of view.

relative: *chess.engine.Score*

> The relative `Score`.

turn: chess.Color

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

white() → *chess.engine.Score*

> Gets the score from White's point of view.

black() → *chess.engine.Score*

> Gets the score from Black's point of view.

pov(*color: chess.Color*) → *chess.engine.Score*

> Gets the score from the point of view of the given *color*.

is_mate() → bool

> Tests if this is a mate score.

---

wdl(*, *model: Literal[sf, sf15, sf14, sf12, lichess] = 'sf'*, *ply: int = 30*) → *chess.engine.PovWdl*
    See *wdl()*.

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
```

    abstract score(*, *mate_score: int*) → int
    abstract score(*, *mate_score: Optional[int] = 'None'*) → Optional[int]
        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
```

    abstract mate() → Optional[int]
        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() → bool
        Tests if this is a mate score.

    abstract wdl(*, *model: Literal[sf, sf15, sf14, sf12, lichess] = 'sf'*, *ply: int = 30*) → *chess.engine.Wdl*
        Returns statistics for the expected outcome of this game, based on a *model*, given that this score is reached at *ply*.

        Scores have a total order, but it makes little sense to compute the difference between two scores. For example, going from Cp(-100) to Cp(+100) is much more significant than going from Cp(+300) to Cp(+500). It is better to compute differences of the expectation values for the outcome of the game (based on winning chances and drawing chances).

```
>>> Cp(100).wdl().expectation() - Cp(-100).wdl().expectation()
0.379...
```

```
>>> Cp(500).wdl().expectation() - Cp(300).wdl().expectation()
0.015...
```

Parameters

- **model** –

    - `sf`, the WDL model used by the latest Stockfish (currently `sf15`).

    - `sf15`, the WDL model used by Stockfish 15.

    - `sf14`, the WDL model used by Stockfish 14.

    - `sf12`, the WDL model used by Stockfish 12.

    - `lichess`, the win rate model used by Lichess. Does not use *ply*, and does not consider drawing chances.

- **ply** – The number of half-moves played since the starting position. Models may scale scores slightly differently based on this. Defaults to middle game.

class chess.engine.**PovWdl**(*relative:* chess.engine.Wdl, *turn: chess.Color*)

Relative *win/draw/loss statistics* and the point of view.

Deprecated since version 1.2: Behaves like a tuple (`wdl.relative.wins`, `wdl.relative.draws`, `wdl.relative.losses`) for backwards compatibility. But it is recommended to use the provided fields and methods instead.

relative: *chess.engine.Wdl*
    The relative *Wdl*.

turn: **chess.Color**
    The point of view (`chess.WHITE` or `chess.BLACK`).

white() → *chess.engine.Wdl*
    Gets the *Wdl* from White's point of view.

black() → *chess.engine.Wdl*
    Gets the *Wdl* from Black's point of view.

pov(*color: chess.Color*) → *chess.engine.Wdl*
    Gets the *Wdl* from the point of view of the given *color*.

class chess.engine.**Wdl**(*wins: int*, *draws: int*, *losses: int*)

Win/draw/loss statistics.

wins: **int**
    The number of wins.

draws: **int**
    The number of draws.

losses: **int**
    The number of losses.

total() → int
    Returns the total number of games. Usually, `wdl` reported by engines is scaled to 1000 games.

> **winning_chance**() → float
>> Returns the relative frequency of wins.

> **drawing_chance**() → float
>> Returns the relative frequency of draws.

> **losing_chance**() → float
>> Returns the relative frequency of losses.

> **expectation**() → float
>> Returns the expectation value, where a win is valued 1, a draw is valued 0.5, and a loss is valued 0.

