`basebot` Programmer's Manual

2019-04-12

Contents

Abstract

This document provides an overview over the entry points into writing bots using `basebot`.

Introduction

`basebot` supports two approaches to writing bots, a procedural and an object-oriented one. Both are equivalently powerful; however, although simple bots are written quickly using the procedural approach, implementing more complex functionality using it can become ugly as quickly.

For actually running a bot (be it in production or testing), see the corresponding section.

Because the procedural approach builds upon the object-oriented, the latter is explained first; you can skip to the procedural one if you are not interested.

Object-oriented approach

The main (and historically only) way to create new bots is to inherit from the `basebot.Bot` class. Subclasses may override some class attributes to provide normally static values:

- `BOTNAME`: The "codename" of the bot as used for logging.
- `NICKNAME`: The nickname to set when entering a room. If at the default of `None`, the bot will not set a nickname at all; some functionality will be unavailable.
- `SHORT_HELP`: If not `None`, this is replied with to the bare `!help` command, and if no `LONG_HELP` is set, it is used for the specific `!help` command. See the botrulez for advice on how to format help messages.
- `LONG_HELP`: If `None`, the specific `!help @BotName` command is not replied to. If at the default value of `Ellipsis` *(yes, that is a real singleton)*, the value of `SHORT_HELP` is used instead. Otherwise, this is replied with to a specific `!help` command.

The constructor of the subclass should pass all keyword arguments (and all positional arguments, as applicable) on to the parent class constructor. Some of those may be used to tune responses to standard commands; refer to the reference for details.

There is a plethora of handler methods subclasses can override; a few notable ones are listed here. In every case of overriding a method, the corresponding method of the parent class must be invoked, or undefined behavior occurs.

`handle_chat` — Live message processing

```
handle_chat(msg : Message, meta : dict) -> None
```

This handler is invoked on "live" chat messages (in the library's parlance), *i.e.* `send-event`-s, which correspond to users (or bots) posting new messages.

- `msg` is a `Message` structure, presented as an instance of the `basebot.Record` class that is a dictionary exposing some items as attributes. The most interesting parts of it (but not all; refer to the reference for details) are found at:

  - `msg.id`: The ID of the message.
  - `msg.parent`: The ID of the parent of the message, or `None`.
  - `msg.sender.id`: The user ID of the sender of the message. This may be an `account:` ID for a user logged into an account, or an `agent:` or `bot:` ID otherwise.
  - `msg.sender.name`: The nickname of the sender of the message.
  - `msg.content`: The content of the message.

- `meta` is an ordinary dictionary holding miscellaneous meta-information about the message; most notable is:

  - `reply`: A convenience function with the following signature:

    ```
    reply(content : str, callback : callable = None) -> int
    ```

    This posts a message with a content of `content` as a reply to the message currently being handled (the concrete ID for each instance is stored in the closure and does not change when another message is handled). `callback`, if specified, is called with the server's `send-reply` to "our" reply as the only argument when the server accepts the message; see `send_chat()` for details. The return value is the sequence ID of the `send` packet submitted.

  Further members are omitted here; see the reference for a full listing.

The return value of `handle_chat` is ignored.

`handle_command` — Command handling

```
handle_command(cmdline : list, meta : dict) -> None
```

This handler is invoked when a "live" chat message (as elaborated above) is in addition a bot command, *i.e.*, the first non-whitespace character is an exclamation mark `!`; the method is run after `handle_chat`.

- `cmdline` is a list of `basebot.Token`-s, *i.e.* strings with an additional `offset` attribute unambiguously identifying the position in the "parent" string (see below for how to reach that; the `Token` class does not implement anything beyond; all operations return bare strings). According to the botrulez, the very first item of `cmdline` is the command name (including the leading `!`); further items are the arguments in their original order.

