Évariste Documentation

Release 1.2.1

Louis Paternault

May 06, 2024

CONTENTS

Given a git repository (or any directory), Évariste has two purposes:

- compile every file (*à la* make, with a different configuration);
- generate an HTML page presenting every file (both compiled file and source file), as an annotated directory tree.

For instance, Évariste turns a git repository of hundreds of LaTeX files into an HTML page with annotated source and compiled files.

Layout of this documentation

- Installation is explained in *the next section* (page 5).
- The basic concepts of Évariste are explained in *Getting started* (page 7).
- A more thorough user documentation is available in *User documentation* (page 7).
- Évariste is extensible. Learn about existing plugins, as well as how to write your own plugins, in *Plugins* (page 17).
- Évariste comes with a few helpers tools, which are described in *evs tools* (page 41).
- Developers might want to have a glance at *Library documentation* (page 43).

Enjoy!

CHAPTER

ONE

USE CASE

1.1 TL;DR

Évariste turns a git repository of hundreds of LaTeX files into an HTML page with annotated source and compiled files.

1.2 More details, please?

Louis is a math teacher. He has every course material in several git repositories (one per course). Let's take a look at this repository.

This repository contain tens or hundreds of LaTeX files (most of them being compiled with a single pass of LuaLaTeX, some of them require several passes, a few are compiled using LaTeX+dvipdf), a few LibreOffice documents, and probably a few other files.

Louis has two copies of this repository: one on his computer at home, and one on his USB key thas is carried at work, and that he uses to print documents on the work printer, and to display them using a beamer during his lessons. Louis uses git to synchronize those copies.

1.2.1 Purpose #1

At home, Louis has (almost) finished working on some material for his students. He commits the LaTeX files in his git repository, pushes them to some server, and, on his USB key (the one he carries at work):

- he pulls the changes (so that this key contains the latest version of the LaTeX *source* files);
- he runs Évariste (so that the new or recently modified LaTeX source files are compiled to PDF files that he can print or show to his students).

1.2.2 Purpose #2

Louis would be happy if other teachers reused his course material, so he publishes his repository on a public git repository. But this repository only contains source files (and some of Louis's colleagues have never heard about LaTeX), and navigating those files is not friendly. So, when Louis pushes his changes to this public repository, using continuous integration:

- Évariste compiles every single LateX file (at least, those which have changed);
- Évariste generates a HTML page which displays every single file of this repository, together with its compiled (PDF) version, and, optionnaly, some annotation.

CHAPTER

TWO

DOWNLOAD AND INSTALL

Évariste can be installed using pip:

```
python3 -m pip install evariste
```

You can build your own Debian (and Ubuntu?) package using stdeb:

```
python3 setup.py --command-packages=stdeb.command bdist_deb
sudo dpkg -i deb_dist/evariste-<VERSION>_all.deb
```

CHAPTER

THREE

USER DOCUMENTATION

3.1 Getting started

You have a directory that you want to be processed using Évariste.

3.1.1 Minimal configuration file

Create a `evariste.setup` file containing the following text:

```
[setup]
plugins = vcs.fs
```

Note that if your directory is a git repository, you can use `vcs.git` instead of `vcs.fs`. That way, only files handled by git will be processed (more information about *vcs plugins* (page 32) and *setup files* (page 10)).

That's it! You can now run `evariste` on this file:

```
evariste evariste.setup
```

And nothing happens... You need to give Évariste to pieces of information:

- how files are to be compiled;
- what should be the output.

3.1.2 Compile file

To actually compile files, you need to enable one or several *action plugins* (page 17) in the `enable_plugin` option of the configuration file (see first section).

Let's use the *action.command* (page 19) and *action.autocommand* (page 17) plugins. Our setup file now looks like this:

```
[setup]
plugins =
    vcs.git
    action.command action.autocommand

[action.autocommand.latex]
extensions = tex
targets = {basename}.pdf
command =
  latex {basename}
  dvipdf {basename}
```

The `action.command` plugin is not used yet. The `action.autocommand` is used, and the `action.autocommand.latex` means: Every file with extension `.tex` will be compiled (in its directory) using command `pdflatex {basename}` (where `{basename}` is replaced with the base name of the file, that is, without directory or extension; more info in *String formatting* (page 14)), and will produce a `{basename}.pdf` file.

Now, that particular `foo.tex` file must be compiled using `lualatex`. Let's use the `action.command` plugin, and write a small configuration file for it. This file can be named either `foo.tex.evsconfig` or `.foo.tex.evsconfig`, and contains:

```
[action]
plugin = command

[action.command]
targets = {basename}.pdf
command = lualatex {basename}
```

This means:

- for this file, and this file only, the `action.command` will be used;
- it will be compiled using the `lualatex foo` command.

Let's run `evariste` again, this time with the `--verbose` option:

```
evariste evariste.setup --verbose
```

You can see that your latex files are correctly compiled.

More information, as well as the list of action plugins, can be found in *Action plugins* (page 17).

3.1.3 Output

Right now, nothing is displayed at the end of the compilation. Let's improve thit.

Text renderer

Let's enable the *renderer_text plugin* (page 31). The `[setup]` section of your setup file now looks like this:

```
[setup]
plugins =
    vcs.git
    action.command action.autocommand
    renderer.text
```

And a tree is displayed at the end of the `evariste evariste.setup` call: it lists all the files that were compiled, with their status (success or failed compilation).

HTML renderer

Now you want to publish your directory as an HTML page like this one. To do so, we simply enable the *renderer_html plugin* (page 27).

```
[setup]
plugins =
    vcs.git
    action.command action.autocommand
    renderer.text renderer.html
```

Let's run `evariste evariste.setup` again, and *voilà!*, we get a `index.html` file listing the files of our repository as a tree, linking to both as source (latex) and compiled (pdf) files.

