reconchess Documentation

Release 1.6.9

Corey Lowman, Casey Richardson

Feb 17, 2022

Contents

CHAPTER 1

Introduction

Reconnaissance Chess is a chess variant (more precisely, a family of chess variants) invented as an R&D project at Johns Hopkins Applied Physics Laboratory (JHU/APL). Reconnaissance Chess adds the following elements to standard (classical) chess: sensing; incomplete information; decision making under uncertainty; coupled management of 'battle forces' and 'sensor resources'; and adjudication of multiple, simultaneous, and competing objectives. Reconnaissance chess is a paradigm and test bed for understanding and experimenting with autonomous decision making under uncertainty and in particular managing a network of sensors to maintain situational awareness informing tactical and strategic decision making.

The game implemented in this python package is a relatively basic version using only one kind of sensor that provides perfect information in a small region of the chess board. In the future, extended versions may include noisy sensors of different types; multiple sensing actions per turn; the need to divide attention and resources among multiple, concurrent games; and other complicating factors.

This package includes a "game arbiter" which controls the game flow, maintains the ground truth game board, and notifies players of information collected by sense and move actions. The package also contains a client API for interacting with the arbiter, which can be used by bot players or other game interfaces.

CHAPTER 2

Installation

```
pip install reconchess
```

CHAPTER 3

Documentation

Documentation is hosted by Read the Docs.

CHAPTER 4

License

Distributed under BSD 3-Clause License, for details see LICENSE file.

CHAPTER 5

Contents

5.1 Installation and Quick Start

5.1.1 Install

Install the python package

```
$ pip install reconchess
```

5.1.2 Quick Start

You can run a test game between two of the baseline bots:

```
$ rc-bot-match reconchess.bots.random_bot reconchess.bots.random_bot
```

Then replay the game:

```
$ rc-replay <game-output-json-file>
```

Or, play against a bot yourself:

```
$ rc-play reconchess.bots.random_bot
```

Now you are ready to make your own bot algorithm to play Reconnaissance Chess.

5.1.3 Server Quick Start

Register your account by visiting https://rbc.jhuapl.edu/register.

Then you can play games on the server with `rc-connect` and providing your bot username and password when prompted:

```
$ rc-connect src/my_awesome_bot.py
Username: MyAwesomeUsername
Password: ...
[<time>] Connected successfully to server!
```

Playing ranked matches on the server is as easy as specifying the `--ranked` flag:

```
$ rc-connect --ranked src/my_awesome_bot.py
Username: MyAwesomeUsername
Password: ...
Are you sure you want to participate in ranked matches as v1 (currently v0)? [y/n]y
[<time stamp>] Connected successfully to server!
```

5.2 Introduction to Reconnaissance Chess

The overall idea of Reconnaissance Chess is that you play chess without sight of the opponent's pieces, and you gain information about the opponent's position by sensing. On your turn, you first pick a part of the board to sense. Your sensor has a 3 square x 3 square field of view, and once you pick where to sense you are shown what the ground truth board position looks like in that 3x3 window. Then you select a move, similar to classical chess.

Most of the rules follow classical chess, including piece movements, initial board configuration, en passant capture, pawn promotion, castling, etc. However, there are some modifications, many of which are required because of the uncertainty in the game. One major change from classical chess is that in Recon Chess the goal is to simply capture the opponent king, there are no notions of checkmate or check. Also, because the true board position is uncertain, you may try to make a move that is actually illegal; this results in either a loss of turn or a modification to your requested move. See the full rules below.

In addition to sensing, there are other sources of information about the true position. You are always notified about where your pieces move (or don't) and when one of your pieces was captured by the opponent (but you are not told the piece that did the capturing); this information allows you to keep track of the true position of your own pieces. You are notified if you make a capture (but not which piece is captured). And there is also information to be gained about the opponent from moves you attempted but where not legal (e.g., attempted pawn captures). It is up to the player to fuse this information during the game to form their best representation of the true board state (i.e., a "world model"). It is the job of the "game arbiter" (implemented as part of this python package) to maintain the ground truth board position, control the game flow, and notify each player about any information they have gained through sensing, moving, or captures.

5.3 Rules of Reconnaissance Chess

Reconnaissance Chess can be thought of as a family of chess variants that incorporate sensing and incomplete information. This python package implements a version of the game with the following rules.

1. The Rules of Standard Chess apply, with the exceptions and modifications that follow.
2. The objective of the game is to capture the king, not deliver checkmate. When one player captures the other player's king, the game is ended, and the capturing player wins.
3. The rules associated with check are all eliminated. This includes: a player is not told if their king is in check, a player may make a move that leaves their king in check, and can castle into or through check.
4. All rules associated with stalemates are eliminated. Automatic draws can be included, but are declared by the arbiter rather than requested by a player. This library defaults to a non-optional version of the "50-move rule."
5. Each player's turn has three phases, played in this order: turn start phase, sense phase, and move phase.

a. Turn Start phase: the player's turn begins, and if the opponent captured a piece on their turn, the current player is given the capture square (thus the current player also knows which piece was captured).

b. Sense phase: the player chooses any square on the chessboard to target their sensor. Then, the true state of the game board in a three square by three square window centered at the chosen square is revealed to the sensing player. This includes showing all pieces and empty squares in the 3x3 window.

c. Move phase: the player chooses any chess move, or chooses to "pass." If the move is a pawn promotion and the player does not specify a piece to promote to, then a queen promotion is assumed. Then, given that move, one of three conditions holds:

   i. The move is legal on the game board.

   ii. The moving piece is a queen, bishop, or rook and the move is illegal on the game board because one or more opponent pieces block the path of the moving piece. Then, the move is modified so that the destination square is the location of the first obstructing opponent piece, and that opponent piece is captured. (Note: this rule does not apply to a castling king).

   iii. The moving piece is a pawn, moving two squares forward on that pawn's first move, and the move is illegal because an opponent's piece blocks the path of the pawn. Then, the move is modified so that the pawn moves only one square if that modified move is legal, otherwise the player's move is illegal.

   iv. If any of (i)-(iii) do not hold, the move is considered illegal (or the player chose to pass which has the same result).

   The results of the move are then determined: if condition (iv) holds, then no changes are made to the board, the player is notified that their move choice was illegal (or the pass is acknowledged), and the player's turn is over. Otherwise the move is made on the game board. If the move was modified because of (ii) or (iii), then the modified move is made, and the current player is notified of the modified move in the move results. If the move results in a capture, the current player is notified that they captured a piece and which square the capture occurred, but not the type of opponent piece captured (the opponent will be notified of the capture square on their turn start phase). If the move captures the opponent's king, the game ends and both players are notified. The current player's turn is now over and play proceeds to the opponent.

6. The only information revealed to either player about the game or opponent actions is that explicitly stated in (5).

7. The game can also be played with a chess clock, in which case the player is notified of their remaining clock time, but not their opponent's remaining clock time. Both players are notified if either player loses on time.

5.4 Creating a bot

To create a reconchess bot, extend the `reconchess.Player` base class and implement the abstract methods that it has. In order to use the reconchess scripts, the main python file you pass into the scripts must contain exactly 1 sub class of `reconchess.Player`.

For more information on the API see the `reconchess.Player` section on the *reconchess API* page.

5.4.1 Example bot: Random bot

The random bot takes random actions each turn, for both sensing and moving. It only really implements the `reconchess.Player.choose_sense()` and `reconchess.Player.choose_move()` methods.

```
import random
from reconchess import *


```

(continues on next page)

(continued from previous page)

```
class RandomBot(Player):
    def handle_game_start(self, color: Color, board: chess.Board, opponent_name: str):
        pass

    def handle_opponent_move_result(self, captured_my_piece: bool, capture_square:␣
↪Optional[Square]):
        pass

    def choose_sense(self, sense_actions: List[Square], move_actions: List[chess.
↪Move], seconds_left: float) -> \
            Optional[Square]:
        return random.choice(sense_actions)

    def handle_sense_result(self, sense_result: List[Tuple[Square, Optional[chess.
↪Piece]]]):
        pass

    def choose_move(self, move_actions: List[chess.Move], seconds_left: float) ->␣
↪Optional[chess.Move]:
        return random.choice(move_actions + [None])

    def handle_move_result(self, requested_move: Optional[chess.Move], taken_move:␣
↪Optional[chess.Move],
                           captured_opponent_piece: bool, capture_square:␣
↪Optional[Square]):
        pass

    def handle_game_end(self, winner_color: Optional[Color], win_reason:␣
↪Optional[WinReason],
                        game_history: GameHistory):
        pass
```

5.4.2 Example bot: Trout bot

The trout bot is a baseline that you can test your bot against. It keeps track of a single `chess.Board` and uses the Stockfish engine to make a move. When it gets information back from the game, it naively applies that information to its `chess.Board`.

NOTE You will need to download Stockfish and create an environment variable called *STOCKFISH_EXECUTABLE* that has the path to the Stockfish executable to use TroutBot.

```
import chess.engine
import random
from reconchess import *
import os

STOCKFISH_ENV_VAR = 'STOCKFISH_EXECUTABLE'


class TroutBot(Player):
    """
    TroutBot uses the Stockfish chess engine to choose moves. In order to run␣
↪TroutBot you'll need to download
    Stockfish from https://stockfishchess.org/download/ and create an environment␣
↪variable called STOCKFISH_EXECUTABLE
```

(continues on next page)

(continued from previous page)

```
    that is the path to the downloaded Stockfish executable.
    """

    def __init__(self):
        self.board = None
        self.color = None
        self.my_piece_captured_square = None

        # make sure stockfish environment variable exists
        if STOCKFISH_ENV_VAR not in os.environ:
            raise KeyError(
                'TroutBot requires an environment variable called "{}" pointing to␣
↪the Stockfish executable'.format(
                    STOCKFISH_ENV_VAR))

        # make sure there is actually a file
        stockfish_path = os.environ[STOCKFISH_ENV_VAR]
        if not os.path.exists(stockfish_path):
            raise ValueError('No stockfish executable found at "{}"'.format(stockfish_
↪path))

        # initialize the stockfish engine
        self.engine = chess.engine.SimpleEngine.popen_uci(stockfish_path,␣
↪setpgrp=True)

    def handle_game_start(self, color: Color, board: chess.Board, opponent_name: str):
        self.board = board
        self.color = color

    def handle_opponent_move_result(self, captured_my_piece: bool, capture_square:␣
↪Optional[Square]):
        # if the opponent captured our piece, remove it from our board.
        self.my_piece_captured_square = capture_square
        if captured_my_piece:
            self.board.remove_piece_at(capture_square)

    def choose_sense(self, sense_actions: List[Square], move_actions: List[chess.
↪Move], seconds_left: float) -> \
            Optional[Square]:
        # if our piece was just captured, sense where it was captured
        if self.my_piece_captured_square:
            return self.my_piece_captured_square

        # if we might capture a piece when we move, sense where the capture will occur
        future_move = self.choose_move(move_actions, seconds_left)
        if future_move is not None and self.board.piece_at(future_move.to_square) is␣
↪not None:
            return future_move.to_square

        # otherwise, just randomly choose a sense action, but don't sense on a square␣
↪where our pieces are located
        for square, piece in self.board.piece_map().items():
            if piece.color == self.color:
                sense_actions.remove(square)
        return random.choice(sense_actions)

    def handle_sense_result(self, sense_result: List[Tuple[Square, Optional[chess.
↪Piece]]]):
```