- `meta` is again an ordinary dictionary holding references to some objects of interest. Particularly interesting may be:

  - `line`: The entire unfiltered command line as a string.
  - `msgid`: The ID of the message.
  - `sender`: The nickname of the author of the message.
  - `sender_id`: The (agent) ID of the sender.
  - `reply`, a convenience function for replying elaborated upon above.

The return value of `handle_command` is, again, ignored.

Additional handlers

- `handle_login() -> None` — *Initial actions*

  This method is invoked after the bot has successfully authenticated in a room, but has not set a nickname yet. The return value is ignored.

- `handle_nick_set() -> None` — *Late initial actions*

  This method is invoked after the bot has set its nick; it can be used to post messages announcing the bot's appearance. The return value is ignored again.

- `handle_logout(ok : bool, final : bool) -> None` — *Early final actions*

  This method is the inverse of `handle_login`; it is invoked just before the bot disconnects. The return value is ignored.

  `ok` tells whether the connection is being terminated normally (`True`) or was severed abruptly (`False`); if it is true, the bot may post a final message. `final` tells whether the log-out is a temporary disconnect (`False`) or the bot shutting down terminally (`True`).

`send_chat` — Post a message

```
send_chat(content : str, parent : str = None) -> int
```

This method — which is *not* a handler (but may be overridden anyway) — posts a chat message. `content` is the text of the message, `parent` is either the ID of the parent of the tentative message, or `None` for starting a new thread. The function returns the sequence ID (`id` in the packet descritpion) of the `send` submitted.

An additional keyword-only argument `_callback` may be passed; it is a function with the following signature:

```
callback(packet : Packet) -> None
```

The callback is invoked with the server's `send-reply` to "our" `send` (when it arrives) as the only argument. The return value is ignored.

Procedural approach

The other bot writing approach `basebot` supports is procedural, and avoids the use of own classes altogether. Instead, (named) arguments are passed to the `basebot.run_minibot` function (or, alternatively, to the constructor of `basebot.MiniBot`, which inherits from `basebot.Bot`) for setting all configuration values pointed out above.

- `botname`: The name of the bot for logging; corresponds to the `BOTNAME` class attribute.
- `nickname`: The default nickname of the bot; corresponds to `NICKNAME`.

- `short_help`: The reply for the general `!help` command; corresponds to `SHORT_HELP`.
- `long_help`: The reply for the specific `!help @BotName` command; if not specified, `short_help` is used instead (if given). Corresponds to `LONG_HELP`.

The following argument may be used to implement the functionality of a bot (defined indeed by parents of the `Bot` class):

- `command_handlers`: A mapping from command names (*without* the leading exclamation marks `!`; the `None` singleton matches any command) to functions (or lists of functions, but not strings), which are invoked when the command identified by the key is encountered; the signature of the handler functions is the same as of `handle_command` above.

The following functionality is only present at `MiniBot`:

- `regexes`: A mapping from regular expression strings to reply specifications. The regular expressions are matched against the message (using `re.search()`) in traversal order, *i.e.*, in general, **in undefined order**, and the value corresponding to the first regular expression that matched is used.
- `match_self`: If the argument is absent or false (the default), messages from the bot itself are *not* interpreted by it to avoid infinite loops; if true, they are.
- `match_all`: If the argument is absent or false (the default), the behavior described above under `regexes` is implemented; if true, *all* regular expressions are matched against the input in traversal order.

If a particular ordering of regular expression triggers is desired, a `collections.OrderedDict` instance may be passed as `regexes`; the warning about undefined order becomes void in that case.

Reply specifications

The following steps are applied to an object when the regular expression corresponding to it has matched:

1. If the object is callable, it is called as follows:

   ```
   handler(match : re.match, meta : dict) -> object
   ```

   The first argument is the match object gained from matching the corresponding regular expression against the message; the second one is a dictionary containing meta-information that may be of interest (the items described here are indeed identical to the ones mentioned under `handle_command`):

   - `msgid`: The ID of the message being processed.
   - `sender`: The nickname of the sender of the message.