8.6.3 Indefinite or infinite analysis

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

```python
import chess
import chess.engine

engine = chess.engine.SimpleEngine.popen_uci("/usr/bin/stockfish")

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

        # Arbitrary stop condition.
        if info.get("seldepth", 0) > 20:
            break

engine.quit()
```

```python
import asyncio
import chess
import chess.engine

async def main() -> None:
    transport, engine = await chess.engine.popen_uci("/usr/bin/stockfish")

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

            # Arbitrary stop condition.
            if info.get("seldepth", 0) > 20:
                break

    await engine.quit()

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

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

abstract async analysis(*board: chess.Board, limit: Optional[chess.engine.Limit] = None, *, multipv: Optional[int] = None, game: Optional[object] = None, info: chess.engine.Info = <Info.ALL: 31>, root_moves: Optional[Iterable[chess.Move]] = None, options: Mapping[str, Optional[Union[str, int, bool]]] = {}*) → *chess.engine.AnalysisResult*

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 analysis at any time.

class chess.engine.**AnalysisResult**(*stop: Optional[Callable[], None]] = None*)

Handle to ongoing engine analysis. Returned by `chess.engine.Protocol.analysis()`.

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

Automatically stops the analysis when used as a context manager.

multipv: List[`chess.engine.InfoDict`]

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

property info

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

stop() → None

Stops the analysis as soon as possible.

async wait() → *chess.engine.BestMove*

Waits until the analysis is finished.

async get() → *chess.engine.InfoDict*

Waits for the next dictionary of information from the engine and returns it.

It might be more convenient to use `async for info in analysis:  ....`

> **Raises** `chess.engine.AnalysisComplete` if the analysis is complete (or has been stopped) and all information has been consumed. Use `next()` if you prefer to get None instead of an exception.

would_block() → bool

Checks if calling `get()`, calling `next()`, or advancing the iterator one step would require waiting for the engine.

These functions would return immediately if information is pending (queue is not *empty*) or if the search is finished.

empty() → bool
    Checks if all current information has been consumed.

    If the queue is empty, but the analysis is still ongoing, then further information can become available in the future.

class chess.engine.**BestMove**(*move: Optional[chess.Move], ponder: Optional[chess.Move]*)
    Returned by *chess.engine.AnalysisResult.wait()*.

    move: Optional[*chess.Move*]
        The best move according to the engine, or None.

    ponder: Optional[*chess.Move*]
        The response that the engine expects after *move*, or None.

8.6.4 Options

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

```
>>> import chess.engine
>>>
>>> engine = chess.engine.SimpleEngine.popen_uci("/usr/bin/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})
>>>
>>> # [...]
```

```python
import asyncio
import chess.engine

async def main() -> None:
    transport, engine = await chess.engine.popen_uci("/usr/bin/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.**Protocol**
    Protocol for communicating with a chess engine process.

    options: MutableMapping[str, *Option*]
        Dictionary of available options.

abstract async configure(*options: Mapping[str, Optional[Union[str, int, bool]]]*) → None
> 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**(*name: str, type: str, default: Optional[Union[str, int, bool]], min: Optional[int], max: Optional[int], var: Optional[List[str]]*)
> Information about an available engine option.

> **name: str**
> > The name of the option.

> **type: str**
> > The type of the option.

type	UCI	CECP	value
check	X	X	`True` or `False`
spin	X	X	integer, between *min* and *max*
combo	X	X	string, one of *var*
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: Optional[Union[str, int, bool]]**
> > The default value of the option.

> **min: Optional[int]**
> > The minimum integer value of a *spin* option.

> **max: Optional[int]**
> > The maximum integer value of a *spin* option.

> **var: Optional[List[str]]**
> > A list of allowed string values for a *combo* option.

> **is_managed**() → bool
> > Some options are managed automatically: `UCI_Chess960`, `UCI_Variant`, `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.

```python
import logging

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

8.6.6 AsyncSSH

chess.engine.Protocol can also be used with AsyncSSH (since 1.16.0) to communicate with an engine on a remote computer.

```python
import asyncio
import asyncssh
import chess
import chess.engine


async def main() -> None:
    async with asyncssh.connect("localhost") as conn:
        channel, engine = await conn.create_subprocess(chess.engine.UciProtocol, "/
→usr/bin/stockfish")
        await engine.initialize()

