rivescript Documentation

Release 1.14.9

Noah Petherbridge <root@kirsle.net>

Sep 21, 2017

Contents

Contents:

CHAPTER 1

rivescript package

Submodules

rivescript.rivescript module

class `rivescript.rivescript.`**`RiveScript`**(*debug=False*, *strict=True*, *depth=50*, *log=None*, *utf8=False*, *session_manager=None*)

Bases: `object`

A RiveScript interpreter for Python 2 and 3.

Parameters

- **`debug`** (*`bool`*) – Set to `True` to enable verbose logging to standard out.
- **`strict`** (*`bool`*) – Enable strict mode. Strict mode causes RiveScript syntax errors to raise an exception at parse time. Strict mode is on (`True`) by default.
- **`log`** (*`str or fh`*) – Specify a path to a file or a filehandle opened in write mode to direct log output to. This can send debug logging to a file rather than to `STDOUT`.
- **`depth`** (*`int`*) – Set the recursion depth limit. This is how many times RiveScript will recursively follow redirects before giving up with a `DeepRecursionError` exception. The default is `50`.
- **`utf8`** (*`bool`*) – Enable UTF-8 mode. When this mode is enabled, triggers in RiveScript code are permitted to contain foreign and special symbols. Additionally, user messages are allowed to contain most symbols instead of having all symbols stripped away. This is considered an experimental feature because all of the edge cases of supporting Unicode haven't been fully tested. This option is `False` by default.
- **`session_manager`** (`SessionManager`) – By default RiveScript uses an in-memory session manager to keep track of user variables and state information. If you have your own session manager that you'd like to use instead, pass its instantiated class instance as this parameter.

classmethod VERSION()
Return the version number of the RiveScript library.

This may be called as either a class method or a method of a RiveScript object instance.

clear_uservars(*user=None*)
Delete all variables about a user (or all users).

Parameters **user** (*str*) – The user ID to clear variables for, or else clear all variables for all users if not provided.

current_user()
Retrieve the user ID of the current user talking to your bot.

This is mostly useful inside of a Python object macro to get the user ID of the person who caused the object macro to be invoked (i.e. to set a variable for that user from within the object).

This will return `None` if used outside of the context of getting a reply (the value is unset at the end of the `reply()` method).

deparse()
Dump the in-memory RiveScript brain as a Python data structure.

This would be useful, for example, to develop a user interface for editing RiveScript replies without having to edit the RiveScript source code directly.

Return dict JSON-serializable Python data structure containing the contents of all RiveScript replies currently loaded in memory.

freeze_uservars(*user*)
Freeze the variable state for a user.

This will clone and preserve a user's entire variable state, so that it can be restored later with `thaw_uservars()`.

Parameters **user** (*str*) – The user ID to freeze variables for.

get_global(*name*)
Retrieve the current value of a global variable.

Parameters **name** (*str*) – The name of the variable to get.

Return str The value of the variable or `"undefined"`.

get_uservar(*user*, *name*)
Get a variable about a user.

Parameters

- **user** (*str*) – The user ID to look up a variable for.
- **name** (*str*) – The name of the variable to get.

Returns

The user variable, or `None` or `"undefined"`:

- If the user has no data at all, this returns `None`.
- If the user doesn't have this variable set, this returns the string `"undefined"`.
- Otherwise this returns the string value of the variable.

get_uservars(*user=None*)
Get all variables about a user (or all users).

Parameters **str user** (*optional*) – The user ID to retrieve all variables for. If not passed, this function will return all data for all users.

Return dict All the user variables.

- If a `user` was passed, this is a `dict` of key/value pairs of that user's variables. If the user doesn't exist in memory, this returns `None`.
- Otherwise, this returns a `dict` of key/value pairs that map user IDs to their variables (a `dict` of `dict`).

get_variable(*name*)
Retrieve the current value of a bot variable.

Parameters **name** (*str*) – The name of the variable to get.

Return str The value of the variable or `"undefined"`.

last_match(*user*)
Get the last trigger matched for the user.

Parameters **user** (*str*) – The user ID to get the last matched trigger for.

Return str The raw trigger text (tags and all) of the trigger that the user most recently matched. If there was no match to their last message, this returns `None` instead.

load_directory(*directory*, *ext=None*)
Load RiveScript documents from a directory.