(continues on next page)

(continued from previous page)

```
        # add the pieces in the sense result to our board
        for square, piece in sense_result:
            self.board.set_piece_at(square, piece)

    def choose_move(self, move_actions: List[chess.Move], seconds_left: float) ->␣
↪Optional[chess.Move]:
        # if we might be able to take the king, try to
        enemy_king_square = self.board.king(not self.color)
        if enemy_king_square:
            # if there are any ally pieces that can take king, execute one of those␣
↪moves
            enemy_king_attackers = self.board.attackers(self.color, enemy_king_square)
            if enemy_king_attackers:
                attacker_square = enemy_king_attackers.pop()
                return chess.Move(attacker_square, enemy_king_square)

        # otherwise, try to move with the stockfish chess engine
        try:
            self.board.turn = self.color
            self.board.clear_stack()
            result = self.engine.play(self.board, chess.engine.Limit(time=0.5))
            return result.move
        except chess.engine.EngineTerminatedError:
            print('Stockfish Engine died')
        except chess.engine.EngineError:
            print('Stockfish Engine bad state at "{}"'.format(self.board.fen()))

        # if all else fails, pass
        return None

    def handle_move_result(self, requested_move: Optional[chess.Move], taken_move:␣
↪Optional[chess.Move],
                           captured_opponent_piece: bool, capture_square:␣
↪Optional[Square]):
        # if a move was executed, apply it to our board
        if taken_move is not None:
            self.board.push(taken_move)

    def handle_game_end(self, winner_color: Optional[Color], win_reason:␣
↪Optional[WinReason],
                        game_history: GameHistory):
        try:
            # if the engine is already terminated then this call will throw an␣
↪exception
            self.engine.quit()
        except chess.engine.EngineTerminatedError:
            pass
```

5.5 Playing games

5.5.1 Bot vs Bot

Command Line

Playing two bots against each other is as easy as playing against your bot, using the built in script `rc-bot-match`. Similar to `rc-play`, `rc-bot-match` gets added to your path so you can execute it from the command line. It takes two bots and plays them against each other using a `LocalGame`:

```
rc-bot-match --help
rc-bot-match <white bot> <black bot>
rc-bot-match reconchess.bots.random_bot src/my_awesome_bot.py
rc-bot-match reconchess.bots.random_bot reconchess.bots.random_bot
rc-bot-match src/my_okay_bot.py src/my_awesome_bot.py
```

Use the `--help` flag for more information about the arguments.

PyCharm

If you use PyCharm for development, you can create a run configuration to run your bot from PyCharm by creating a new run configuration that targets a module instead of a script. Target the `reconchess.scripts.rc-bot-match` module:

5.5.2 Playing against your bot

Playing against your bot is very easy with the built in script `rc-play`. When you install the reconchess package, the rc-play script gets added to your path so you can run it like an executable. `rc-play` expects an argument that will point it to the bot to play against. It uses *`reconchess.load_player()`* to load the bot, so it can accept either a path to a python source file, or a python module name. To play against one of your own bots you will use a path to the source file.

In either case, rc-play will create an instance of the provided bot, and a *`reconchess.LocalGame`*. It will then open up a window using PyGame that you can play against the bot with. rc-play handles running the *`reconchess.LocalGame`*, and interfacing your actions with the game.

```
rc-play --help
rc-play <path to bot source file or bot module>
rc-play reconchess.bots.random_bot
rc-play src/my_awesome_bot.py
```

Use the `--help` flag for more information about the arguments.

Sensing

When it is your turn to sense, the squares under your mouse will be highlighted, indicating which squares will be sensed if you choose the square underneath your mouse.

Click the square to select it as the sensing action.

Moving

When it is your turn to move (after sensing), hovering over a piece will indicate the move actions you can take. Note that just because a move is shown doesn't mean it is valid.

Click and drag a piece to one of the highlighted squares to make a move action.

Changing pieces on the board

If you ever want to adjust the pieces on your board, for example if you know a piece isn't there, you can **right click a square to cycle through the possible pieces**.

Captures

If you capture a piece, or one of your pieces is captured, the square will be highlighted in red for the remainder of the turn.

5.6 Replaying games

You can replay a game that was already played using the built in script `rc-replay`. When you install the reconchess package, the rc-play script gets added to your path so you can run it like an executable. `rc-replay` expects an argument that points to a saved game history file, which will be a JSON file. `rc-play` and `rc-bot-match` automatically save a game history file in the directory where they are run, so you can use those files as input to `rc-replay`. See *`reconchess.Player.handle_game_end()`* for using the game history object and *`reconchess.GameHistory.save()`* for saving it.

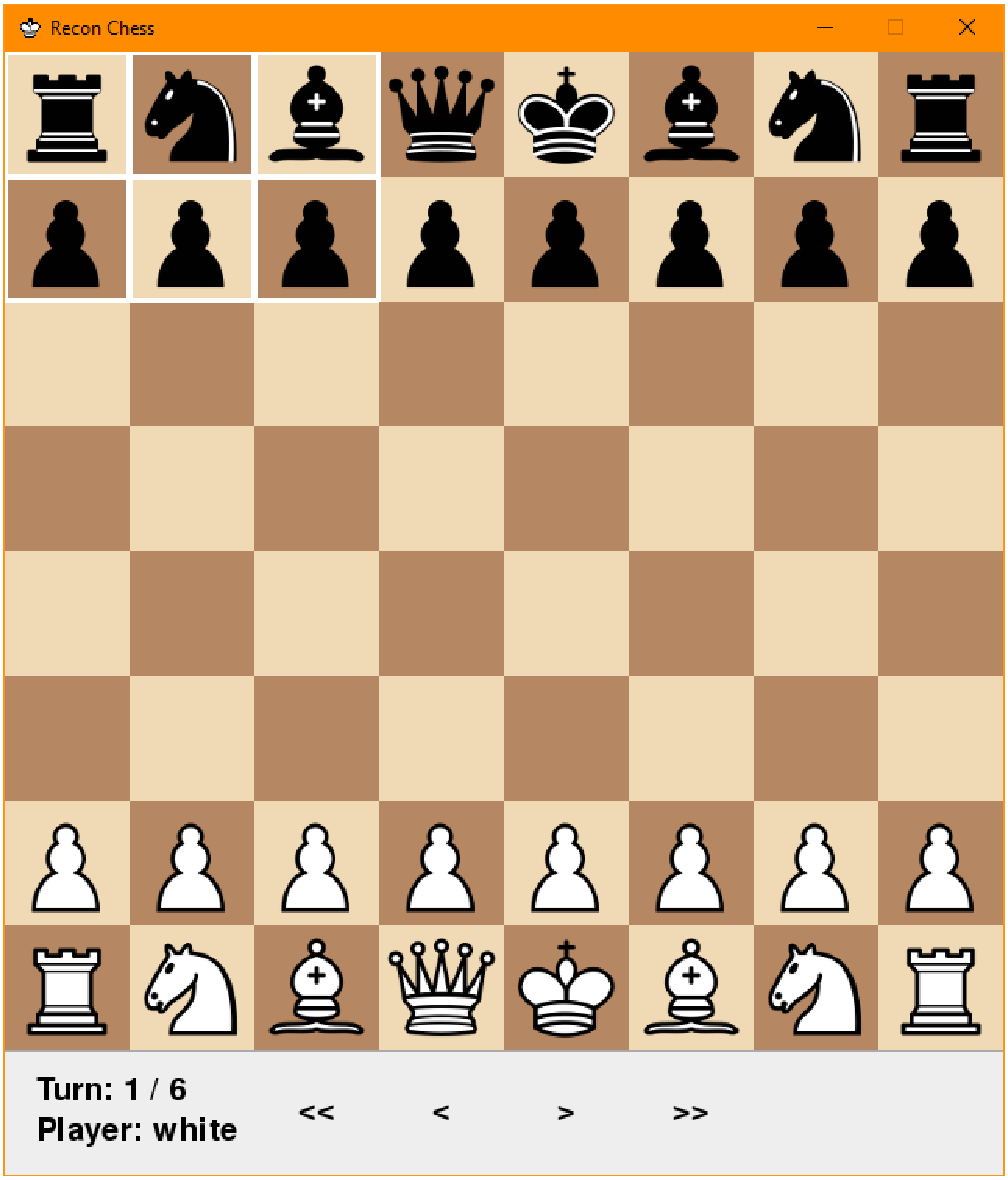
```
rc-replay --help
rc-replay <path to saved game history file>
rc-replay crazy_game.json
```

Use the `--help` flag for more information about the arguments.