This plugin can be configured (*renderer.html — HTML renderer* (page 27)), but you might prefer the *HTMLplus renderer* (page 29), which add a bit of CSS and javascript to make the output nicer.

3.1.4 Conclusion

Évariste is very configurable. There is a lot more to discover: *more options* (page 10), *configure and ignore files* (page 14), several *action plugins* (page 17) or *renderer plugins* (page 26), or *more* (page 17).

Enjoy!

3.2 Usage

Here are the command line options for *evariste*. Note that:

- other tools are installed together *evariste*: *evs tools* (page 41);
- you might be interested in the *logging plugins* (page 24) to configure output.

Recursively compile files in a directory, and render result.

```
usage: evariste [-h] [--version] [-v] [-q] [-j JOBS] [-B] SETUP
```

3.2.1 Positional Arguments

SETUP Setup file to process.

3.2.2 Named Arguments

--version Show version

-v, --verbose Verbose. Repeat for more details.

-q, --quiet Quiet. Does not print anything to standard output.

Default: False

-j, --jobs Specify the number of jobs to run simultaneously. Default is one more than the number of CPUs.

Default: 3

-B, --always-compile Unconditionally make all targets

Default: False

Note that *evariste ARGS* and *evs compile ARGS* are the same command.

3.3 Setup file

The setup file contains:

- general configuration about how Évariste should handle this repository;
- configuration that is to be applied to every single file and directory (and that can be *overloaded later* (page 13)).

The file is parsed using `configparser`, you can use any feature of this module.

It is organized in sections (`setup`, `renderer.text`, etc.), each section containing some options.

```
[setup]
source = .
extends = foo.setup
cachedir = .foo.cache
libdirs =
    plugins1
    plugins2
plugins =
    vcs.git
    renderer.text
    action.autocmd
    action.command

[renderer.html.readme.mdwn]
enable = yes

[renderer.text]
ascii = True

[renderer.htmlplus]
enable = yes
destfile = public/index.html
destdir = public
staticdir = public/static

[renderer.htmlplus.templatevar]
title = My awesome title!
lang = fr
```

3.3.1 `[setup]` section

The only mandatory section is `[setup]`. Every other section is optionnal, and depends on which plugins are enabled. The options are:

- `source`: The root of the directory that is to be processed by Évariste, absolute (starting with `/` or relative to the directory of the setup file). Default is `.` (the same directory as the setup file).
- `extends`: A list of configuration files. If set, Évariste first loads the first of those files, then the second (overwriting options that are already defined), then the third, and so on, and loads this file last. This can be useful if you have: - one setup file that setup up the compilation of files (that you want to perform on your home computer or USB key); - another file that is processed by the continuous integration system of your public hosting software, that extends the first one by adding the generation of an HTML page.

- `cachedir`: The cache directory, if different from the default one.
- `plugins`: The list of plugins to enable. Note that plugins can also be enabled individually: see *Enabling plugins* (page 12). This list must include exactly one *VCS plugin* (page 32).
- `libdirs`: The list of directories the plugins (as python files) are to be searched in: see *Plugin paths* (page 40).

3.3.2 Enabling plugins

There are two ways to enable plugins, which can be used at the same time.

- `plugins` option of the `[setup]` section:

```
[setup]
plugins = foo bar baz
```

- `enable` option of the section of each plugin:

```
[foo]
enable = true
```

Évariste includes several plugins; you can also *write your own* (page 32). Those plugins are python files, that are searched in plugin directories: see *Plugin paths* (page 40).

3.3.3 Other sections

Each plugin can define its own sections (or read sections of other plugins). Generally, a plugin `foo` will have a corresponding section, and might have other sections `[foo.SOMETHING]`:

```
[foo]
bar = baz

[foo.bar]
toto = titi

[foo.baz]
tagada = tsoin tsoin
```

3.4 Per-file and per-directory configuration files

The *setup file* (page 10) applies to every single file and directory of the source directory. You might wants more granular settings.

3.4.1 File precedence

The deepest the configuration file, the more precedence it has. For instance, consider a file `foo/bar/baz.odt`. The list of setup and configuration files that apply (in that order) are:

- the *setup file* (page 10);
- `.evsconfig`;
- `foo/.evsconfig`;
- `foo/bar/.evsconfig`;
- `foo/bar/baz.odt.evsconfig`.

Note that if `foo/bar/baz.odt.evsconfig` is defined, other files are not discarded: they all are merged together, and if an option is defined in several files, the precedence order defined above applies.

3.4.2 Per-directory setting

The configuration set up in a `.evsconfig` file in a directory applies to this directory, and every file and directory included in it. To make it apply to this directory only, use the `recursive` option:

```
[setup]
recursive = false
```

3.4.3 Per-file setting

For any file `foo.bar`, you can define some setting that apply to this file and this file only in configuration file `foo.bar.evsconfig` or `.foo.bar.evsconfig`.

3.4.4 On configuration file names

Configuration files can have arbitrary names. If they contain a `source` option in the `setup` section, then this is considered to be the name (relative to the directory of this configuration file) that this configuration applies to.

For instance, if file `foo/bar/baz.evsignore` contains:

```
[setup]
source = ../toto/titi.txt
```

Then the configuration in this file applies to file `foo/bar/../toto/titi.txt`, that is `foo/toto/titi.txt`.

Using this feature, one can define *both* a recursive and non-recursive configuration for the same directory.

3.5 String formatting

In the *setup* (page 10) and *per-file and per-directory configuration* (page 13) files, strings related to path are formatted by replacing part of the string by part of the file name. Using this, one can define a string which applies to *all* files (instead of having to rewrite the option for every single file).