        # Play, analyse, ...
        await engine.ping()

asyncio.run(main())
```

8.6.7 Reference

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

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

class chess.engine.**AnalysisComplete**
    Raised when analysis is complete, all information has been consumed, but further information was requested.

async chess.engine.**popen_uci**(*command:* *Union[str,* *List[str]],* **,* *setpgrp:* *bool* *=* *False,* ***popen_args:* *Any*) → Tuple[asyncio.transports.SubprocessTransport, *chess.engine.UciProtocol*]
    Spawns and initializes a 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.

async chess.engine.**popen_xboard**(*command:* *Union[str,* *List[str]],* **,* *setpgrp:* *bool* *=* *False,* ***popen_args:* *Any*) → Tuple[asyncio.transports.SubprocessTransport, *chess.engine.XBoardProtocol*]
    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.**Protocol**

Protocol for communicating with a chess engine process.

id: Dict[str, str]

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

returncode: asyncio.Future[int]

Future: Exit code of the process.

abstract async initialize() → None

Initializes the engine.

abstract async ping() → None

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

abstract async quit() → None

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: asyncio.transports.SubprocessTransport*, *protocol: chess.engine.Protocol*, *, *timeout: Optional[float] = 10.0*)

Synchronous wrapper around a transport and engine protocol pair. Provides the same methods and attributes as *chess.engine.Protocol* 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() → None

Closes the transport and the background event loop as soon as possible.

classmethod popen_uci(*command: Union[str, List[str]]*, *, *timeout: Optional[float] = 10.0*, *debug: bool = False*, *setpgrp: bool = False*, ***popen_args: Any*) → *chess.engine.SimpleEngine*

Spawns and initializes a UCI engine. Returns a *SimpleEngine* instance.

classmethod popen_xboard(*command: Union[str, List[str]]*, *, *timeout: Optional[float] = 10.0*, *debug: bool = False*, *setpgrp: bool = False*, ***popen_args: Any*) → *chess.engine.SimpleEngine*

Spawns and initializes an XBoard engine. Returns a *SimpleEngine* instance.

class chess.engine.**SimpleAnalysisResult**(*simple_engine: chess.engine.SimpleEngine*, *inner: chess.engine.AnalysisResult*)

Synchronous wrapper around *AnalysisResult*. Returned by chess.engine.SimpleEngine.analysis().

`chess.engine.`**`EventLoopPolicy`**`()` → None

> An event loop policy for thread-local event loops and child watchers. Ensures each event loop is capable of spawning and watching subprocesses, even when not running on the main thread.
>
> Windows: Uses `ProactorEventLoop`.
>
> Unix: Uses `SelectorEventLoop`. If available, `PidfdChildWatcher` is used to detect subprocess termination (Python 3.9+ on Linux 5.3+). Otherwise, the default child watcher is used on the main thread and relatively slow eager polling is used on all other threads.

8.7 SVG rendering

The `chess.svg` module renders SVG Tiny 1.2 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: chess.Piece, size: Optional[int] = None)` → str

> 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: Optional[chess.BaseBoard] = None, *, orientation: chess.Color = True, lastmove: Optional[chess.Move] = None, check: Optional[chess.Square] = None, arrows: Iterable[Union[chess.svg.Arrow, Tuple[chess.Square, chess.Square]]] = [], fill: Dict[chess.Square, str] = {}, squares: Optional[chess.IntoSquareSet] = None, size: Optional[int] = None, coordinates: bool = True, colors: Dict[str, str] = {}, flipped: bool = False, style: Optional[str] = None)` → str

> 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.
>
> - **orientation** – The point of view, defaulting to `chess.WHITE`.
>
> - **lastmove** – A `chess.Move` to be highlighted.
>
> - **check** – A square to be marked indicating a 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)]`.
>
> - **fill** – A dictionary mapping squares to a colors that they should be filled with.
>
> - **squares** – A `chess.SquareSet` with selected squares to mark with an X.
>
> - **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.
>
> - **coordinates** – Pass `False` to disable the coordinate margin.
>
> - **colors** – A dictionary to override default colors. Possible keys are `square light`, `square dark`, `square light lastmove`, `square dark lastmove`, `margin`, `coord`, `arrow green`, `arrow blue`, `arrow red`, and `arrow yellow`. Values should look like `#ffce9e` (opaque), or `#15781B80` (transparent).

- **flipped** – Pass `True` to flip the board.

- **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")
>>>
>>> chess.svg.board(
...     board,
...     fill=dict.fromkeys(board.attacks(chess.E4), "#cc0000cc") | {chess.E4: "
↪#00cc00cc"},
...     arrows=[chess.svg.Arrow(chess.E4, chess.F6, color="#0000cccc")],
...     squares=chess.SquareSet(chess.BB_DARK_SQUARES & chess.BB_FILE_B),
...     size=350,
... )
```

Deprecated since version 1.1: Use *orientation* with a color instead of the *flipped* toggle.

class chess.svg.**Arrow**(*tail: chess.Square*, *head: chess.Square*, *, *color: str = 'green'*)
Details of an arrow to be drawn.

tail: chess.Square
Start square of the arrow.

head: chess.Square
End square of the arrow.

color: str
Arrow color.

pgn() → str
Returns the arrow in the format used by `[%csl ...]` and `[%cal ...]` PGN annotations, e.g., `Ga1` or `Ya2h2`.