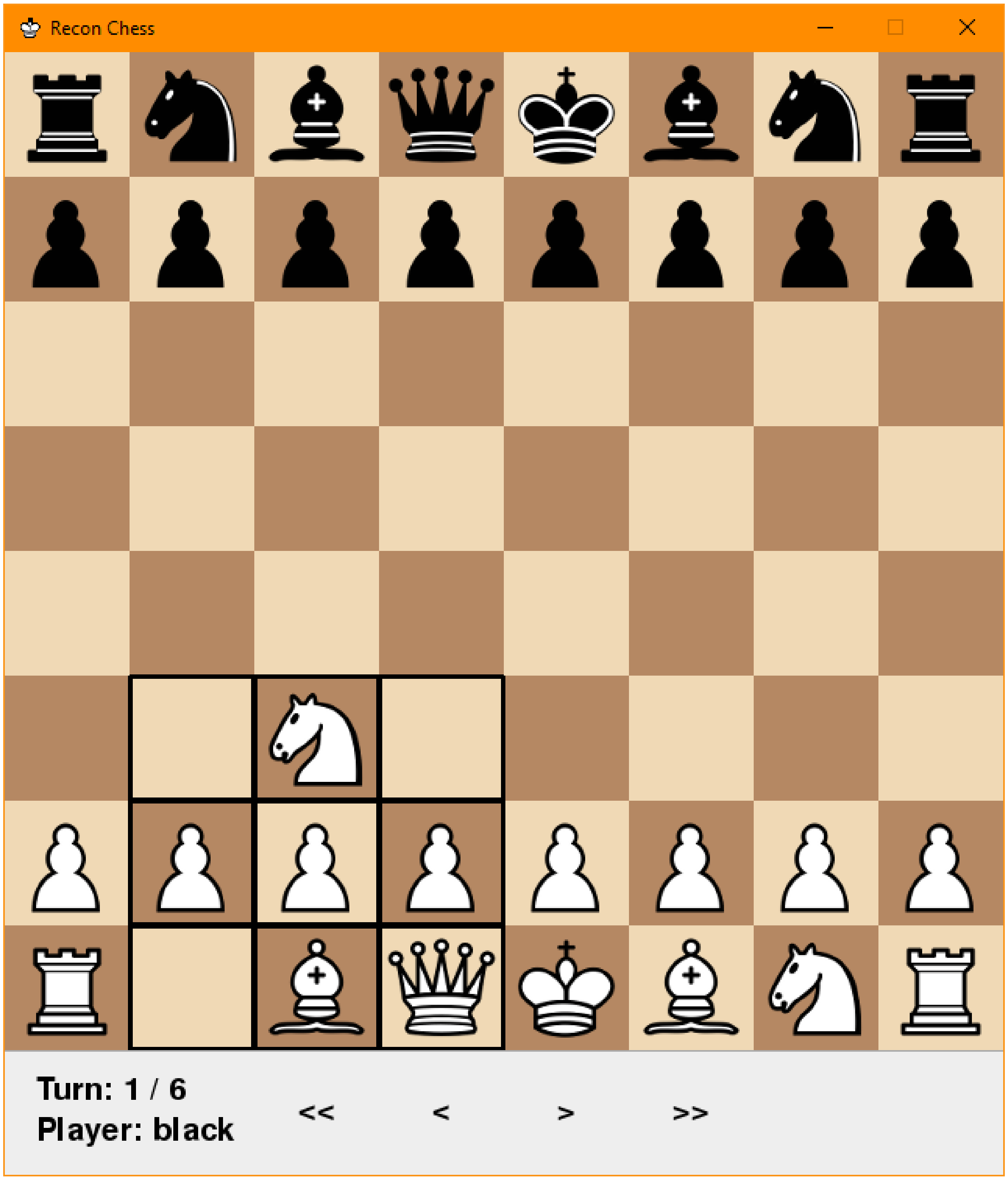
5.6.1 Sense Action

Senses are shown by highlighting the squares that the sense covered in the color of the player.

Here the white player senses over black's knight at B8:



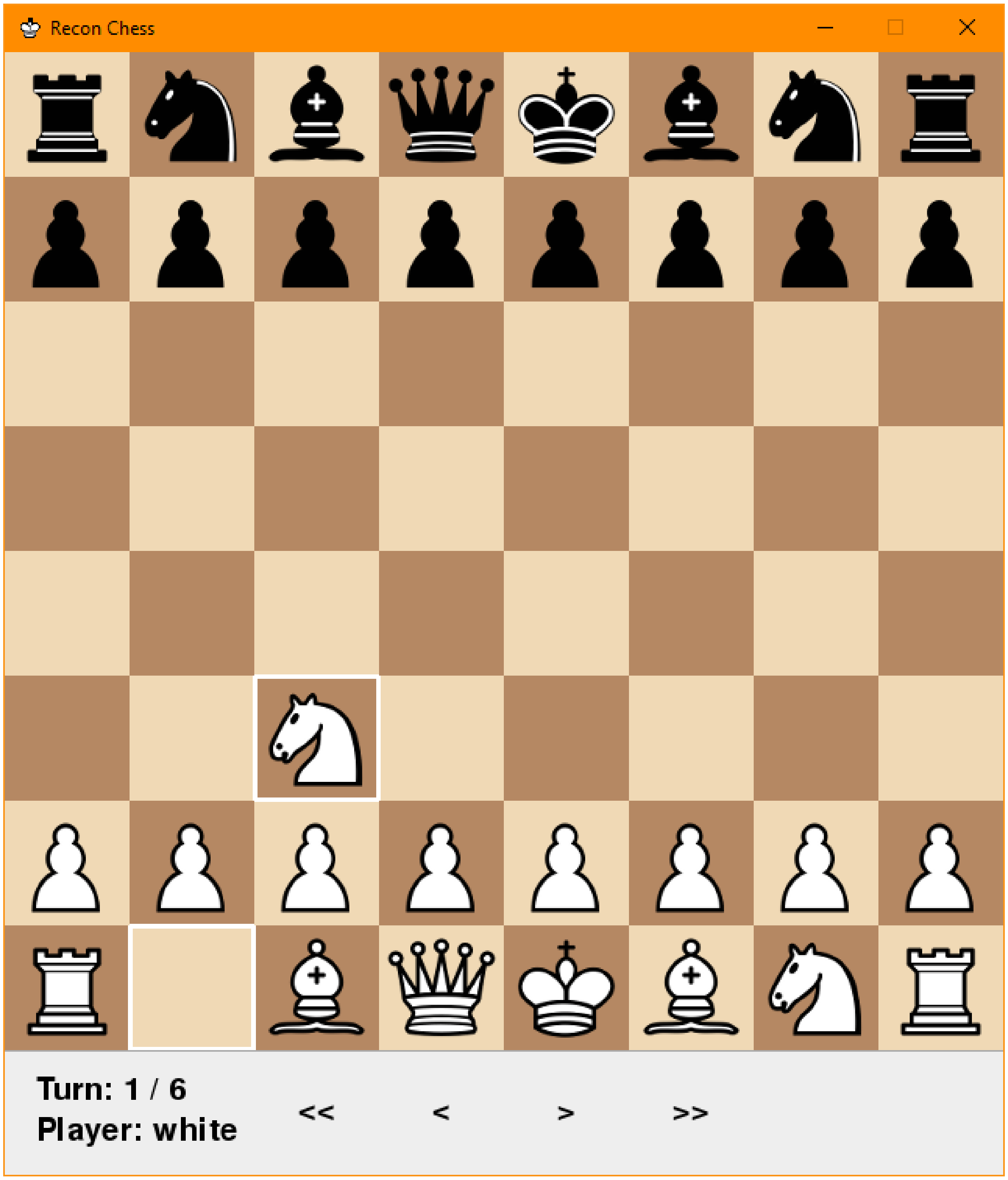
Here the black player senses over white's pawn at C2:



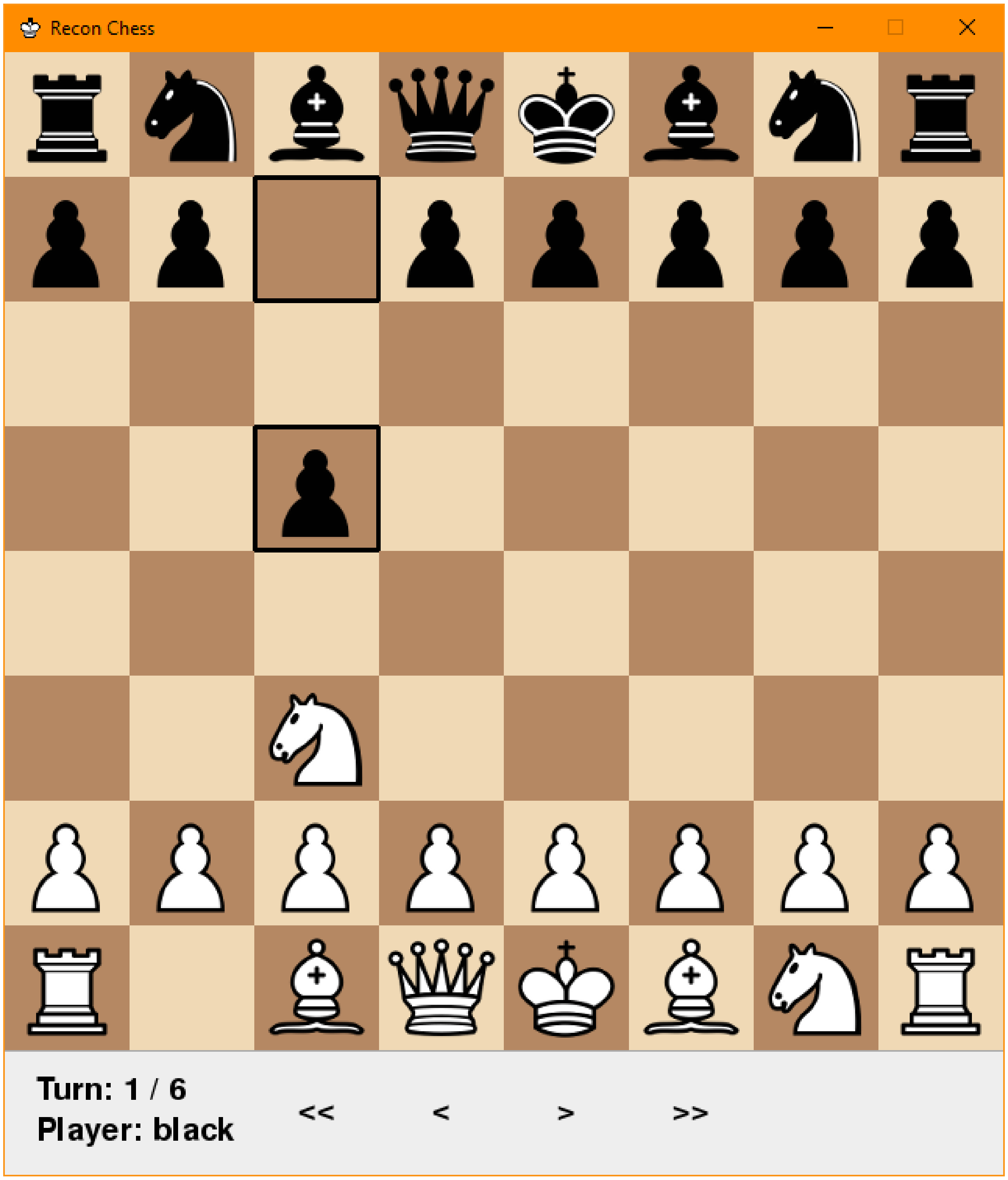
5.6.2 Move Action

Moves are shown by highlighting the square the piece moved from, and the square the piece moved to.