See *evariste.tree.Tree.format()* (page 58) for the list of replacements.

Note: Implementation detail

Right now, those strings are processed using `str.format()`, and you might want to use some of rich features of the Python string formatting. However, this is only an implementation detail, and might change in the future without notice.

3.6 Source

In the *Setup file* (page 10), the `source` option of section `[setup]` defines the directory that is to be processed by Évariste.

- *Configuration files* (page 15)
- *Ignore files* (page 15)
- *READMEs* (page 16)

3.6.1 Configuration files

To apply specitif configuration to a single file or directory, or to any subfile and subdirectory of a given directory, use: *Per-file and per-directory configuration files* (page 13).

3.6.2 Ignore files

You might want to ignore some files (no *compilation* (page 17), nor *rendering* (page 26)). There are two ways of doing this.

Ignore one file

For any file `foo.bar`, if a file `foo.bar.evsignore` or `.foo.bar.evsignore` exists, then `foo.bar` is ignored. The content of the `*.evsignore` files here is not read: their mere existence is sufficient.

Ignore several files

Patterns of file to ignore can be set in `.evsignore` files in any directory.

- Each line contains a pattern of some files to ignore.
- Empty lines and line starting with # are ignored.
- Lines starting with / match absolute path, while line not starting with / match relative path. For instance, let us consider the following directory tree.

```
+ foo
  + bar
  + baz
    + bar
```

  In a file `foo/.evsignore`, pattern `/bar` would ignore `foo/bar` but not `foo/baz/bar`, while `bar` would ignore both `foo/bar` and `foo/baz/bar`.

- In patterns:
  - * matches everything;
  - ? matches any single character;
  - `[seq]` matches any character in seq;
  - `[!seq]` matches any character not in seq.

3.6.3 READMEs

Annotation of files is implemented in the *HTML* (page 27) and *HTMLplus* (page 29) plugins.

CHAPTER

FOUR

PLUGINS

4.1 Mandatory plugins

The *following plugins* (page 46) are mandatory: they are enabled by default, and cannot be disabled. They are necessary for Évariste to run.

```
MANDATORY_PLUGINS = {
    "action.cached",
    "action.directory",
    "action.noplugin",
    "action.raw",
    "changed",
    "logging",
    "tree",
}
```

End user do not interact directly with most of them. Otherwise, they are documented elsewhere in this documentation.

4.2 Action plugins

Action plugins control how files should be compiled.

There is a list of plugins shipped with Évariste, but you can also *write your own* (page 39).

4.2.1 `action.autocommand` — Compile file according to mime type or extension

Like the *action.command* (page 19) plugin, this plugin is used to define which command should be used to compile some files. But with this plugin, you can define several rules that will apply dependinng of the mime type or extension of the file to compile.

Scope

Commands defined in the *setup file* (page 10) apply to the whole repository. Commands defined in the *configuration file* (page 13) of a directory recursively apply to this directory.

Options

Each rule is defined into its own section `[action.autocommand.FOO]`.

Common options of *action plugins* (page 17) and options of *action.command* (page 19) also apply here (`strace`, `command`, `targets`). New options are:

- `priority` (50): If several rules apply to a file, the one with highest priority applies.
- `extensions`: Space separated list of file extensions this rule should apply to.
- `mimetypes`: Space separated list of mime types this rule should apply to.

Note that:

- if `extensions` and `mimetypes` are both set, the rule applies to files that match *either* of them.
- if neither `extensions` nor `mimetypes` are set, the end of the section is considered to be the extension (that is, a section `[action.autocommand.ods]` with no `extensions` or `mimetypes` option would apply to `.ods` files).

Examples

- Compile LaTeX files using `latex+dvipdf`:

```
[action.autocommand.tex]
targets = {basename}.pdf
command =
    latex {basename}
    dvipdf {basename}
```

Note: Shameless self-promotion

If you have several LaTeX files that require different compilation tools, you might be interested in SpiX, which reads the compilation chain that has been written into the `tex` file itself.

- Convert OpenDocuments to PDF:

```
[action.autocommand.opendocument]
mimetypes = application/vnd.oasis.opendocument.*
extensions = fods fodt
command = libreoffice --headless --convert-to pdf {filename}
targets = {basename}.pdf
```

- Convert Gimp files to png:

```
[action.autocommand.xcf]
command = echo "\
      (define (convert-xcf-to-png filename outfile) \
         (let* \
            ( \
               (image (car (gimp-file-load RUN-NONINTERACTIVE␣
↪filename filename))) \
               (drawable (car (gimp-image-merge-visible-layers␣
↪image CLIP-TO-IMAGE))) \
            ) \
            (file-png-save RUN-NONINTERACTIVE image drawable␣
↪outfile outfile 0 9 0 0 0 0 0) \
         ) \
      ) \
      (convert-xcf-to-png \"{filename}\" \"{basename}.png\") \
      (gimp-quit 0)" | \
      gimp -i -b -
targets = {basename}.png
```

4.2.2 action.command — Explicitly set the command to compile a file

Using this action plugin, one can explicitely set the command used to compile a file.

Note: Althought one can configure this plugin in the *setup file* (page 10) or in the *configuration file* (page 13) of a directory, so that it applies to every single file of this repository or directory, you should probably use *action.autocommand — Compile file according to mime type or extension* (page 17) for this purpose.

Example

Let's say file `foo.tex` has the following configuration file `foo.tex.evsconfig`.

Listing 1: Example

```
[action]
plugin = command

[action.command]
targets = {basename}.pdf
command =
  latex {basename}
  dvipdf {basename}.dvi {basename}.pdf
```

The `plugin` option in the `[action]` section means that this plugin is to be used to compile this file. Then, in the `[action.command]` section:

- the `targets` option gives the name(s) of the compiled file(s);
- the `command` option defines the shell command to use to compile this file.