Colors other than `red`, `yellow`, and `blue` default to green.

classmethod from_pgn(*pgn: str*) → *chess.svg.Arrow*
Parses an arrow from the format used by `[%csl ...]` and `[%cal ...]` PGN annotations, e.g., `Ga1` or `Ya2h2`.

Also allows skipping the color prefix, defaulting to green.

> **Raises** `ValueError` if the format is invalid.

8.8 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).xboard_variant
'giveaway'
>>> type(board).starting_fen
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w - - 0 1'
```

Variant	Board class	UCI/XBoard	Syzygy
Standard	*chess.Board*	chess/normal	.rtbw, .rtbz
Suicide	chess.variant.SuicideBoard	suicide	.stbw, .stbz
Giveaway	chess.variant.GiveawayBoard	giveaway	.gtbw, .gtbz
Antichess	chess.variant.AntichessBoard	antichess	.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: str*) → Type[*chess.Board*]
> Looks for a variant board class by variant name. Supports many common aliases.

8.8.1 Game end

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

Note that if all of them return False, the game may still be over and decided by standard conditions like *is_checkmate()*, *is_stalemate()*, *is_insufficient_material()*, move counters, repetitions, and legitimate claims.

8.8.2 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.

8.8.3 Crazyhouse

class chess.variant.**CrazyhousePocket**(*symbols: Iterable[str] = ''*)
> A Crazyhouse pocket with a counter for each piece type.

> **reset**() → None
>> Clears the pocket.

> **add**(*piece_type: int*) → None
>> Adds a piece of the given type to this pocket.

> **remove**(*piece_type: int*) → None
>> Removes a piece of the given type from this pocket.

    count(*piece_type:* *int*) → int

        Returns the number of pieces of the given type in the pocket.

    copy() → CrazyhousePocketT

        Returns a copy of this pocket.

class chess.variant.**CrazyhouseBoard**(*fen: Optional[str]   =   'rn-bqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR[] w KQkq - 0 1', chess960: bool = False*)

    pockets = [chess.variant.CrazyhousePocket(), chess.variant.CrazyhousePocket()]

        Pockets for each color. For example, `board.pockets[chess.WHITE]` are the pocket pieces available to White.

    legal_drop_squares() → *chess.SquareSet*

        Gets the squares where the side to move could legally drop a piece. Does *not* check whether they actually have a suitable piece in their pocket.

        It is legal to drop a checkmate.

        Returns a *set of squares*.

8.8.4 Three-check

class chess.variant.**ThreeCheckBoard**(*fen: Optional[str]   =   'rn-bqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 3+3 0 1', chess960: bool = False*)

    remaining_checks = [3, 3]

        Remaining checks until victory for each color. For example, `board.remaining_checks[chess.WHITE] == 0` implies that White has won.

8.8.5 UCI/XBoard

Multi-Variant Stockfish and other engines have an `UCI_Variant` option. XBoard engines may declare support for `variants`. This is automatically managed.

```
>>> import chess.engine
>>>
>>> engine = chess.engine.SimpleEngine.popen_uci("stockfish-mv")
>>>
>>> board = chess.variant.RacingKingsBoard()
>>> result = engine.play(board, chess.engine.Limit(time=1.0))
```

8.8.6 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)
```

8.9 Changelog for python-chess

8.9.1 New in v1.9.3

Bugfixes:

- Fix some valid characters were not accepted in PGN tag names.

Changes:

- Skip over syntactically invalid PGN tags.
- Detect Antichess insufficient material with two opposing knights.

New features:

- Add `chess.Board.unicode(..., orientation=chess.WHITE)`.