Here the white player moves their knight from B1 to C3:



Here the black player moves their pawn from C7 to C5:



5.6.3 Stepping through the game

There are 4 buttons at the bottom of the window that you can use to step through the game.

Going to Last Action

The >> button goes to the last action of the game.

Going to Start of Game

The << button goes to the start of the game.

Stepping Backward

The < button goes to the previous action taken.

Stepping Forward

The > button goes to the next action taken.

5.7 Debugging your bot

5.7.1 Diagnosing Errors

When your bot encounters an exception or error during a game, reproducing that error is essential to diagnosing and fixing it. The built in script `rc-playback` takes a game history, a bot, and the color the bot played as, and plays back the actions the bot took during the game. This allows you to exactly replicate the error producing conditions. Built in scripts like `rc-play` and `rc-bot-match` will produce game history files when an error occurs.

```
rc-playback --help
rc-playback <game history path> <bot> <color>
rc-playback bad_game.json src/my_awesome_bot.py white
```

Use the `--help` flag for more information about the arguments.

5.7.2 Debugging with PyCharm

You can create a run configuration to run your bot from PyCharm by targeting one of the scripts provided for running bots, like `reconchess.scripts.rc-bot-match` or `reconchess.scripts.rc-play`, as a module:

You can then choose to run the configuration you made in debug mode, and PyCharm will hit any breakpoints you set. It can do this because `rc-bot-match` and `rc-play` load your bot code into the same python process (see *`reconchess.load_player()`*).

5.7.3 Debugging with output

You can use ordinary print statements in your bot, and they will appear on the command line if you use `reconchess.scripts.rc-bot-match` or `reconchess.scripts.rc-play`. If you want to your output to go to a file, use a logging library (e.g. the built in logging module).

5.8 Connecting your bot to the server

5.8.1 Registering on the server

Before connecting your bot to the server, you'll need to register on the server. You can do this through the server's website, or by using one of the built in scripts.

If the registration is successful, you will receive an email from neurips_rbc_comp@listserv.jhuapl.edu about confirming your email. Click the link included in the email to verify your email.

Website registration

Visit https://rbc.jhuapl.edu/register to register.

Command Line registration

Use the built in script `rc-register`:

```
rc-register --help
rc-register
rc-register --username <username> --email <email> --affiliation <affiliation> --
↪password <password>
```

This script will prompt you for the username, email, affiliation, and password after you run it, or you can specify them with command line arguments.

Use the `--help` flag for more information about the arguments.

5.8.2 Connecting to the server

Note: By default rc-connect connects you to the server in unranked mode, so no ranked matches will be scheduled by default. See the Playing Ranked matches section for information on how to play ranked matches.

python

Use the built in script `rc-connect` to connect to the server and let your bot play games:

```
rc-connect --help
rc-connect <bot path>
rc-connect reconchess.bots.random_bot
rc-connect src/my_awesome_bot.py
```

Use the `--help` flag for more information about the arguments.

This script will prompt you for your username and password after you run it, or you can specify the username and password with command line arguments.

```
$ rc-connect src/my_awesome_bot.py
Username: my_awesome_bot
Password: ...
[<time>] Connected successfully to server!
```

```
$ rc-connect src/my_awesome_bot.py --username my_awesome_bot --password ...
[<time>] Connected successfully to server!
```

Other languages

If you are not using python or not using the reconchess package, you will need to implement logic to handle talking to the server. This is done through a RESTful HTTP API, and should be straightforward to implement.

See the *HTTP API* page for more information

If this applies to you please send us an email at **neurips_rbc_comp@listserv.jhuapl.edu** for help.

5.9 Playing Ranked Matches

Ranked matches allow you to quickly compare your bot to other bots with an ELO score based on the ranked matches you play.

Pass the `--ranked` flag to the built in script `rc-connect` to notify the server that you want to play ranked matches:

```
rc-connect --help
rc-connect --ranked <bot path>
rc-connect --ranked --keep-version <bot path>
rc-connect --ranked reconchess.bots.random_bot
```

Use the `--help` flag for more information about the arguments.

5.9.1 Disconnecting your bot

On termination, `rc-connect` will safely disconnect your bot from the server so that any in progress ranked matches are not aborted before they finish.

Pressing *ctrl+c* or sending a kill signal to the `rc-connect` process will cause the following to happen:

1. `rc-connect` will send a message to the server telling it to not schedule any more ranked games for you.
2. `rc-connect` will wait for any in progress games to finish.
3. Once all the in progress games are finished, it will exit.

Note if your bot opens new subprocesses (e.g. StockFish), you will have to make sure signals sent to `rc-connect` aren't also sent to the subprocesses. For example, if using StockFish with the python-chess library, you should open StockFish like this:

```
# the setpgrp=True flag will make sure the Stockfish process won't receive signals␣
↪sent to the rc-connect process.
chess.engine.SimpleEngine.popen_uci(stockfish_path, setpgrp=True)
```

5.9.2 Versioning

When you use the `--ranked` flag of `rc-connect`, it will prompt you about the version of your bot. Versions give you a way to track different updates to your bot.

The server tracks the ELO of each of your versions separately, so your old version's performance will not impact a new version's performance. In fact, when you create a new version the ELO starts from scratch to give you better accuracy.

There are two prompts you will have to answer when using the `--ranked` flag:

The first prompt asks whether you want to connect as a new version. Answer with `y` if you want to create a new version, and `n` otherwise.

```
> rc-connect --ranked src/my_awesome_bot.py
...
Is this a new version of your bot? [y/n]
```

The second prompt is a confirmation prompt and indicates the last version you connected as, and what version you will connect to the server as currently. Answer with `y` if you want to connect, and answering with `n` will exit the script.

```
> rc-connect --ranked src/my_awesome_bot.py
...
Are you sure you want to participate in ranked matches as v<new version> (currently v
↪<old version>)? [y/n]
```

Example of connecting to the server in ranked mode using the same version as last time:

```
> rc-connect --ranked src/my_awesome_bot.py
...
Is this a new version of your bot? [y/n]n
Are you sure you want to participate in ranked matches as v1 (currently v1)? [y/n]y
[<time stamp>] Connected successfully to server!
```

Example of connecting to the server in ranked mode as a new version:

```
> rc-connect --ranked src/my_awesome_bot.py
...
Is this a new version of your bot? [y/n]y
Are you sure you want to participate in ranked matches as v2 (currently v1)? [y/n]y
[<time stamp>] Connected successfully to server!
```

5.9.3 Other languages

If you are not using python or not using the reconchess package, you will need to implement logic to handle talking to the server. This is done through a RESTful HTTP API, and should be straightforward to implement.

See the *HTTP API* page for more information

If this applies to you please send us an email at **neurips_rbc_comp@listserv.jhuapl.edu** for help.

5.10 reconchess API

5.10.1 Types

class `reconchess.`**`Square`**

A type alias for an integer.

See `chess.A1`, `chess.B1`, . . . , and `chess.H8` for specifying specific squares.

See `chess.SQUARES` for referencing all squares.

class `reconchess.`**`Color`**

A type alias for a boolean.

See `chess.WHITE` and `chess.BLACK`.

`chess.WHITE` = True

`chess.BLACK` = False

class `reconchess.`**`PieceType`**

A type alias for an integer.

See `chess.PAWN`, `chess.KNIGHT`, `chess.BISHOP`, `chess.ROOK`, `chess.QUEEN`, `chess.KING` for specifying specific piece types

See `chess.PIECE_TYPES` for referencing all piece types.

class `reconchess.`**`WinReason`**

The reason the game ended

`KING_CAPTURE = 1`

The game ended because one player captured the other's king.

`TIMEOUT = 2`

The game ended because one player ran out of time.

`RESIGN = 3`

The game ended because one player resigned.

`TURN_LIMIT = 4`

The game ended because it exceeded the full turn limit.

`MOVE_LIMIT = 5`

The game ended because it exceeded the reversible move limit.

5.10.2 Player

class `reconchess.`**`Player`**

Base class of a player of Recon Chess. Implementation of a player is done by sub classing this class, and implementing each of the methods detailed below. For examples see *examples*.

The order in which each of the methods are called looks roughly like this:

1. *`handle_game_start()`*
2. *`handle_opponent_move_result()`*
3. *`choose_sense()`*
4. *`handle_sense_result()`*
5. *`choose_move()`*

6. *handle_move_result()*
7. *handle_game_end()*

Note that the *handle_game_start()* and *handle_game_end()* methods are only called at the start and the end of the game respectively. The rest are called repeatedly for each of your turns.

handle_game_start(*color: bool*, *board: chess.Board*, *opponent_name: str*)

Provides a place to initialize game wide structures like boards, and initialize data that depends on what color you are playing as.

Called when the game starts.

The board is provided to allow starting a game from different board positions.

Parameters

- **color** – The color that you are playing as. Either `chess.WHITE` or `chess.BLACK`.
- **board** – The initial board of the game. See `chess.Board`.
- **opponent_name** – The name of your opponent.

handle_opponent_move_result(*captured_my_piece: bool, capture_square: Optional[int]*)

Provides information about what happened on the opponents turn.

Called at the start of your turn.

Example implementation:

```
def handle_opponent_move_result(self, captured_my_piece: bool, capture_
↪square: Optional[Square]):
    if captured_my_piece:
        self.board.remove_piece_at(capture_square)
```

Parameters

- **captured_my_piece** – If the opponent captured one of your pieces, then *True*, otherwise *False*.
- **capture_square** – If a capture occurred, then the *Square* your piece was captured on, otherwise *None*.

choose_sense(*sense_actions: List[int], move_actions: List[chess.Move], seconds_left: float*) → Optional[int]

The method to implement your sensing strategy. The chosen sensing action should be returned from this function. I.e. the value returned is the square at the center of the 3x3 sensing area you want to sense. The returned square must be one of the squares in the *sense_actions* parameter.

You can pass instead of sensing by returning *None* from this function.

Move actions are provided through *move_actions* in case you want to sense based on a potential move.