Note that strings are *formatted* (page 14).

Options

Here are the options of this plugin:

- `command` (`""`): Command to run.
- `strace` (`"false"`): If true, the command is run using strace to automatically find the dependencies of this file (the other files of this repository that are used to compile this file). Note that the compilation is slower, and this option is experimental.
- `targets` (`""`): Space-separated list of names of the compiled files. See *Targets* (page 22).
- `depends` (`""`): Space-separated list of names of the files that are used to compile this file. See *Depends* (page 23).

Example with action.autocommand

Imagine every single file of your repository is to be compiled with `pdflatex`, excepted for that file `foo.tex` that is to be compiled with `latex+dvipdf`. What you would do is:

- In the *setup file* (page 10) (or the *configuration file* (page 13) of the root directory), use *action.autocommand* (page 17) to specify that every LaTeX file should be compiled using `pdflatex`:

Listing 2: evariste.setup

```
[action.autocommand.latex]
extensions = tex
targets = {basename}.pdf
command = pdflatex {basename}
```

- In the *configuration file* (page 13) of `foo.tex` (that is: `foo.tex.evsconfig`), explicitely set the command to compile this file:

Listing 3: foo.tex.evsconfig

```
[action]
plugin = command

[action.command]
command =
    latex {basename}
    dvipdf {basename}
```

Since the configuration file for `foo.tex` has precedence over the other configuration files, or the setup file itself, this will do the trick.

4.2.3 `action.make` — Compile file using a Makefile

Compile file using a Makefile.

There is no automatic Makefile detection: you have to explicitely assign this action to a file.

Options

- `bin` (`make`): Binary.
- `options` (`""`): Options to call `make` with.

For a given targe `foo`, the command called is: `{bin} {options} foo`.

Listing 4: Example

```
[action.make]
bin = make
options = -j3
```

4.2.4 `action.raw` — Do not compile file: use file as-is

The file is not compiled. This is a default plugin (enabled by default, and cannot be disabled). For any file, if no other *action plugin* (page 17) matches, then this one is used as last resort.

4.2.5 Which plugin applies to which file?

Automatic selection

Each *action plugin* (page 49) have a *priority* (page 47) and a *match()* (page 47) method. By default, the plugin used to compile a file is the plugin with the highest priority that matches the file (i.e. `myplugin.match(file)` returns `True`).

Manual selection

However, user can explicitely choose a plugin for a given file, in an *evsignore* (page 15) file:

Listing 5: Manual selection of an action plugin

```
[action]
plugin = foo

[action.foo]
bar = options for plugin foo
```

4.2.6 Options

Each plugin has its own set of options. However, every action plugin accepts options *Targets* (page 22) and *Depends* (page 23).

Targets

Évariste cannot guess the name of the files that will be produced by an action. Use this option to define the targets, i.e. a space separated list of files that are generated by a given action. This option is *formatted* (page 14).

In the following example, the command produces two files: a *pdf* and a *png*.

Listing 6: Example of the `targets` option.

```
[action.command]
targets = {basename}.pdf {basename}.png
command =
   latex {basename}
   convert {basename}.pdf {basename}.png
```

Depends

The source files of a compiled one should not appear in the output of Évariste. By default, the file "trigerring" the action is considered the source file, and discarded. But, for instance, if you compile a *tex* files that includes a *png* image, you would like both files to be ignored in the final output. To do so, `depends` option is a space separated list of files that the compiled file depends on, and which should be discarded in the final output.

In the following example, the `image.png` file will be ignored in the final output.

Listing 7: Example of the `depends` option.

```
[action.command]
depends = image.png
targets = {basename}.pdf
command = pdflatex {basename}
```

Note that the *command plugin* (page 19) (and the plugins that inherit from it) accept the experimental, slower *strace option* (page 20), which automatically detects the files that the compiled file depends on.

4.3 Misc plugins

Some plugins that do not belong to any other category.

4.3.1 copy — Copy files at the end of compilation

In the `copy` section of the *setup file* (page 10), each option starting with `copy` is a copy instruction: the first words are source paths, the last one is the destination path. All paths (source and destination) are relative to the directory of the setup file.

Consider the source path `foo` and the destination path `dest`.

- If the source is a file, it is copied into the destination: `foo` is copied to `dest/foo`.
- If the source is a directory, its content is copied into the destination: `foo/bar` is copied to `dest/bar`.

`Pattern matching` is performed to the source files:

- ?: a single character;
- *: everything, excepted directory separators;
- **: everything, including directory separators;
- `[seq]` any characteur in `seq`.

Listing 8: Example

```
[copy]
copy_foo = foo* bar baz
copy_toto =
  toto
  titi*
  tata
```

4.3.2 `debug.hooks` — Print hook calls

This plugins can help *writing new plugins* (page 32): it prints to standard output each *hook* (page 35), and a few more things.

4.4 Logging plugins

Warning: The logging plugins are not the first ones to be loaded. So other plugins might have logged things using the Python `logging` module when those plugins start handling logs.

Those plugins define how log is displayed. To select a logger (i.e. a plugin logger), use the configuration file:

Listing 9: Enable the *logging.foo* plugin.

```
[logging]
logger = foo
```

In the above example, plugin `logging.foo` is used as the logging plugin (if this option is not set, `logging.auto` is used by default).

4.4.1 `logging.quiet` — Does not log anything

Note that things logged *before* this plugin is enabled are still logged using the default Python module.

4.4.2 `logging.stdlib` — Use the Python `logging` module

Default logger, that uses the `logging` Python module. The format string can be set in the configuration file:

Listing 10: Define format string

```
[logging.stdlib]
format = %%(asctime)s XXX %%(message)s
```

You can use the attribute names defined by the logging module. Note that you need to escape `%` with double `%`, because `configparser` formats strings found in configuration files.