Parameters

- **directory** (*str*) – The directory of RiveScript documents to load replies from.
- **ext** (*[]str*) – List of file extensions to consider as RiveScript documents. The default is `[".rive", ".rs"]`.

load_file(*filename*)
Load and parse a RiveScript document.

Parameters **filename** (*str*) – The path to a RiveScript file.

reply(*user*, *msg*, *errors_as_replies=True*)
Fetch a reply from the RiveScript brain.

Parameters

- **user** (*str*) – A unique user ID for the person requesting a reply. This could be e.g. a screen name or nickname. It's used internally to store user variables (including topic and history), so if your bot has multiple users each one should have a unique ID.
- **msg** (*str*) – The user's message. This is allowed to contain punctuation and such, but any extraneous data such as HTML tags should be removed in advance.
- **errors_as_replies** (*bool*) – When errors are encountered (such as a deep recursion error, no reply matched, etc.) this will make the reply be a text representation of the error message. If you set this to `False`, errors will instead raise an exception, such as a `DeepRecursionError` or `NoReplyError`. By default, no exceptions are raised and errors are set in the reply instead.

Returns The reply output.

Return type str

set_global(*name*, *value*)
Set a global variable.

Equivalent to `! global` in RiveScript code.

Parameters

- **name** (*str*) – The name of the variable to set.
- **value** (*str*) – The value of the variable. Set this to `None` to delete the variable.

set_handler(*language*, *obj*)
Define a custom language handler for RiveScript objects.

Pass in a `None` value for the object to delete an existing handler (for example, to prevent Python code from being able to be run by default).

Look in the `eg` folder of the rivescript-python distribution for an example script that sets up a JavaScript language handler.

Parameters

- **language** (*str*) – The lowercased name of the programming language. Examples: python, javascript, perl
- **obj** (*class*) – An instance of an implementation class object. It should provide the following interface:

```
class MyObjectHandler:
    def __init__(self):
        pass
    def load(self, name, code):
        # name = the name of the object from the RiveScript code
        # code = the source code of the object
    def call(self, rs, name, fields):
        # rs     = the current RiveScript interpreter object
        # name   = the name of the object being called
        # fields = array of arguments passed to the object
        return reply
```

set_person(*what*, *rep*)
Set a person substitution.

Equivalent to `! person` in RiveScript code.

Parameters

- **what** (*str*) – The original text to replace.
- **rep** (*str*) – The text to replace it with. Set this to `None` to delete the substitution.

set_subroutine(*name*, *code*)
Define a Python object from your program.

This is equivalent to having an object defined in the RiveScript code, except your Python code is defining it instead.

Parameters

- **name** (*str*) – The name of the object macro.
- **code** (*def*) – A Python function with a method signature of `(rs, args)`

This method is only available if there is a Python handler set up (which there is by default, unless you've called `set_handler("python", None)`).

set_substitution(*what*, *rep*)
Set a substitution.

Equivalent to `! sub` in RiveScript code.

Parameters

- **what** (*str*) – The original text to replace.
- **rep** (*str*) – The text to replace it with. Set this to `None` to delete the substitution.

set_uservar(*user*, *name*, *value*)
Set a variable for a user.

This is like the `<set>` tag in RiveScript code.

Parameters

- **user** (*str*) – The user ID to set a variable for.
- **name** (*str*) – The name of the variable to set.
- **value** (*str*) – The value to set there.

set_uservars(*user*, *data=None*)
Set many variables for a user, or set many variables for many users.

This function can be called in two ways:

```
# Set a dict of variables for a single user.
rs.set_uservars(username, vars)

# Set a nested dict of variables for many users.
rs.set_uservars(many_vars)
```

In the first syntax, `vars` is a simple dict of key/value string pairs. In the second syntax, `many_vars` is a structure like this:

```
{
    "username1": {
        "key": "value",
    },
    "username2": {
        "key": "value",
    },
}
```

This way you can export *all* user variables via `get_uservars()` and then re-import them all at once, instead of setting them once per user.

Parameters

- **str user** (*optional*) – The user ID to set many variables for. Skip this parameter to set many variables for many users instead.
- **data** (*dict*) – The dictionary of key/value pairs for user variables, or else a dict of dicts mapping usernames to key/value pairs.