Called after *handle_opponent_move_result()*.

Example implementation:

```
def choose_sense(self, sense_actions: List[Square], move_actions: List[chess.
↪Move], seconds_left: float) -> Square:
    return random.choice(sense_actions)
```

Parameters

- **sense_actions** – A `list` containing the valid squares to sense over.

- **move_actions** – A `list` containing the valid moves that can be returned in *choose_move()*.
- **seconds_left** – The time in seconds you have left to use in the game.

Returns a *Square* that is the center of the 3x3 sensing area you want to get information about.

handle_sense_result(*sense_result: List[Tuple[int, Optional[chess.Piece]]]*)

Provides the result of the sensing action. Each element in *sense_result* is a square and the corresponding `chess.Piece` found on that square. If there is no piece on the square, then the piece will be *None*.

Called after *choose_sense()*.

Example implementation:

```
def handle_sense_result(self, sense_result: List[Tuple[Square, Optional[chess.
↪Piece]]]):
    for square, piece in sense_result:
        if piece is None:
            self.board.remove_piece_at(square)
        else:
            self.board.set_piece_at(square, piece)
```

Parameters **sense_result** – The result of the sense. A *list* of *Square* and an optional `chess.Piece`.

choose_move(*move_actions: List[chess.Move], seconds_left: float*) → Optional[chess.Move]

The method to implement your movement strategy. The chosen movement action should be returned from this function. I.e. the value returned is the move to make. The returned move must be one of the moves in the *move_actions* parameter.

The pass move is legal, and is executed by returning *None* from this method.

Called after *handle_sense_result()*.

Example implementation:

```
def choose_move(self, move_actions: List[chess.Move], seconds_left: float) ->␣
↪Optional[chess.Move]:
    return random.choice(move_actions)
```

Parameters

- **move_actions** – A *list* containing the valid `chess.Move` you can choose.
- **seconds_left** – The time in seconds you have left to use in the game.

Returns The `chess.Move` to make.

handle_move_result(*requested_move: Optional[chess.Move], taken_move: Optional[chess.Move], captured_opponent_piece: bool, capture_square: Optional[int]*)

Provides the result of the movement action. The *requested_move* is the move returned from *choose_move()*, and is provided for ease of use. *taken_move* is the move that was actually performed. Note that *taken_move*, can be different from *requested_move*, due to the uncertainty aspect.

Called after *choose_move()*.

Example implementation:

```
def handle_move_result(self, requested_move: chess.Move, taken_move: chess.
↪Move,
           captured_opponent_piece: bool, capture_square: Optional[Square]):
    if taken_move is not None:
        self.board.push(taken_move)
```

Note: In the case of playing games on a server, this method is invoked during your opponents turn. This means in most cases this method will not use your play time. However if the opponent finishes their turn before this method completes, then time will be counted against you.

Parameters

- **`requested_move`** – The `chess.Move` you requested in *`choose_move()`*.
- **`taken_move`** – The `chess.Move` that was actually applied by the game if it was a valid move, otherwise *None*.
- **`captured_opponent_piece`** – If *taken_move* resulted in a capture, then *True*, otherwise *False*.
- **`capture_square`** – If a capture occurred, then the *`Square`* that the opponent piece was taken on, otherwise *None*.

`handle_game_end`(*winner_color: Optional[bool], win_reason: Optional[reconchess.types.WinReason], game_history: reconchess.history.GameHistory*)

Provides the results of the game when it ends. You can use this for post processing the results of the game.

Parameters

- **`winner_color`** – If the game was a draw, then *None*, otherwise, the color of the player who won the game.
- **`win_reason`** – If the game was a draw, then *None*, otherwise the reason the game ended specified as *`WinReason`*
- **`game_history`** – *`GameHistory`* object for the game, from which you can get the actions each side has taken over the course of the game.

`reconchess.`**`load_player`**(*source_path: str*) → Tuple[str, Type[reconchess.player.Player]]

Loads a subclass of the Player class that is contained in a python source file or python module. There should only be 1 such subclass in the file or module. If there are more than 1 subclasses, then you have to define a function named *get_player* in the same module that returns the subclass to use.

Example of single class definition:

```
# this will import fine
class MyBot(Player):
    ...
```

Example of multiple class definition:

```
class MyBot1(Player):
    ...

class MyBot2(Player):
    ...

# need to define this function!
def get_player():
    return MyBot1
```

Example of another situation where you may need to define *get_player*:

```
from my_helper_module import MyPlayerBaseClass

class MyBot1(MyPlayerBaseClass):
    ...

# you need to define this because both MyBot1 and MyPlayerBaseClass are␣
↪subclasses of Player
def get_player():
    return MyBot1
```

Example usage:

```
name, cls = load_player('my_player.py')
player = cls()

name, cls = load_player('reconchess.bots.random_bot')
player = cls()
```

Parameters **source_path** – the path to the source file to load

Returns Tuple where the first element is the name of the loaded class, and the second element is the class type

5.10.3 Game

class reconchess.**Game**

Abstract class that represents an instantiation of a Reconnaissance Chess Game. See *LocalGame* and *RemoteGame* for implementations.

sense_actions() → List[int]

Returns List of *Square* that the player can choose for sense phase.

move_actions() → List[chess.Move]

Returns List of chess.Move that the player can choose for move phase.

get_seconds_left() → float

Returns float indicating the seconds the player has left to play.

start()

Starts the game and the timers for each player.

opponent_move_results() → Optional[int]

Returns *Square* where opponent captured a piece last turn if they did, otherwise *None*.

sense(*square: Optional[int]*) → List[Tuple[int, Optional[chess.Piece]]]

Execute a sense action and get the sense result.

An example result returned from sensing *B7* at the beginning of the game:

```
[
    (A8, Piece(ROOK, BLACK)), (B8, Piece(KNIGHT, BLACK)), (C8, Piece(BISHOP,␣
↪BLACK)),
    (A7, Piece(PAWN, BLACK)), (B7, Piece(PAWN, BLACK)), (C7, Piece(PAWN,␣
↪BLACK)),
```

(continues on next page)

(continued from previous page)

```
    (A6, None), (B6, None), (C8, None)
]
```

Parameters **square** – The *Square* to sense.

Returns A list of tuples, where each tuple contains a *Square* in the sense, and if there was a piece on the square, then the corresponding `chess.Piece`, otherwise *None*.

move(*requested_move: Optional[chess.Move]*) → Tuple[Optional[chess.Move], Optional[chess.Move], Optional[int]]
Execute a move action and get the result.

Parameters **requested_move** – The `chess.Move` to execute.

Returns A tuple containing the requested `chess.Move`, the taken `chess.Move`, and the *Square* that a capture occurred on if one occurred.

end_turn()
End the current player's turn.

is_over() → bool
The game is over if either player has run out of time, or if either player's King has been captured.

This will always return *True* after `end()` has been called.

Returns Returns *True* if the game is over, otherwise *False*.

get_winner_color() → Optional[bool]
Returns the color of the player who won the game. If the game is not over, or is over but does not have a winner (i.e. `end()` has been called), then this will return *None*.

Returns *Color* of the winner if the game has ended and has a winner, otherwise *None*.

get_win_reason() → Optional[reconchess.types.WinReason]
Returns the reason the player who won won the game. If the game is not over, or is over but does not have a winner (i.e. `end()` has been called), then this will return *None*.

Returns *WinReason* of the winner if the game has ended and has a winner, otherwise *None*.

get_game_history() → Optional[reconchess.history.GameHistory]
Get the history of the game.

Returns *GameHistory* if the game is over, otherwise *None*.

class reconchess.**LocalGame**(*seconds_per_player: Optional[float] = 900*, *seconds_increment: Optional[float] = 5*, *reversible_moves_limit: Optional[int] = 100*, *full_turn_limit: Optional[int] = None*)
The local implementation of *Game*. Used to run games locally instead of remotely via a server.

__init__(*seconds_per_player: Optional[float] = 900*, *seconds_increment: Optional[float] = 5*, *reversible_moves_limit: Optional[int] = 100*, *full_turn_limit: Optional[int] = None*)
Constructs the Game object

Parameters

- **seconds_per_player** – Initial clock limit for each player in seconds. Use None for unlimited. Default is 900.
- **seconds_increment** – Seconds added to a player's clock limit on each turn. Will treat None as 0. Default is 5.

- **reversible_moves_limit** – Maximum number of moves without pawn moves or captures before game is a draw. Use None for unlimited. Default is 100 (a non-optional version of the "50-move rule").
- **full_turn_limit** – Maximum number of full turns (both players move) before game is a draw. Use None for unlimited. Default is None.

`class reconchess.RemoteGame(server_url, game_id, auth)`

The remote implementation of `Game`. Used to play games remotely via a server.

All the methods implemented are pass-throughs to the server. Each method submits a HTTP request to the corresponding end point on the server.

5.10.4 GameHistory

`class reconchess.Turn(color: bool, turn_number: int)`

The representation of a single turn in a game. Contains the color of the player who played this turn, as well as the number of turns the player has taken so far.

next

Returns The `Turn` that happens immediately after this, which is the other player's next turn.

previous

Returns The :class: *Turn* that happens immediately before this, which is the other player's previous turn.

`class reconchess.GameHistory`

Implements method for processing and querying a Game.

Here are some example uses:

Extracting data to train a sensing policy to sense opponent:

```
opponent_moves = history.collect(history.taken_move, history.turns(not self.
→color))
target_senses = [move.to_square for move in opponent_moves]
```

Extracting data to train a movement policy:

```
for turn in history.turns(self.color):
    if history.has_move(turn):
        move = history.requested_move(turn)
        board = history.truth_board_before_move(turn)
        # do something with move and board...
```

Randomly sampling from the turns instead of using them in sequential order:

```
for turn in random.sample(history.turns(self.color), N):
    ...
```

Giving rewards based on validity of the move:

```
for turn in history.turns(self.color):
    if history.has_move(turn):
        if history.requested_move(turn) != history.taken_move(turn):
            reward = -1
        else:
            reward = 1
```

Giving rewards for moving out of check:

```
for turn in history.turns(self.color):
    if history.has_move(turn):
        board_before = history.truth_board_before_move(turn)
        board_after = history.truth_board_after_move(turn)

        if board_before.is_check() and not board_after.is_check()
            reward = 1
        else:
            reward = -1
```

save (*filename*)
Save the game history to a json file.