4.4.3 `logging.auto` — Automatic plugin selection

If standard output is a tty, use the `logging.rich` plugin. Otherwise, log without frills using the `logging.stdlib` plugin.

4.4.4 `logging.rich` — Logging with colors and progress bar

Log stuff using colors and a progress bar (uses the `rich` module).

4.5 Renderer plugins

Renderer plugins define what should be done at the end of the compilation: *display something in the standard output* (page 31), *build an HTML page* (page 27), etc. Some plugins are shipped with Évariste, but you can also *write your own* (page 40).

4.5.1 `renderer.jinja2` — `jinja2` renderer

This plugin is an abstract plugin: it cannot be directly used, but several plugins with common features inherit from it.

This page describes those common features.

Options

Here are the common options to any plugin that is a subclass of this one.

- `templatedirs` : Additionnal directories where templates are being searched. By default, the following directories are used:
  - some directory containing the default templates of this plugin;
  - `.evariste/templates` (relative to the directory of the *setup file* (page 10));
  - `~/.config/evariste/templates`;
  - `~/.evariste/templates`;
  - `/usr/share/evariste/templates`.
- `template`: The name of the template used to render the tree.

Template

The following template variables are defined:

- `destdir`: Destination directory.
- `shared`: `shared` data (see *`evariste.builder.Builder.shared`* (page 43)).
- `local`: Local reference to the shared data (see *`evariste.shared.Shared.get_plugin_view()`* (page 56)).
- `sourcepath`: Source path of the repository.
- `render_file`: Function that renders the *`file`* (page 60) given in argument (this functions uses the *file renderers* (page 29)).
- `render_readme`: Function that renders the README of a file (this functions uses the *README renderers* (page 29)).
- `render_template`: Function that renders the template given in argument.

- `templatevar`: Dictionary of template variables (see *Template variables* (page 27)).
- `tree`: The *Root* (page 61) being rendered.

Template variables

It can be convenient to define template variables in the *setup file* (page 10) (the *htmlplus* (page 29) plugins uses this). A dictionnary `templatevar` is available in the template, and contains the following items:

- `date`: Compilation date.
- `time`: Compilation time.
- `datetime`: Compilation date and time.
- `aftertree`: A credit line (with the date and Évariste version, and a link to the Évariste website).

It also contains any option that has been defined in the setup file, in the `renderer.{keyword}.templatevar` option (where `keyword` is the keyword of the plugin).

Listing 11: Example of template variables for the HTML template

```
[renderer.html.templatevar]
title = This is the value of the <em>title</em> jinja2 template␣
↪variable.
```

4.5.2 `renderer.html` — HTML renderer

This plugin renders the repository as an HTML tree with annotated files, both as source and compiled. Note that you might want to use *renderer.htmlplus* (page 29) instead.

- *renderer.html.readme.rst — RestructuredText README renderer* (page 29)

Options

Options are defined in section `renderer.html` of the *setup file* (page 10). The *options of any jinja2 plugin* (page 26) also apply, and this plugin also defines:

Listing 12: example

```
[renderer.html]
destfile = index.html
destdir = html
href_prefix = html/
```

- `destfile` (`"index.html"`) : Destination file.
- `destdir` (`"html"`) : Destination directory: the source and compiled files will be copied there (respecting the tree structure of the original repository).
- `href_prefix` (`""`) : This string is added at the beginning of each link to the source and compiled files in the destination file.
- `template` (`"tree.html"`) : The name of the template used to render the tree. The default templates is only an HTML list. If you want a full HTML page, see *renderer.htmlplus — HTML renderer, with a bit of CSS and javascript* (page 29).

Some template variables can also be defined in the setup file. See *Template variables* (page 28).

Template

The template variables defined in any Jinja2 renderer are available in any HTML template as well. See *Template* (page 26).

Template variables

The `templatevar` mechanism defined for any Jinja2 renderer are available in any HTML template as well. See *Template variables* (page 27).

File plugins

Every single file is not rendererd the same way. You can enable plugins to configure this.

`renderer.html.file.default` — Default file renderer

This plugin is enabled by default.

`renderer.html.file.image` — Render images

This plugins displays a thumbnail of the image next to its name.

Annotation: README plugins

READMEs can be written in several languages.

`renderer.html.readme.html` — HTML README renderer

Given a file `foo`, a `foo.html` will be pasted raw as its annotation.

`renderer.html.readme.mdwn` — Markdown README renderer

Given a file `foo`, a `foo.md` or `foo.mdwn` will be rendered as its annotation.

`renderer.html.readme.rst` — RestructuredText README renderer

Given a file `foo`, a `foo.rst` will be rendered as its annotation.

4.5.3 `renderer.htmlplus` — HTML renderer, with a bit of CSS and javascript

Like *renderer.html — HTML renderer* (page 27), this plugin renders the repository as an HTML tree with annotated files, both as source and compiled. The difference is that it adds a bit of CSS and javascript to make the end result nicer.

Options

Options are defined in section `renderer.htmlplus` of the *setup file* (page 10).

Listing 13: example

```
[renderer.htmlplus]
destfile = index.html
destdir = html
href_prefix = html/
display_log = no
```

The options of *renderer.html — HTML renderer* (page 27) also apply here. This plugin adds the following options.