This may raise a `TypeError` exception if you pass it invalid data types. Note that only the standard `dict` type is accepted, but not variants like `OrderedDict`, so if you have a dict-like type you should cast it to `dict` first.

set_variable(*name*, *value*)
Set a bot variable.

Equivalent to `! var` in RiveScript code.

Parameters

- **name** (*str*) – The name of the variable to set.
- **value** (*str*) – The value of the variable. Set this to `None` to delete the variable.

sort_replies(*thats=False*)
Sort the loaded triggers in memory.

After you have finished loading your RiveScript code, call this method to populate the various internal sort buffers. This is absolutely necessary for reply matching to work efficiently!

stream(*code*)
Stream in RiveScript source code dynamically.

Parameters **code** – Either a string containing RiveScript code or an array of lines of RiveScript code.

thaw_uservars(*user*, *action=u'thaw'*)
Thaw a user's frozen variables.

Parameters **action** (*str*) – The action to perform when thawing the variables:

- `discard`: Don't restore the user's variables, just delete the frozen copy.
- `keep`: Keep the frozen copy after restoring the variables.
- `thaw`: Restore the variables, then delete the frozen copy (this is the default).

trigger_info(*trigger=None*, *dump=False*)
Get information about a trigger.

Pass in a raw trigger to find out what file name and line number it appeared at. This is useful for e.g. tracking down the location of the trigger last matched by the user via `last_match()`. Returns a list of matching triggers, containing their topics, filenames and line numbers. Returns `None` if there weren't any matches found.

The keys in the trigger info is as follows:

•`category`: Either 'topic' (for normal) or 'thats' (for %Previous triggers)

•`topic`: The topic name

•`trigger`: The raw trigger text

•`filename`: The filename the trigger was found in.

•`lineno`: The line number the trigger was found on.

Pass in a true value for `dump`, and the entire syntax tracking tree is returned.

Parameters

- **trigger** (*str*) – The raw trigger text to look up.
- **dump** (*bool*) – Whether to dump the entire syntax tracking tree.

Returns A list of matching triggers or `None` if no matches.

write(*fh*, *deparsed=None*)
Write the currently parsed RiveScript data into a file.

Pass either a file name (string) or a file handle object.

This uses `deparse()` to dump a representation of the loaded data and writes it to the destination file. If you provide your own data as the `deparsed` argument, it will use that data instead of calling `deparse()`

itself. This way you can use `deparse()`, edit the data, and use that to write the RiveScript document (for example, to be used by a user interface for editing RiveScript without writing the code directly).

Parameters

- **fh** (*str or file*) – a string or a file-like object.
- **deparsed** (*dict*) – a data structure in the same format as what `deparse()` returns. If not passed, this value will come from the current in-memory data from `deparse()`.

rivescript.sessions module

class rivescript.sessions.**MemorySessionStorage**(*warn=None*, **args*, ***kwargs*)
Bases: *rivescript.sessions.SessionManager*

The default in-memory session store for RiveScript.

This session manager keeps all user and state information in system memory and doesn't persist anything to disk by default. This is suitable for many simple use cases. User variables can be persisted and reloaded from disk by using the RiveScript API functions `get_uservars()` and `set_uservars()` – for example, you can get export all user variables and save them to disk as a JSON file when your program shuts down, and on its next startup, read the JSON file from disk and use `set_uservars()` to put them back into the in-memory session manager.

If you'd like to implement your own session manager, for example to use a database to store/retrieve user variables, you should extend the base `SessionManager` class and implement all of its functions.

Parameters **warn** (*function*) – A function to be called with an error message to notify when one of the functions fails due to a user not existing. If not provided, then no warnings will be emitted from this module.

class rivescript.sessions.**NullSessionStorage**
Bases: *rivescript.sessions.SessionManager*

The null session manager doesn't store any user variables.

This is used by the unit tests and isn't practical for real world usage, as the bot would be completely unable to remember any user variables or history.

class rivescript.sessions.**SessionManager**
Bases: `object`

Base class for session management for RiveScript.