- `sender_id`: The (agent) ID of the sender of the message.
- `reply`: A function that, when called, posts a message as a reply to the current message. As an additional argument, a callback may be passed that is called with the server's `send-reply` to "our" reply as the argument. See the description of `reply` in the section above for more details.

Additional members can be found in the reference.

The return value of the function, or the object itself if it is not one, is used in the next step.

2. If the object is a list or tuple, all of its entries are considered by the next step individually; if it is a string, only it itself is; if it is `None`, the next step is not performed.
3. Unless the original object (in the first step) had been a function, the current string is passed through the match object's `expand` method (see the Python documentation), and then replied with to the message being processed.

Hence, the following patterns for handling a message can be highlighted:

- A (mostly) fixed reply (or list thereof) is hard-coded and expanded with groupings from the match.
- A handler function (or lambda) processes the match and returns a string (or list of strings) to reply with.
- A handler function processes the match and returns nothing (but has side effects, or stores the `reply` closure for later use).

Shared state

Apart from storing state globally (which is frowned upon), the callbacks (in fact all the ones mentioned above which have a `meta` argument) can access the `MiniBot` (or `Bot`) instance used as the `meta['self']` member. In addition, the following callbacks may be specified for the bot to handle state initialization / cleanup:

- `init_cb` is a function invoked once the bot connects for the first time. It takes a single positional argument, namely the bot instance going to run.
- `close_cb` is similarly invoked at the very end of the bot's main loop with it as the only argument.

Running bots

`basebot` provides automated means of setting up a bot (along with other facilities such as logging) with two module-level functions:

- `basebot.run_bot` is called with the bot class as the only positional argument (and optional configuration via keyword arguments), and spawns the bot defined by the given class in the rooms specified on the command line.

- `basebot.run_minibot` takes no positional arguments at all, but keyword arguments only. It is identical from `run_bot`, except that it substitutes `MiniBot` as the bot class (if none is given explicitly). See above for some arguments of note.

Advanced

This section covers topics not immediately needed for writing a basic bot.

Bot managers

The `basebot.BotManager` class is responsible for parsing the command line, creating bot instances, in potentially multiple rooms, and overseeing their execution. Functionality available across room boundaries should be bound here.

The manager class to use can be specified using the `mgrcls` keyword argument to `run_bot` or `run_minibot`; its (class) methods are invoked by the latters for the principial actions. Of note are the following:

- `prepare_parser(parser : argparse.ArgumentParser, config : dict) -> None` — *Argument parser initialization*

  This method declares command-line options; refer to the Python documentation for details (see in particular the `add_argument` method). `config` is the dictionary of keyword arguments as passed to `run_bot` or `run_minibot`.

- `interpret_args(arguments : object, config : dict) -> (bots : list, config : dict)` — *Argument processing*

  This method receives the argument values and the `config` object mentioned above as arguments and returns a list of bot specifications and a keyword argument dictionary for the `BotManager` constructor. Overriding classes are advised to invoke the parent class' method and to amend the `config` returned by it with values from the `arguments` object, transforming as necessary; these get passed down to the `BotManager` constructor and to the bots' constructors.

The constructor of a subclass should pass all positional and keyword arguments on to the parent class constructor and perform initialization as necessary.

Individual bot instances can access the `BotManager` that created them via the `manager` attribute, and other bot instances via it (although there is no standard way for that).

See also the subsection about thread safety just below.

Thread safety

Each bot instance is run in an own thread; as long as there is only one or the bot instances do not interact with each other, no particular precautions do need to be taken.

While the handler methods of bots are protected by the bot's state manipulation lock, arbitrary accesses from outside are not. Implement inter-bot interaction in a threadsafe way.

`basebot` provides convenience functions for spawning daemonic and non-daemonic worker threads at module level; be careful not to spawn them in methods that may be run multiple times over the lifetime of a bot (unless that is intended).

Further reading

The inline documentation of basebot.py provides a thorough reference of all components included.