- `template` (`"page.html"`): Name of the template to use to render the page. This option has the same meaning as the one in *renderer.html — HTML renderer* (page 27), but the default value is different: by default, it renders a full HTML page (instead of some HTML code to be includede into an HTML page).
- `staticdir` (`"static"`): Directory (relative to the directory of the *setup file* (page 10) where static files (CSS and Javascript files) should be copied at the end of compilation.
- `display_log` (`"errors"`): Defines what to do with compilation logs. - `"yes"`: Include all logs. This can produce huge HTML pages. - `"no"`: Do not include any log. - `"errors"`: Only include logs of files when compilation failed.

Some template variables can also be defined in the setup file. See *Template variables* (page 28).

Template and template variables

The *template variables* (page 28) defined in the *HTML plugin* (page 27) are also defined here. Moreover, the following variables may be defined in the *setup file* (page 10) to be included in the default `page.html` template:

- `lang`: Language of the page (to be included in the <`html`> tag as <`html lang={{ lang }}`>).
- `title`: Title of the page (as the `title` tag).
- `favicon`: Link to the favicon.
- `head`: Additionnal code to be included at the end of the <`head`> tag.
- `header`: Some HTML code to be included in the body, before the tree.
- `footer`: Some HTML code to be included in the body, after the tree. Default is some credit to Évariste.

All of them are optional.

Listing 14: Example of template variables

```
[renderer.htmlplus.templatevar]
title = This is the value of the {{title}} jinja2 template variable.
```

File and README plugins

The *README plugins* (page 29) and *file plugins* (page 29) of the *HTML renderer* (page 27) also work with this renderer.

4.5.4 `renderer.text` — Text renderer

At the end of compilation, display (in standard output) a tree to sum up the compilation: which files were successfully compiled, which were not...

Listing 15: Example

```
[renderer.text]
enable = true
color = true
display = errors_or_all
```

Options

The default value is given between parenthesis.

`color` (`auto`)
: If `True`, use color to draw the tree. If `auto`, use color only if standard output is not piped or redirected.

`ascii` (`False`)
: If `True` draw tree structure using only ASCII characters. Otherwise, use a wider set of characters.

`reverse` (`False`)
: If `True`, draw tree in reverse order.

`display` (`all`)
: Define which files should be displayed at the end of compilation.

  `all`
  : Display all files.

  `errors`
  : Only display files when their compilation failed.

errors_or_all
If some files did not compile successfully, only display those files. Otherwise, display all files.

4.6 VCS plugins

Those plugins defines which files should be considered by Évariste.

4.6.1 vcs.fs — Process any file of the file system

Consider every file.

4.6.2 vcs.git — Only process files handled by git

Only consider files handled by git. This prevents writing tedious *evsignore* (page 15) files to ignore compiled files, while those are typically ignored by git itself.

4.6.3 vcs.none — Do not process any file

This plugin is used in tests. I do not see why it would be useful to you, but who knows?

4.7 Write your own plugin

4.7.1 Minimum example

A plugin is a subclass of *Plugin* (page 46). Define such a class in a python file located in *the right directory* (page 40).

```
from evariste import plugins

class Foo(plugins.Plugin):
    """Example plugin"""

    keyword = "foo"
```

The only mandatory attribute or method is the `keyword` attribute, which must be unique. It will be used to enable your plugin in the setup file.

That's it! You can now enable it in the *setup file* (page 10):

```
[setup]
plugins = foo
```

You are now a proud owner of a plugin that does... nothing. To interact with Évariste, you can:

- implement some *hooks* (page 35);
- for some plugin types, implement some methods (see for instance *VCS plugins* (page 32) or *Action plugins* (page 17)).

Of cours, your plugin can do everything listed above, at once.

4.7.2 Attributes

Several useful attributes are defined for every `Plugin` (page 46) instance; they are defined in the class documentation. The most complex one are `Plugin.shared` (page 47) and `Plugin.local` (page 47).

Plugin.shared

This attribute is a `Shared` (page 56) instance, shared among every `Plugin` (page 46) and `Tree` (page 57) object. It has three attributes, which are all `DeepDict` instances (in the following examples, `shared` is an instance of `Shared` (page 56)):

- `setup` is a representation of the *setup file* (page 10). For instance, option `bar` of section `foo` can be read (and set) as `shared.setup["foo"]["bar"]`.
- `plugin` is a `DeepDict` where each plugin can store data that is cached, and accessible from other plugins. Plugin `foo` can set `shared.plugin["foo"]` at whatever value it wants. Technically, you can get and set values for other plugins, but think twice before doing so: do the other plugin expect you to get and set its data?
- `tree` is a `DeepDict` where each plugin can store data about tree instances that is cached, and accessible from other plugins. Plugin `foo` can set whatever information its want about `Tree` (page 57) instance `tree` in `shared.tree[tree]["foo"]`.

Data that is set in `Shared` (page 56) attributes `plugin` and `tree` is `pickled` so make sure data you save there are *picklable*.

Plugin.local

Most of the time, your plugin will only access its own section in the *setup file* (page 10), or in the other attributes of the *Plugin.shared* (page 33) attribute. To make things easier, the very same data is also available in `Plugin.local` (page 47). Let's consider an instance `foo` of a plugin `foo`

- `foo.local.setup` is a dictionary of the options of `foo` in the *setup file* (page 10): `foo.local.setup` is a shortcut for `foo.shared.setup["foo"]`.
- `foo.local.plugin` is a shortcut for `foo.shared.plugin["foo"]` (cached data for this plugin).
- Given a `Tree` (page 57) object `mytree`, then `foo.local.tree[mytree]` is a shortcut for `foo.shared.tree[mytree]["foo"]`.

Plugin.default_setup and Plugin.global_default_setup

For any plugin, attributes *Plugin.default_setup* (page 46) is default setup of the section *Plugin.keyword* (page 47), while *Plugin.global_default_setup* (page 47) is the whole default setup (for all sections).