The session manager keeps track of getting and setting user variables, for example when the `<set>` or `<get>` tags are used in RiveScript or when the API functions like `set_uservar()` are called.

By default RiveScript stores user sessions in memory and provides methods to export and import them (e.g. to persist them when the bot shuts down so they can be reloaded). If you'd prefer a more 'active' session storage, for example one that puts user variables into a database or cache, you can create your own session manager that extends this class and implements its functions.

See the `eg/sessions` example from the source of rivescript-python at https://github.com/aichaos/rivescript-python for an example.

The constructor takes no required parameters. You can feel free to define `__init__()` however you need to.

default_session()
The default session data for a new user.

You do not need to override this function. This returns a `dict` with the default key/value pairs for new sessions. By default, the session variables are as follows:

```
{
    "topic": "random"
}
```

Returns A dict of default key/value pairs for new user sessions.

Return type dict

freeze(*username*)
Make a snapshot of the user's variables.

This should clone and store a snapshot of all stored variables for the user, so that they can later be restored with `thaw()`. This implements the RiveScript `freeze_uservars()` method.

Parameters **username** (*str*) – The username to freeze variables for.

get(*username*, *key*)
Retrieve a stored variable for a user.

If the user doesn't exist, this should return `None`. If the user *does* exist, but the key does not, this should return the string value `"undefined"`.

Parameters

- **username** (*str*) – The username to retrieve variables for.
- **key** (*str*) – The specific variable name to retrieve.

Returns The value of the requested key, "undefined", or `NoneType`.

Return type str

get_all()
Retrieve all variables about all users.

This should return a dict of dicts, where the top level keys are the usernames of every user your bot has data for, and the values are dicts of key/value pairs of those users. For example:

```
{ "user1": {
    "topic": "random",
    "name": "Alice",
    },
  "user2": {
    "topic": "random",
    "name": "Bob",
    },
}
```

Returns dict

get_any(*username*)
Retrieve all stored variables for a user.

If the user doesn't exist, this should return `None`.

Parameters **username** (*str*) – The username to retrieve variables for.

Returns Key/value pairs of all stored data for the user, or `NoneType`.

Return type dict

reset(*username*)
Reset all variables stored about a particular user.

Parameters **username** (*str*) – The username to flush all data for.

reset_all()
Reset all variables for all users.

set(*username*, *args*)
Set variables for a user.

Parameters

- **username** (*str*) – The username to set variables for.
- **args** (*dict*) – Key/value pairs of variables to set for the user. The values are usually strings, but they can be other types as well (e.g. arrays or other dicts) for some internal data structures such as input/reply history. A value of `NoneType` should indicate that the key should be deleted from the session store.

thaw(*username*, *action=u'thaw'*)
Restore the frozen snapshot of variables for a user.

This should replace *all* of a user's variables with the frozen copy that was snapshotted with `freeze()`. If there are no frozen variables, this function should be a no-op (maybe issue a warning?)

Parameters

- **username** (*str*) – The username to restore variables for.
- **action** (*str*) – An action to perform on the variables. Valid options are:
  - `thaw`: Restore the variables and delete the frozen copy (default).
  - `discard`: Don't restore the variables, just delete the frozen copy.
  - `keep`: Restore the variables and keep the copy still.

rivescript.exceptions module

exception `rivescript.exceptions.`**DeepRecursionError**
Bases: *rivescript.exceptions.RiveScriptError*

A deep recursion condition was detected and a reply can't be given.

This error can occur when you have triggers that redirect to each other in a circle, for example:

```
+ one
@ two

+ two
@ one
```

By default, RiveScript will only recursively look for a trigger up to 50 levels deep before giving up. This should be a large enough window for most use cases, but if you need to increase this limit you can do so by setting a higher value for the `depth` parameter to the constructor or changing it in your RiveScript source code, for example:

```
! global depth = 75
```

The text version is `[ERR: Deep recursion detected]`

exception `rivescript.exceptions.`**`NoDefaultRandomTopicError`**
Bases: `exceptions.Exception`

No default topic could be found.

This is a critical error and usually means no replies were loaded into the bot. Very unlikely is it the case that all replies belong to other topics than the default (`random`).

exception `rivescript.exceptions.`**`NoMatchError`**
Bases: *rivescript.exceptions.RiveScriptError*

No reply could be matched.