8.9.2 New in v1.9.2

Bugfixes:

- Fix recursive Crazyhouse move generation sometimes failing with with `RuntimeError`.
- Fix rendering of black pawn SVG on dark background.

New features:

- Add `chess.engine.AnalysisResult.would_block()`.

8.9.3 New in v1.9.1

Bugfixes:

- Reject pawn capture SAN if the original file is not specified, e.g., `d5` will no longer match `cxd5`.

Changes:

- Tweak handling of whitespace in PGN comments: When parsing, any leading and trailing whitespace (beyond one space) is preserved. When joining multiple PGN comments, they are now separated with a space instead of a newline character. When removing annotations from comments, leftover whitespace is avoided.

New features:

- Add model `sf15` for `chess.engine.Score.wdl()`.

8.9.4 New in v1.9.0

Bugfixes:

- Expand position validation to detect check conflicting with en passant square.

New features:

- Add `chess.svg.board(..., fill=...)`.
- Let `chess.svg.board()` add ASCII board as description of SVG.
- Add hint when engine process dies due to illegal instruction.

8.9.5 New in v1.8.0

Bugfixes:

- Fix `SquareSet.issuperset()` and `SquareSet.issubset()` by swapping their respective implementations.

New features:

- Read and write PGN comments like `[%emt 0:05:21]`.

8.9.6 New in v1.7.0

New features:

- Add new models for `chess.engine.Score.wdl()`: `sf` (the new default) and `sf14`.
- Add `chess.Board.piece_map()`.

Bugfixes:

- `chess.pgn`: Fix skipping with nested variations.
- `chess.svg`: Make check gradient compatible with QtSvg.

8.9.7 New in v1.6.1

Bugfixes:

- Make `chess.engine.SimpleEngine.play(..., draw_offered=True)` available. Previously only added for `chess.engine.Protocol`.

8.9.8 New in v1.6.0

New features:

- Allow offering a draw to XBoard engines using `chess.engine.Protocol.play(..., draw_offered=True)`.
- Now detects insufficient material in Horde. Thanks @stevepapazis!

Changes:

- `chess.engine.popen_engine(..., setpgrp=True)` on Windows now merges `CREATE_NEW_PROCESS_GROUP` into `creationflags` instead of overriding. On Unix it now uses `start_new_session` instead of calling `setpgrp` in `preexec_fn`.
- Declare that `chess.svg` produces SVG Tiny 1.2, and prepare SVG 2 forwards compatibility.

Bugfixes:

- Fix slightly off-center pawns in `chess.svg`.
- Fix typing error in Python 3.10 (due to added `int.bit_count`).

8.9.9 New in v1.5.0

Bugfixes:

- Fixed typing of `chess.pgn.Mainline.__reversed__()`. It is now a generator, and `chess.pgn.ReverseMainline` has been **removed**. This is a breaking change but a required bugfix.

- Implement UCI **ponderhit** for consecutive calls to `chess.engine.Protocol.play(..., ponder=True)`. Previously, the pondering search was always stopped and restarted.

- Provide the full move stack, not just the position, for UCI pondering.

- Fixed XBoard level in sudden death games.

- Ignore trailing space after ponder move sent by UCI engine. Previously, such a move would be rejected.

- Prevent cancelling engine commands after they have already been cancelled or completed. Some internals (`chess.engine.BaseCommand`) have been changed to accomplish this.

New features:

- Added `chess.Board.outcome()`.

- Implement and accept usermove feature for XBoard engines.

Special thanks to @MarkZH for many of the engine related changes in this release!

8.9.10 New in v1.4.0

New features:

- Let `chess.pgn.GameNode.eval()` accept PGN comments like `[%eval 2.5,11]`, meaning 250 centipawns at depth 11. Use `chess.pgn.GameNode.eval_depth()` and `chess.pgn.GameNode.set_eval(..., depth)` to get and set the depth.

- Read and write PGN comments with millisecond precision like `[%clk 1:23:45.678]`.

Changes:

- Recover from invalid UTF-8 sent by an UCI engine, by ignoring that (and only that) line.

8.9.11 New in v1.3.3

Bugfixes:

- Fixed unintended collisions and optimized `chess.Piece.__hash__()`.

- Fixed false-positive `chess.STATUS_IMPOSSIBLE_CHECK` if checkers are aligned with other king.