When reading the setup file, options that are not set are filled with options of *Plugin.default_setup* (page 46), and sections that are not set are filled with sections of *Plugin.global_default_setup* (page 47).

For instance, consider the following plugin:

```
class Foo(Plugin):
    keyword = "foo"
    default_setup = {
        "foo1": "default1",
        "foo2": "default2",
        }
    global_default_setup = {
        "bar": {
            "bar1": "global1",
            "bar2": "global2",
            },
        "foo": {
            "foo1": "global1",
            "foo3": "global3",
            },
        }
```

Now, this plugin is loaded with the following *setup file* (page 10):

```
[setup]
plugins = foo

[foo]
foo2 = setup2
foo4 = setup4

[bar]
bar1 = setup1
bar3 = setup3
```

Then, once the setup file, and both *Plugin.default_setup* (page 46) and *Plugin.global_default_setup* (page 47) has been taken into account, the resulting setup is equivalent to:

```
[setup]
plugins = foo
```

(continues on next page)

(continued from previous page)

```
[foo]
foo1 = default1
foo2 = setup2
foo4 = setup4

[bar]
bar1 = setup1
bar2 = global2
bar3 = setup3
```

Notice that:

- whatever have been set in the setup file is kept;
- options of *Plugin.default_setup* (page 46) and *Plugin.global_default_setup* (page 47) may be overwritten by the setup file;
- whole sections of *Plugin.global_default_setup* (page 47) may be overwritten by the section of *Plugin.default_setup* (page 46).

4.7.3 Current working directory

Note that as early as possible, the working directory is changed to the directory of the setup file given in argument to *evariste* (page 10).

4.7.4 Interacting with Évariste

Hooks

Registering a method of your plugin as a hook means that this method will be called at a particular point during *evariste* (page 10) execution.

Hook types

Method hooks

Method hooks are defined as decorator: they return a wrapped function, may (or may not) call the original function, and may (or may not) change the returned value.

Listing 16: Example of a method hook.

```
@methodhook("File.make_archive")
def make_archive(self, function):
    """Do something wile building archive"""
```

(continues on next page)

(continued from previous page)

```
    @functools.wraps(function)
    def wrapped(tree, destdir):
        """Wrapped function."""
        # Do something before original function call.
        # Then call the original function.
        value = function(tree, destdir)
        # Do something after the original function call.
        # Maybe change the returnd value.
        return value

    return wrapped
```

Context hooks

Most of the time, you want to use a method hook, without the hassle of defining a wrapped function (because you won't change the arguments or return value of the original function call). Any hook defined as a method hook can also be used as a context hook.

Your function must be a context manager (a `contextlib.contextmanager()` would make it even easier). Besides `self`, it is passed the arguments of the original function, and that original function is called between the `__enter__()` and `__exit__()` calls.

Listing 17: Example of a context hook

```
@contexthook("Builder.compile")
@contextlib.contextmanager
def builder_compile(self, builder):
    # Do something before calling the original function

    # Call the original function
    yield

    # Do something after having called the original function
```

Iteration hooks

The last hook type are *iteration hooks*. Functions registered as such a hook must be iterators, and Évariste will aggregate all the item iterated by all functions registered as this hook.

Listing 18: Example of an iteration hook

```
@iterhook("Tree.prune_before")
def foo(self, tree):
    yield from self.bar(tree)
```

Chronological list of hooks

Here is the chronological list of hooks, that is, the list of the hooks, in the order in which they are called when running Évariste.

Just enable the *debug.hooks plugin* (page 24) to print this list to standard output.

1. An instance of the plugin is created: `plugins.Plugin.__init__()`.
2. `Tree` (page 57) (*Method hooks* (page 35)): for every file and directory in the *repository* (page 32), the *Method hooks* (page 35) `Tree` is called (`__enter__()` and `__exit__()`).
3. `Builder.compile.__enter__()` (page 43) (*Method hooks* (page 35)): About to build the tree.
4. `Tree.prune_before()` (*Iteration hooks* (page 36)): Methods must iterate files that will be pruned from the tree before file compilation (files and directories that won't be compiled, and won't appear in the final output). This method is called once for every file and directory of the tree. Argument: a `Tree` (page 57) object.
5. `File.compile.__enter__()` (page 60) (*Method hooks* (page 35)): About to compile the file. This method is called for every file in the repository. Note that file compilation is done in threads, so you don't know in which order files will be compiled.
6. `File.compile.__exit__()` (page 60) (*Method hooks* (page 35)): Done compiling the file. Same remarks as above.
7. `Tree.prune_after()` (*Iteration hooks* (page 36)): Methods must iterate files that will be pruned from the tree after file compilation (files and directories that may have been compiled, and won't appear in the final output). This method is called once for every file and directory of the tree. Argument: a `Tree` (page 57) object.
8. `File.make_archive.__enter__()` (page 60) (*Method hooks* (page 35)): About to build the archive of the current file and *its dependencies* (page 23). This hook is called once for every file in the repository.
9. `File.make_archive.__exit__()` (page 60) (*Method hooks* (page 35)): Done building the archive.
10. `Builder.compile.__exit__()` (page 43) (*Method hooks* (page 35)): Done building the tree.
11. `Builder.close.__enter__()` (page 43) (*Method hooks* (page 35)): About to close the builder. This method is only called if compilation was successful.
12. `Builder.close.__exit__()` (page 43) (*Method hooks* (page 35)): About to close the builder. See remark above.

Create your own hooks

Method and Context hooks

Defining a new method hook is done using the *hooks.setmethodhook()* (page 45). The following example defines a context hook.

Listing 19: Definition of a context hook

```
from evariste.hooks import setmethodhook

class Foo:

    @setmethodhook()
    def bar(self, baz):
        bla_bla_bla()
```

Then, any plugin can register the method or context hook `Foo.bar` (class name dot method name, or class name only for the constructor) that will be called whenever method `Foo.bar()` is called.