This means that no trigger was a match for the user's message. To avoid this error, add a trigger that only consists of a wildcard:

```
+ *
- I do not know how to reply to that.
```

The lone-wildcard trigger acts as a catch-all fallback trigger and will ensure that every message the user could send will match at least one trigger.

The text version is `[ERR: No reply matched]`

exception `rivescript.exceptions.`**`NoReplyError`**
Bases: *rivescript.exceptions.RiveScriptError*

No reply could be found.

This means that the user's message matched a trigger, but the trigger didn't yield any response for the user. For example, if a trigger was followed only by `*Conditions` and none of them were true and there were no normal replies to fall back on, this error can come up.

To avoid this error, always make sure you have at least one `-Reply` for every trigger.

The text version is `[ERR: No reply found]`.

exception `rivescript.exceptions.`**`ObjectError`**(*error_message=u'[ERR: Error when executing Python object]'*)
Bases: *rivescript.exceptions.RiveScriptError*

An error occurred when executing a Python object macro.

This will usually be some kind of run-time error, like a `ZeroDivisionError` or `IndexError` for example.

The text version is `[ERR: Error when executing Python object]`

exception `rivescript.exceptions.`**`RepliesNotSortedError`**
Bases: `exceptions.Exception`

You attempted to get a reply before sorting the triggers.

You should call `sort_replies()` after loading your RiveScript code and before calling `reply()` to look up a reply.

exception `rivescript.exceptions.`**`RiveScriptError`**(*error_message=None*)
Bases: `exceptions.Exception`

RiveScript base exception class.

rivescript.interactive module

rivescript.interactive.**interactive_mode**()

The built-in RiveScript Interactive Mode.

This feature of RiveScript allows you to test and debug a chatbot in your terminal window. There are two ways to invoke this mode:

```
# By running the Python RiveScript module directly:
python rivescript eg/brain

# By running the shell.py in the source distribution:
python shell.py eg/brain
```

The only required command line parameter is a filesystem path to a directory containing RiveScript source files (with the `*.rive` file extension).

Additionally, it accepts command line flags.

Parameters

- **--utf8** – Enable UTF-8 mode.
- **--json** – Use JSON to communicate with the bot instead of plain text. See the JSON Mode documentation below for advanced details.
- **--debug** – Enable verbose debug logging.
- **--log** (*str*) – The path to a text file you want the debug logging to be written to. This is to be used in conjunction with `--debug`, for the case where you don't want your terminal window to be flooded with debug messages.
- **--depth** (*int*) – Override the recursion depth limit (default `50`).
- **--nostrict** – Disable strict syntax checking when parsing the RiveScript files. By default a syntax error raises an exception and will terminate the interactive mode.
- **--help** – Show the documentation of command line flags.
- **path** (*str*) – The path to a directory containing `.rive` files.

JSON Mode

By invoking the interactive mode with the `--json` (or `-j`) flag, the interactive mode will communicate with you via JSON messages. This can be used as a "bridge" to enable the use of RiveScript from another programming language that doesn't have its own native RiveScript implementation.

For example, a program could open a shell pipe to the RiveScript interactive mode and send/receive JSON payloads to communicate with the bot.

In JSON mode, you send a message to the bot in the following format:

```
{
    "username": "str username",
    "message": "str message",
    "vars": {
        "topic": "random",
        "name": "Alice"
    }
}
```

The `username` and `message` keys are required, and `vars` is a key/value object of all the variables about the user.

After sending the JSON payload over standard input, you can either close the input file handle (send the EOF signal; or Ctrl-D in a terminal), or send the string `__END__` on a line of text by itself. This will cause the bot to parse your payload, get a reply for the message, and respond with a similar JSON payload:

```
{
    "status": "ok",
    "reply": "str response",
    "vars": {
        "topic": "random",
        "name": "Alice"
    }
}
```

The `vars` structure in the response contains all of the key/value pairs the bot knows about the username you passed in. This will also contain a lot of internal data, such as the user's history and last matched trigger.

To keep a stateful session, you should parse the `vars` returned by RiveScript and pass them in with your next request so that the bot can remember them for the next reply.