Parameters **filename** – The file to save to.

classmethod from_file (*filename*)

Parameters **filename** – The json file to load the *GameHistory* object from.

Returns The *GameHistory* object that was originally saved to the file using *save()*.

get_white_player_name () → str
Get the name of white. :return: str

get_black_player_name () → str
Get the name of black. :return: str

is_empty () → bool
Get whether or not the game had any turns in it.

Examples:

```
>>> history.is_empty()
False
```

Returns *True* if there are no turns to query in this object, *False* otherwise.

num_turns (*color: bool = None*) → int
Get the number of turns taken in the game. Optionally specify the color of the player to get the number of turns for.

Examples:

```
>>> history.num_turns()
9
```

```
>>> history.num_turns(WHITE)
5
```

```
>>> history.num_turns(BLACK)
4
```

Parameters **color** – Optional player color indicating which player's number of turns to return.

Returns The number of turns saved in this object. If *color* is specified, get the number of turns for that player.

turns (*color: bool = None*, *start=0*, *stop=inf*) → Iterable[reconchess.history.Turn]
Get all the turns that happened in the game in order. Optionally specify a single player to get only that player's turns.

Examples:

```
>>> list(history.turns())
[Turn(WHITE, 0), Turn(BLACK, 0), Turn(WHITE, 1), ..., Turn(BLACK, 23)]
```

```
>>> list(history.turns(WHITE))
[Turn(WHITE, 0), Turn(WHITE, 1), ..., Turn(WHITE, 22)]
```

```
>>> list(history.turns(BLACK))
[Turn(BLACK, 0), Turn(BLACK, 1), ..., Turn(BLACK, 23)]
```

```
>>> list(history.turns(start=1))
[Turn(WHITE, 1), Turn(BLACK, 1), Turn(WHITE, 2), ..., Turn(BLACK, 23)]
```

```
>>> list(history.turns(stop=2))
[Turn(WHITE, 0), Turn(BLACK, 0), Turn(WHITE, 1), Turn(BLACK, 1)]
```

```
>>> list(history.turns(WHITE, stop=2))
[Turn(WHITE, 0), Turn(WHITE, 1)]
```

```
>>> list(history.turns(start=1, stop=2))
[Turn(WHITE, 1), Turn(BLACK, 1)]
```

Parameters

- **color** – Optional player color indicating which player's turns to return.
- **start** – Optional starting turn number.
- **stop** – Optional stopping turn number.

Returns An iterable of *Turn* objects that are in the same order as they occurred in the game. If *color* is specified, gets the turns only for that player.

is_first_turn (*turn: reconchess.history.Turn*)
Checks whether *turn* is the first turn of the game.

Examples:

```
>>> history.is_first_turn(Turn(BLACK, 0))
False
```

```
>>> history.is_first_turn(Turn(WHITE, 0))
True
```

Parameters **turn** – the *Turn* in question.

Returns *True* if *turn* is the first turn in the game, *False* otherwise.

first_turn (*color: bool = None*) → reconchess.history.Turn
Gets the first turn of the game.

Examples:

```
>>> history.first_turn()
Turn(WHITE, 0)
```

```
>>> history.first_turn(WHITE)
Turn(WHITE, 0)
```

```
>>> history.first_turn(BLACK)
Turn(BLACK, 0)
```

Parameters **`color`** – Optional color indicating which player's first turn to return.

Returns The *`Turn`* that is the first turn in the game.

`is_last_turn`(*turn: reconchess.history.Turn*)
Checks whether *turn* is the last turn of the game.

Examples:

```
>>> history.is_last_turn(Turn(BLACK, 23))
False
```

```
>>> history.is_last_turn(Turn(WHITE, 24))
True
```

Parameters **`turn`** – the *`Turn`* in question.

Returns *True* if *turn* is the last turn in the game, *False* otherwise.

`last_turn`(*color: bool = None*) → reconchess.history.Turn
Gets the last turn of the game.

Examples:

```
>>> history.last_turn()
Turn(WHITE, 24)
```

```
>>> history.first_turn(WHITE)
Turn(WHITE, 24)
```

```
>>> history.first_turn(BLACK)
Turn(BLACK, 23)
```

Parameters **`color`** – Optional color indicating which player's last turn to return.

Returns The *`Turn`* that is the last turn in the game.

`get_winner_color`() → Optional[bool]

Returns the color of the player who won the game. If the game is not over, or is over but does not have a winner
then this will return *None*.

Returns *`Color`* of the winner if the game has ended and has a winner, otherwise *None*.

get_win_reason() → Optional[reconchess.types.WinReason]
Returns the reason the player who won won the game. If the game is not over, or is over but does not have a winner, then this will return *None*.

Returns `WinReason` of the winner if the game has ended and has a winner, otherwise *None*.

has_sense(*turn: reconchess.history.Turn*) → bool
Checks to see if the game has a sense action for the specified turn. The game may not if it ended because of timeout.

This intended use is to call this before calling `sense()` and `sense_result()`, to verify that there is a sense before querying for it.

Examples:

```
>>> history.has_sense(Turn(WHITE, 0))
True
```

```
>>> history.has_sense(Turn(WHITE, 432))
False
```

Parameters **turn** – The `Turn` in question.

Returns *True* if there is a sense action, *False* otherwise.

sense(*turn: reconchess.history.Turn*) → Optional[int]
Get the sense action on the given turn.

Examples:

```
>>> history.sense(Turn(WHITE, 0))
E7
```

```
>>> history.sense(Turn(BLACK, 0))
E2
```

```
>>> history.sense(Turn(WHITE, 1))
None
```

Parameters **turn** – The `Turn` in question.

Returns The executed sense action as a `Square`.

sense_result(*turn: reconchess.history.Turn*) → List[Tuple[int, Optional[chess.Piece]]]
Get the result of the sense action on the given turn.

Examples:

```
>>> history.sense(Turn(WHITE, 0))
B7
>>> history.sense_result(Turn(WHITE, 0))
[
    (A8, Piece(ROOK, BLACK)), (B8, Piece(KNIGHT, BLACK)), (C8,␣
↪Piece(BISHOP, BLACK)),
    (A7, Piece(PAWN, BLACK)), (B7, Piece(PAWN, BLACK)), (C7, Piece(PAWN,␣
↪BLACK)),
    (A6, None), (B6, None), (C8, None)
```

(continues on next page)

(continued from previous page)

```
]
>>> history.sense(Turn(BLACK, 0))
None
>>> history.sense_result(Turn(BLACK, 0))
[]
```

Parameters **turn** – The *Turn* in question.

Returns The result of the executed sense action.

has_move(*turn: reconchess.history.Turn*) → bool

Checks to see if the game has a move action for the specified turn. The game may not if it ended because of timeout.

This intended use is to call this before calling *requested_move()*, *taken_move()*, *capture_square()*, and *move_result()* to verify that there is a move before querying for it.

Examples:

```
>>> history.has_move(Turn(WHITE, 0))
True
```

```
>>> history.has_move(Turn(WHITE, 432))
False
```

Parameters **turn** – The *Turn* in question.

Returns *True* if there is a move action, *False* otherwise.

requested_move(*turn: reconchess.history.Turn*) → Optional[chess.Move]

Get the requested move action on the given turn.

Examples:

```
>>> history.requested_move(Turn(WHITE, 0))
Move(E2, E4)
```

```
>>> history.requested_move(Turn(BLACK, 0))
Move(E7, E5)
```

```
>>> history.requested_move(Turn(WHITE, 1))
None
```

Parameters **turn** – The *Turn* in question.

Returns If the player requested to move, then the requested move action as a `chess.Move`, otherwise *None* if the player requested to pass.

taken_move(*turn: reconchess.history.Turn*) → Optional[chess.Move]

Get the executed move action on the given turn. This may be different than the requested move, as the requested move may not be legal.

Examples:

```
>>> history.requested_move(Turn(WHITE, 0))
Move(E2, D3)
>>> history.taken_move(Turn(WHITE, 0))
None
```

```
>>> history.requested_move(Turn(WHITE, 1))
Move(E2, E4)
>>> history.taken_move(Turn(WHITE, 1))
Move(E2, E3)
```

Parameters **`turn`** – The `Turn` in question.

Returns *None* if the player requested to pass or made an illegal move, the executed move action as a `chess.Move` otherwise.

`capture_square`(*turn: reconchess.history.Turn*) → Optional[int]

Get the square of the opponent's captured piece on the given turn. A capture may not have occurred, in which case *None* will be returned.

Examples:

```
>>> history.capture_square(Turn(WHITE, 0))
None
```

```
>>> history.capture_square(Turn(WHITE, 4))
E4
```

Parameters **`turn`** – The `Turn` in question.

Returns If a capture occurred, then the `Square` where it occurred, otherwise *None*.

`move_result`(*turn: reconchess.history.Turn*) → Tuple[Optional[chess.Move], Optional[chess.Move], Optional[int]]

Get the full move result in one function. Calls `requested_move()`, `taken_move()`, and `capture_square()`.

Examples:

```
>>> history.move_result(Turn(WHITE, 0))
(Move(E2, E4), Move(E2, E3), None)
```

Parameters **`turn`** – The `Turn` in question.

Returns The requested move, the executed move, and a capture square if there was one.

`truth_fen_before_move`(*turn: reconchess.history.Turn*) → str

Get the truth state of the board as a fen string before the move was executed on the given turn. Use `truth_board_before_move()` if you want the truth board as a `chess.Board` object.

Examples:

```
>>> history.truth_fen_before_move(Turn(WHITE, 0))
"rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq -"
>>> history.taken_move(Turn(WHITE, 0))
Move(E2, E4)
```

(continues on next page)

(continued from previous page)

```
>>> history.truth_fen_before_move(Turn(BLACK, 0))
"rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR w KQkq -"
```

Parameters **turn** – The *Turn* in question.

Returns The fen of the truth board.

truth_board_before_move(*turn: reconchess.history.Turn*) → chess.Board

Get the truth state of the board as a `chess.Board` before the move was executed on the given turn. Use *truth_fen_before_move()* if you want the truth board as a fen string.

Examples:

```
>>> history.truth_board_before_move(Turn(WHITE, 0))
Board("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq -")
>>> history.taken_move(Turn(WHITE, 0))
Move(E2, E4)
>>> history.truth_fen_before_move(Turn(BLACK, 0))
Board("rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR w KQkq -")
```

Parameters **turn** – The *Turn* in question.