Changes:

- Also detect `chess.STATUS_IMPOSSIBLE_CHECK` if checker is aligned with en passant square and king.

New features:

- Implemented Lichess winning chance model for `chess.engine.Score`: `score.wdl(model="lichess")`.

8.9.12 New in v1.3.2

Bugfixes:

- Added a new reason for `board.status()` to be invalid: `chess.STATUS_IMPOSSIBLE_CHECK`. This detects positions where two sliding pieces are giving check while also being aligned with the king on the same rank, file, or diagonal. Such positions are impossible to reach, break Stockfish, and maybe other engines.

8.9.13 New in v1.3.1

Bugfixes:

- `chess.pgn.read_game()` now properly detects variant games with Chess960 castling rights (as well as mislabeled Standard Chess960 games). Previously, all castling moves in such games were rejected.

8.9.14 New in v1.3.0

Changes:

- Introduced `chess.pgn.ChildNode`, a subclass of `chess.pgn.GameNode` for all nodes other than the root node, and converted `chess.pgn.GameNode` to an abstract base class. This improves ergonomics in typed code.

  The change is backwards compatible if using only documented features. However, a notable undocumented feature is the ability to create dangling nodes. This is no longer possible. If you have been using this for subclassing, override `GameNode.add_variation()` instead of `GameNode.dangling_node()`. It is now the only method that creates child nodes.

Bugfixes:

- Removed broken `weakref`-based caching in `chess.pgn.GameNode.board()`.

New features:

- Added `chess.pgn.GameNode.next()`.

8.9.15 New in v1.2.2

Bugfixes:

- Fixed regression where releases were uploaded without the `py.typed` marker.

8.9.16 New in v1.2.1

Changes:

- The primary location for the published package is now https://pypi.org/project/chess/. Thanks to Kristian Glass for transferring the namespace.

  The old https://pypi.org/project/python-chess/ will remain an alias that installs the package from the new location as a dependency (as recommended by PEP423).

  `ModuleNotFoundError: No module named 'chess'` after upgrading from previous versions? Run `pip install --force-reinstall chess` (due to https://github.com/niklasf/python-chess/issues/680).

8.9.17 New in v1.2.0

New features:

- Added `chess.Board.ply()`.

- Added `chess.pgn.GameNode.ply()` and `chess.pgn.GameNode.turn()`.

- Added `chess.engine.PovWdl`, `chess.engine.Wdl`, and conversions from scores: `chess.engine.PovScore.wdl()`, `chess.engine.Score.wdl()`.

- Added `chess.engine.Score.score(*, mate_score: int) -> int` overload.

Changes:

- The `PovScore` returned by `chess.pgn.GameNode.eval()` is now always relative to the side to move. The ambiguity around `[%eval #0]` has been resolved to `Mate(-0)`. This makes sense, given that the authors of the specification probably had standard chess in mind (where a game-ending move is always a loss for the opponent). Previously, this would be parsed as `None`.

- Typed `chess.engine.InfoDict["wdl"]` as the new `chess.engine.PovWdl`, rather than `Tuple[int, int, int]`. The new type is backwards compatible, but it is recommended to use its documented fields and methods instead.

- Removed `chess.engine.PovScore.__str__()`. String representation falls back to `__repr__`.

- The `en_passant` parameter of `chess.Board.fen()` and `chess.Board.epd()` is now typed as `Literal["legal", "fen", "xfen"]` rather than `str`.

8.9.18 New in v1.1.0

New features:

- Added `chess.svg.board(..., orientation)`. This is a more idiomatic way to set the board orientation than `flipped`.

- Added `chess.svg.Arrow.pgn()` and `chess.svg.Arrow.from_pgn()`.

Changes:

- Further relaxed `chess.Board.parse_san()`. Now accepts fully specified moves like `e2e4`, even if that is not a pawn move, castling notation with zeros, null moves in UCI notation, and null moves in XBoard notation.

8.9.19 New in v1.0.1

Bugfixes:

- `chess.svg`: Restored SVG Tiny compatibility by splitting colors like `#rrggbbaa` into a solid color and opacity.

8.9.20 New in v1.0.0

See `CHANGELOG-OLD.rst` for changes up to v1.0.0.

INDICES AND TABLES

- genindex
- search