If you closed the filehandle (Ctrl-D, EOF) after your input payload, the interactive mode will exit after giving the response. If, on the other hand, you sent the string `__END__` on a line by itself after your payload, the RiveScript interactive mode will do the same after its response is returned. This way, you can re-use the shell pipe to send and receive many messages over a single session.

rivescript.parser module

class `rivescript.parser.`**`Parser`**(*strict=True*, *utf8=False*, *on_debug=None*, *on_warn=None*)

Bases: `object`

The RiveScript language parser.

This module can be used as a stand-alone parser for third party developers to use, if you want to be able to simply parse (and syntax check!) RiveScript source code and get an "abstract syntax tree" back from it.

To that end, this module removed all dependencies on the parent RiveScript class. When the RiveScript module uses this module, it passes its own debug and warning functions as the `on_debug` and `on_warn` parameters, but these parameters are completely optional.

Parameters

- **`strict`** (*`bool`*) – Strict syntax checking (true by default).
- **`utf8`** (*`bool`*) – Enable UTF-8 mode (false by default).
- **`on_debug`** (*`func`*) – An optional function to send debug messages to. If not provided, you won't be able to get debug output from this module. The debug function's prototype is: `def f(message)`
- **`on_warn`** (*`func`*) – An optional function to send warning/error messages to. If not provided, you won't be able to get any warnings from this module. The warn function's prototype is `def f(message, filename='', lineno='')`

`check_syntax`(*cmd*, *line*)

Syntax check a line of RiveScript code.

Parameters

- **cmd** (*str*) – The command symbol for the line of code, such as one of +, −, *, >, etc.
- **line** (*str*) – The remainder of the line of code, such as the text of a trigger or reply.

Returns A string syntax error message or `None` if no errors.

Return type str

parse (*filename*, *code*)

Read and parse a RiveScript document.

Returns a data structure that represents all of the useful contents of the document, in this format:

```
{
    "begin": { # "begin" data
        "global": {}, # map of !global vars
        "var": {},    # bot !var's
        "sub": {},    # !sub substitutions
        "person": {}, # !person substitutions
        "array": {},  # !array lists
    },
    "topics": { # main reply data
        "random": { # (topic name)
            "includes": {}, # map of included topics (values=1)
            "inherits": {}, # map of inherited topics
            "triggers": [   # array of triggers
                {
                    "trigger": "hello bot",
                    "reply": [], # array of replies
                    "condition": [], # array of conditions
                    "redirect": None, # redirect command
                    "previous": None, # 'previous' reply
                },
                # ...
            ]
        }
    }
    "objects": [ # parsed object macros
        {
            "name": "",     # object name
            "language": "", # programming language
            "code": [],     # array of lines of code
        }
    ]
}
```

Parameters

- **filename** (*str*) – The name of the file that the code came from, for syntax error reporting purposes.
- **code** (*str[]*) – The source code to parse.

Returns The aforementioned data structure.

Return type dict

rivescript.python module

class rivescript.python.**PyRiveObjects**
Bases: object

A RiveScript object handler for Python code.

This class provides built-in support for your RiveScript documents to include and execute object macros written in Python. For example:

```
> object base64 python
    import base64 as b64
    return b64.b64encode(" ".join(args))
< object

+ encode * in base64
- OK: <call>base64 <star></call>
```

Python object macros receive these two parameters:

- rs (RiveScript): The reference to the parent RiveScript instance.
- args ([]str): A list of argument words passed to your object.

Python support is on by default. To turn it off, just unset the Python language handler on your RiveScript object:

```
rs.set_handler("python", None)
```

call(*rs*, *name*, *user*, *fields*)
Invoke a previously loaded object.

Parameters

- **rs** (RiveScript) – the parent RiveScript instance.
- **name** (*str*) – The name of the object macro to be called.
- **user** (*str*) – The user ID invoking the object macro.
- **fields** (*[]str*) – Array of words sent as the object's arguments.

Return str The output of the object macro.

load(*name*, *code*)
Prepare a Python code object given by the RiveScript interpreter.

Parameters

- **name** (*str*) – The name of the Python object macro.
- **code** (*[]str*) – The Python source code for the object macro.

exception rivescript.python.**PythonObjectError**
Bases: exceptions.Exception

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

T

V

W