Returns A `chess.Board` object.

truth_fen_after_move(*turn: reconchess.history.Turn*) → str

Get the truth state of the board as a fen string after the move was executed on the given turn. Use *truth_board_after_move()* if you want the truth board as a `chess.Board` object.

Examples:

```
>>> history.truth_fen_before_move(Turn(WHITE, 0))
"rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq -"
>>> history.taken_move(Turn(WHITE, 0))
Move(E2, E4)
>>> history.truth_fen_after_move(Turn(WHITE, 0))
"rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR w KQkq -"
```

Parameters **turn** – The *Turn* in question.

Returns The fen of the truth board.

truth_board_after_move(*turn: reconchess.history.Turn*) → chess.Board

Get the truth state of the board as a `chess.Board` after the move was executed on the given turn. Use *truth_fen_after_move()* if you want the truth board as a fen string.

Examples:

```
>>> history.truth_board_before_move(Turn(WHITE, 0))
Board("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq -")
>>> history.taken_move(Turn(WHITE, 0))
Move(E2, E4)
>>> history.truth_fen_after_move(Turn(WHITE, 0))
Board("rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR w KQkq -")
```

Parameters **turn** – The *Turn* in question.

Returns A `chess.Board` object.

collect (*get_turn_data_fn: Callable[[reconchess.history.Turn], T], turns: Iterable[reconchess.history.Turn]*) → Iterable[T]

Collect data from multiple turns using any of *sense()*, *sense_result()*, *requested_move()*, *taken_move()*, *capture_square()*, *move_result()*, *truth_fen_before_move()*, *truth_board_before_move()*, *truth_fen_after_move()*, or *truth_board_after_move()*.

Examples:

```
>>> history.collect(history.sense, [Turn(WHITE, 0), Turn(BLACK, 0)])
[E7, E2]
```

```
>>> history.collect(history.requested_move, history.turns(WHITE))
[Move(E2, E4), Move(D1, H5), ...]
```

Parameters

- **get_turn_data_fn** – One of the getter functions of the history object.
- **turns** – The turns in question.

Returns A list of the data, where each element is the value of the getter function on the corresponding turn.

5.10.5 Functions for playing games

reconchess.**play_local_game** (*white_player: reconchess.player.Player, black_player: reconchess.player.Player, game: reconchess.game.LocalGame = None, seconds_per_player: float = 900*) → Tuple[Optional[bool], Optional[reconchess.types.WinReason], reconchess.history.GameHistory]

Plays a game between the two players passed in. Uses *LocalGame* to run the game, and just calls *play_turn()* until the game is over:

```
while not game.is_over():
    play_turn(game, player)
```

Parameters

- **white_player** – The white *Player*.
- **black_player** – The black *Player*.
- **game** – The *LocalGame* object to use.
- **seconds_per_player** – The time each player has to play. Only used if *game* is not passed in.

Returns The results of the game, also passed to each player via *Player.handle_game_end()*.

reconchess.**play_remote_game** (*server_url, game_id, auth, player: reconchess.player.Player*)

reconchess.**play_turn** (*game: reconchess.game.Game, player: reconchess.player.Player, end_turn_last=False*)

Coordinates playing a turn for *player* in *game*. Does the following sequentially:

1. *notify_opponent_move_results()*

2. `play_sense()`
3. `play_move()`

See `play_move()` for more info on *end_turn_last*.

Parameters

- **game** – The `Game` that *player* is playing in.
- **player** – The `Player` whose turn it is.
- **end_turn_last** – Flag indicating whether to call `Game.end_turn()` before or after `Player.handle_move_result()`

`reconchess.`**`notify_opponent_move_results`**(*game: reconchess.game.Game*, *player: reconchess.player.Player*)

Passes the opponents move results to the player. Does the following sequentially:

1. Get the results of the opponents move using `Game.opponent_move_results()`.
2. Give the results to the player using `Player.handle_opponent_move_result()`.

Parameters

- **game** – The `Game` that *player* is playing in.
- **player** – The `Player` whose turn it is.

`reconchess.`**`play_sense`**(*game: reconchess.game.Game*, *player: reconchess.player.Player*, *sense_actions: List[int]*, *move_actions: List[chess.Move]*)

Runs the sense phase for *player* in *game*. Does the following sequentially:

1. Get the sensing action using `Player.choose_sense()`.
2. Apply the sense action using `Game.sense()`.
3. Give the result of the sense action to player using `Player.handle_sense_result()`.

Parameters

- **game** – The `Game` that *player* is playing in.
- **player** – The `Player` whose turn it is.
- **sense_actions** – The possible sense actions for *player*.
- **move_actions** – The possible move actions for *player*.

`reconchess.`**`play_move`**(*game: reconchess.game.Game*, *player: reconchess.player.Player*, *move_actions: List[chess.Move]*, *end_turn_last=False*)

Runs the move phase for *player* in *game*. Does the following sequentially:

1. Get the moving action using `Player.choose_move()`.
2. Apply the moving action using `Game.move()`.
3. Ends the current player's turn using `Game.end_turn()`.
4. Give the result of the moveaction to player using `Player.handle_move_result()`.

If *end_turn_last* is True, then `Game.end_turn()` is called last instead of before `Player.handle_move_result()`.

Parameters

- **game** – The `Game` that *player* is playing in.

- **player** – The *Player* whose turn it is.
- **move_actions** – The possible move actions for *player*.
- **end_turn_last** – Flag indicating whether to call *Game.end_turn()* before or after *Player.handle_move_result()*

5.11 HTTP API

This is the HTTP API of the reconchess server, and is used when you play a remote game. We provide scripts and code to use this HTTP API with the python package by default, but if you are using another language you will need to make HTTP requests to these endpoints to play remote games.

5.11.1 Authorization

The HTTP API uses basic authorization, so you will need to provide the username and password in the Authorization header.

5.11.2 User Endpoints

Endpoints for querying users and updating data for yourself.

GET /api/users/

Get all active users.

Example response content:

```
{
    "usernames": ["foouser", "baruser"]
}
```

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **usernames** (*array<string>*) – usernames of active users.

Status Codes

- 200 OK – Success.
- 401 Unauthorized – Invalid or empty authentication information.

POST /api/users/me

Ping the server to update your connection time.

Example response content:

```
{
    "id": 1,
    "username": "foouser",
    "max_games": 4
}
```

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **id** (*integer*) – Your ID.
- **username** (*string*) – Your username.
- **max_games** (*integer*) – The maximum number of games you can play at one time.

Status Codes

- 200 OK – Success.
- 401 Unauthorized – Invalid or empty authentication information.

POST /api/users/me/max_games

Update the number of max_games you can play at one time.

Example request content:

```
{
    "max_games": 5
}
```

Example response content:

```
{
    "id": 1,
    "username": "foouser",
    "max_games": 5
}
```

Request Headers

- Authorization – Basic Authorization.

Request JSON Object

- **max_games** (*integer*) – The maximum number of games you can play at one time.

Response JSON Object

- **id** (*integer*) – Your ID.
- **username** (*string*) – Your username.
- **max_games** (*integer*) – The maximum number of games you can play at one time.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Invalid request (max_games not present or not an integer).
- 401 Unauthorized – Invalid or empty authentication information.

POST /api/users/me/ranked

Update whether you want to participate in ranked matches or not.

Example request content:

```
{
    "ranked": true
}
```

Example response content:

```
{
    "id": 1,
    "username": "foouser",
    "ranked": true
}
```

Request Headers

- Authorization – Basic Authorization.

Request JSON Object

- **ranked** (*boolean*) – Whether you want to participate in ranked matches or not.

Response JSON Object

- **id** (*integer*) – Your ID.
- **username** (*string*) – Your username.
- **ranked** (*boolean*) – Whether you want to participate in ranked matches or not.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Invalid request (ranked not present or not a boolean).
- 401 Unauthorized – Invalid or empty authentication information.

POST /api/users/me/version

Create a new version of your bot for ranked matches. If no versions exist, this creates version 1, otherwise it increments the last version for your bot.

Example response content:

```
{
    "id": 1,
    "username": "foouser",
    "version": 10
}
```

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **id** (*integer*) – Your ID.
- **username** (*string*) – Your username.
- **version** (*integer*) – The new version number for your bot.

Status Codes

- 200 OK – Success.

- 400 Bad Request – Invalid request (ranked not present or not a boolean).
- 401 Unauthorized – Invalid or empty authentication information.

5.11.3 Invitation Endpoints

The invitation endpoints allow you to send and receive invitations to play games. Example usage can be seen in the `rc_connect` script.

`GET /api/invitations/`

Unaccepted invitations sent to you from other players.

Example response content:

```
{
    "invitations": [1, 2, 5]
}
```

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **`invitations`** (*`array<integer>`*) – id's of your unaccepted invitations.

Status Codes

- 200 OK – Success.
- 401 Unauthorized – Invalid or empty authentication information.

`POST /api/invitations/`

Send an invitation to another player.

Example request content:

```
{
    "opponent": "thatguy",
    "color": true
}
```

Example response content:

```
{
    "game_id": 1
}
```

Request Headers

- Authorization – Basic Authorization.

Request JSON Object

- **`opponent`** (*`string`*) – The name of the player to send the invitation to.
- **`color`** (*`boolean`*) – The color you want to play - `true` for White and `false` for Black.

Response JSON Object

- **`game_id`** (*`integer`*) – The game ID of the resulting game.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Invitation does not exist.
- 401 Unauthorized – Invalid or empty authentication information.

POST /api/invitations/(**int:** *invitation_id*)

Accept the *invitation_id* invitation.

Example response content:

```
{
    "game_id": 1
}
```

Parameters

- **invitation** – The ID of the invitation.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **game_id** (*integer*) – The game ID of the resulting game.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Invitation does not exist.
- 401 Unauthorized – Invalid or empty authentication information.

POST /api/invitations/(**int:** *invitation_id*)**/finish**

Mark the *invitation_id* invitation as finished.

Parameters

- **invitation** – The ID of the invitation.

Request Headers

- Authorization – Basic Authorization.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Invitation does not exist or invitation is not accepted.
- 401 Unauthorized – Invalid or empty authentication information.

5.11.4 Game Endpoints

The game endpoints allow you to send actions to the server and receive their results. Example usage can be seen in the implementation of `reconchess.RemoteGame`.

GET /api/games/(**int:** *game_id*)**/color**

Get the color you are playing as in game *game_id*.

Example response content:

```
{
    "color": true
}
```

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **color** (*boolean*) – The color you are playing as - `true` for White and `false` for Black.

Status Codes

- 200 OK – Success.
- 401 Unauthorized – Invalid or empty authentication information, or not a player in the specified game.
- 404 Not Found – Game does not exist.

GET /api/games/(int: *game_id*)/starting_board

Get the starting board for game *game_id*.

Example response content:

```
{
    "board": {
        "type": "Board",
        "value": "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"
    }
}
```

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **board** (*object*) – The starting board.
- **type** (*string*) – `"Board"`.
- **value** (*string*) – The fen string of the chess board.

Status Codes

- 200 OK – Success.
- 401 Unauthorized – Invalid or empty authentication information, or not a player in the specified game.
- 404 Not Found – Game does not exist.

GET /api/games/(int: *game_id*)/opponent_name

Get the name of your opponent for game *game_id*.

Example response content:

```
{
    "opponent_name": "super evil dude 123"
}
```

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **opponent_name** (*string*) – The name of the opponent.

Status Codes

- 200 OK – Success.
- 401 Unauthorized – Invalid or empty authentication information, or not a player in the specified game.
- 404 Not Found – Game does not exist.

POST /api/games/(int: *game_id*)/ready

Mark yourself as ready to start the game.

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Player already marked as ready.
- 401 Unauthorized – Invalid or empty authentication information, or not a player in the specified game.
- 404 Not Found – Game does not exist.

GET /api/games/(int: *game_id*)/sense_actions

Get the sense actions you can take. See *reconchess.Game.sense_actions()*.

Example response content:

```
{
    "sense_actions": [1, 2, 3, 4]
}
```

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **sense_actions** (*array<integer>*) – A list of squares you can sense.

Status Codes

- 200 OK – Success.
- 401 Unauthorized – Invalid or empty authentication information, or not a player in the specified game.
- 404 Not Found – Game does not exist.

GET /api/games/(int: *game_id*)/move_actions

Get the move actions you can take. See *reconchess.Game.move_actions()*.

Example response content:

```
{
    "move_actions": [
        {"type": "Move", "value": "e2e4"},
        {"type": "Move", "value": "a7a8q"}
    ]
}
```

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **move_actions** (*object*) – A list of the moves you can make.
- **type** (*string*) – "Move".
- **value** (*string*) – The chess move encoded as a UCI string.

Status Codes

- 200 OK – Success.
- 401 Unauthorized – Invalid or empty authentication information, or not a player in the specified game.
- 404 Not Found – Game does not exist.

GET /api/games/(int: *game_id*)/seconds_left

Gets the number of seconds you have left to play. See *reconchess.Game.get_seconds_left()*.

Example response content:

```
{
    "seconds_left": 50
}
```

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **seconds_left** (*float*) – The time you have left to play.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Game is finished.
- 401 Unauthorized – Invalid or empty authentication information, or not a player in the specified game.
- 404 Not Found – Game does not exist.

GET /api/games/(int: *game_id*)/opponent_move_results

Get the result of the opponent's last move. See `reconchess.Game.opponent_move_results()`.

Example response content:

```
{
    "opponent_move_results": 34
}
```

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **opponent_move_results** (*Optional<integer>*) – The square the opponent captured one of your pieces on. `null` if no capture occurred.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Game is finished.
- 401 Unauthorized – Invalid or empty authentication information, or not a player in the specified game.
- 404 Not Found – Game does not exist.

POST /api/games/(int: *game_id*)/sense

Perform a sense action. See `reconchess.Game.sense()`.

Example request content:

```
{
    "square": 5
}
```

Example response content:

```
{
    "sense_result": [
        [54, {"type": "Piece", "value": "p"}],
        [55, null],
        [56, {"type": "Piece", "value": "K"}]
    ]
}
```

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Request JSON Object

- **square** (*integer*) – The square you want to sense.

Response JSON Object

- **sense_result** (*object*) – The list of squares and pieces found from your sense.
- **type** (*string*) – `Piece`.
- **value** (*Optional<string>*) – The symbol of the piece found at the square. `null` if no piece is there.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Game is finished, you already sensed, or malformed request data.
- 401 Unauthorized – Invalid or empty authentication information, or not a player in the specified game.
- 404 Not Found – Game does not exist.

POST /api/games/(int: *game_id*)/move

Perform a move action. See *reconchess.Game.move()*.

Example request content:

```
{
    "requested_move": {"type": "Move", "value": "e2e4"}
}
```

Example response content:

```
{
    "move_result": [
        {"type": "Move", "value": "e2e4"},
        null,
        23
    ]
}
```

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Request JSON Object

- **requested_move** (*object*) – The move you want to perform.

Response JSON Object

- **move_result** (*object*) – The result of your move, a list containing the requested_move, the taken_move, and the capture square if one occurred.
- **type** (*string*) – `Move`.
- **value** (*Optional<string>*) – The move encoded as a UCI string. `null` if no piece is there.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Game is finished, you haven't sensed, you already moved, or malformed request data.
- 401 Unauthorized – Invalid or empty authentication information, or not a player in the specified game.
- 404 Not Found – Game does not exist.

POST /api/games/(int: *game_id*)/end_turn

End your turn. See *reconchess.Game.end_turn()*.

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Game is finished, you haven't sensed and moved.
- 401 Unauthorized – Invalid or empty authentication information, or not a player in the specified game.
- 404 Not Found – Game does not exist.

GET /api/games/(int: *game_id*)/is_over

Whether the game is over. See *reconchess.Game.is_over()*.

We recommend using the *game_status* endpoint for turn management.

Example response content:

```
{
    "is_over": true
}
```

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **is_over** (*boolean*) – Whether the game is over.

Status Codes

- 200 OK – Success.
- 401 Unauthorized – Invalid or empty authentication information.
- 404 Not Found – Game does not exist.

POST /api/games/(int: *game_id*)/resign

Resign from the game. Can only be called during your turn.

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Status Codes

- 200 OK – Success.
- 400 Bad Request – It is not your turn.
- 401 Unauthorized – Invalid or empty authentication information.
- 404 Not Found – Game does not exist.

POST /api/games/(int: *game_id*)/error_resign

Tell the server that you have errored out. This just zeros out any time you have remaining instead of waiting for the time to run out.

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Status Codes

- 200 OK – Success.
- 401 Unauthorized – Invalid or empty authentication information.
- 404 Not Found – Game does not exist.

GET /api/games/(int: *game_id*)/is_my_turn

Whether it is your turn to play.

We recommend using the *game_status* endpoint for turn management.

Example response content:

```
{
    "is_my_turn": true
}
```

Parameters

- **`game_id`** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **`is_my_turn`** (*`boolean`*) – Whether it is your turn to play.

Status Codes

- 200 OK – Success.
- 401 Unauthorized – Invalid or empty authentication information.
- 404 Not Found – Game does not exist.

`GET /api/games/(int: game_id)/game_status`

A combination of the *is_over* and *is_my_turn* endpoints.

Example response content:

```
{
    "is_my_turn": true,
    "is_over": false
}
```

Parameters

- **`game_id`** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **`is_my_turn`** (*`boolean`*) – Whether it is your turn to play.
- **`is_over`** (*`boolean`*) – Whether the game is over.

Status Codes

- 200 OK – Success.
- 401 Unauthorized – Invalid or empty authentication information.
- 404 Not Found – Game does not exist.

`GET /api/games/(int: game_id)/winner_color`

The color of the winner of the game. See *`reconchess.Game.get_winner_color()`*.

Example response content:

```
{
    "winner_color": true
}
```

Parameters

- **`game_id`** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **winner_color** (*Optional<boolean>*) – The color of the player that one the game - `true` for White, `false` for Black, and `null` for a draw.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Game is not over.
- 401 Unauthorized – Invalid or empty authentication information.
- 404 Not Found – Game does not exist.

GET /api/games/(int: *game_id*)/win_reason

The reason the game ended. See *reconchess.Game.get_win_reason()* and *reconchess.WinReason*.

Example response content:

```
{
    "win_reason": {
        "type": "WinReason",
        "value": "KING_CAPTURE"
    }
}
```

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **win_reason** (*Optional<WinReason>*) – The reason the game ended.
- **type** (*string*) – `WinReason`.
- **value** (*string*) – The string version of the values of *reconchess.WinReason*. `null` if a draw.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Game is not over.
- 401 Unauthorized – Invalid or empty authentication information.
- 404 Not Found – Game does not exist.

GET /api/games/(int: *game_id*)/game_history

The history of the game. See *reconchess.Game.get_game_history()* and *reconchess.GameHistory*.

Example response content:

```
{
    "game_history": {
        "type": "GameHistory",
        "senses": {
            "true": [55],
            "false": [null]
        },
        "sense_results": {
            "true": [
                [
                    [54, {"type": "Piece", "value": "p"}],
                    [55, null],
                    [56, {"type": "Piece", "value": "K"}]
                ]
            ],
            "false": [[]]
        },
        "requested_moves": {
            "true": [{"type": "Move", "value": "e2e4"}],
            "false": [{"type": "Move", "value": "e7e8"}]
        },
        "taken_moves": {
            "true": [{"type": "Move", "value": "e2e4"}],
            "false": [null]
        },
        "capture_squares": {
            "true": [23],
            "false": [null]
        },
        "fens_before_move": {
            "true": ["rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"],
            "false": ["rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"]
        },
        "fens_after_move": {
            "true": ["rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"],
            "false": ["rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"]
        }
    }
}
```

Parameters

- **game_id** – The ID of the game.

Request Headers

- Authorization – Basic Authorization.

Response JSON Object

- **game_history** (*object*) – The game history object.
- **type** (*string*) – The type of the object.
- **value** (*string*) – The value of the object.
- **senses** (*object*) – An object containing the senses for each player.
- **sense_results** (*object*) – An object containing the sense_results for each player.

- **`requested_moves`** (*object*) – An object containing the requested_moves for each player.
- **`taken_moves`** (*object*) – An object containing the taken_moves for each player.
- **`capture_squares`** (*object*) – An object containing the capture_squares for each player.
- **`fens_before_move`** (*object*) – An object containing the fens_before_move for each player.
- **`fens_after_move`** (*object*) – An object containing the fens_after_move for each player.

Status Codes

- 200 OK – Success.
- 400 Bad Request – Game is not over.
- 401 Unauthorized – Invalid or empty authentication information.
- 404 Not Found – Game does not exist.

CHAPTER 6

Indices and tables

- genindex
- search

HTTP Routing Table

/api

Index