Any method hook is also a context hook (and it is not possible to define a context hook that is not a method hook).

Iteration hooks

Iteration hooks are defined using the *plugins.Loader.applyiterhook()* (page 47) function (the *plugins.Loader* (page 47) instance being an attribute of the *builder.Builder* (page 43) one).

For instance, if a plugin contains the foolowing lines:

Listing 20: Definition of an iteration hook

```
for item in self.shared.builder.plugins.applyiterhook("foo", bar):
    baz(bar)
```

Then, every method registered as an iteration hook `foo` will be called with the argument `bar`, and whatever they iterate will be iterated in the *for* loop in this example.

Write action plugins

An action plugin is a subclass of *Action* (page 49), that must interpret its `abstract methods`.

Selection

An action plugin has an *match()* (page 49) method and a *priority* (page 47) attribute. To choose which action plugin it should use to compile a `foo` *file* (page 60), Évariste looks for the action plugin with the highest priority, that matches the file (that is: `myplugin.match(foo)` returns True). The algorithms looks like the following:

Listing 21: Algorithm to choose the action plugin used to compile a `foo` file.

```
# Plugins are sorted by their priority attribute
for plugin in sorted(LIST_OF_ACTION_PLUGINS, reverse=True):
    if plugin.match(foo):
        return plugin
```

Threads

The *compile()* (page 49) action must be thread safe. If not, a `Lock` is shared by every action plugin (as attribute *lock* (page 49)).

Listing 22: Example of usage of *lock* (page 49)

```
def compile(self, path):
    # Thread safe part
    foo()

    with self.lock:
        # Non thread-safe part
        bar()

    # Thread safe part
    baz()
```

Write renderer plugins

Contrary to *action* (page 39) and *VCS* (page 40) plugins, renderer plugins are plain *Plugin* (page 46), that implement interesting stuff at the end of the *Builder.compile hook* (page 35).

Jinja2 renderer

If you plan to write a renderer that write some file using the jinja2 module, you should probably subclass *jinja2.Jinja2Renderer* (page 50).

HTML renderer

The *HTMLRenderer* (page 53) is a subclass of *Jinja2Renderer* (page 50) (see above). If you want it to render files and README differently, you can write a file or README renderer, which are subclasses of *HtmlFileRenderer* (page 54) and *HtmlReadmeRenderer* (page 54).

Write VCS plugins

A VCS plugin is a subclass of *VCS* (page 55), that must interpret its `abstract methods`.

Note that it is also possible to write *evs plugins* (page 42).

4.8 Plugin paths

Évariste looks fro new plugins (as python packages) in the following directories (this is relevant when *writing* (page 32) or installing new plugins):

- `.evariste/plugins/foo.py` (relative to the directory of the setup file);
- `~/.local/evariste/plugins/foo.py`
- `~/.evariste/plugins/foo.py`
- `LIBDIR/foo.py` (where `LIBDIR` is any directory of the *libdirs* (page 40) setup option).

CHAPTER

FIVE

EVS TOOLS

Some helpers tools are installed together with Évariste. They are mostly meant to be used by developers (of Évariste, or plugins) rather than end users.

5.1 *evs cache* — Cache management

Using this tool, one can display, explore, or clean cache.

5.2 *evs plugins* — Plugin management

Using this tool, one can display the list of available plugins.

5.3 *evs compile* — Run Évariste

The `evariste` binary is actually a shortcut to this subcommand.

5.4 Write your own

If you want to write your own `evs` tool, simply place an executable file named `evs-foo` in a directory of the `PATH` shell variable. It will be called when calling `evs foo`, with the same command line arguments.

CHAPTER

SIX

LIBRARY DOCUMENTATION

Modules, classes, functions and constants are documented here.

6.1 evariste.builder

Build process: gather files, and compile them.

class evariste.builder.**Builder**(*setup*)

Takes care of build process. Can be used as a context.

shared: *evariste.shared.Shared* **(page 56)**

Object that is shared and accessible by every *evariste.plugins.Plugin* (page 46) and *evariste.tree.Tree* (page 57). See *Plugin.shared* (page 33).

cache: **evariste.cache.Cache**

Data that is cached between compilations. Plugin developpers won't manipulate this attribute directly (see *Plugin.shared* (page 33)).

plugins: *evariste.plugins.Loader* **(page 47)**

Plugin loader: loaded plugins are gathered there.

close()

Perform close operations.

Mainly used as a *Method hooks* (page 35).

compile()

Compile files handled by this builder.

classmethod from_setupdict(*dictionary: Dict[str, Dict[str, str]]*) → *Builder* (page 43)

Factory that returns a builder, given a setup dictionary.

A *setup dictionary* is a dict that mimics configparser structure.

classmethod from_setupname(*name: str*) → *Builder* (page 43)

Factory that returns a builder, given the name of a *setup file* (page 10).

6.2 evariste.hooks

Implement hook mechanism.

See *Hooks* (page 35) for more information.

Note: This implementation of hooks rely on other parts of Évariste (*plugins* (page 46) for example), and cannot be used separatedly.

6.2.1 Example

Listing 1: Example of hook mechanism

```
import contextlib

from evariste import hooks

class A:

    @hooks.setmethodhook()
    def a(self):
        print("Running A.a()...")

class B:

    @hooks.contexthook("A.a"):
    @contextlib.contextmanager
    def b(self):
        print("Before running A.a()...").
        yield
        print("After running A.a()...").

# Let's go!
A.a()
```

In this example, the `A.a()` method has been marked as accepting hooks, and the `B.b()` method has been registered as a hook for `A.a()`.

When `A.a()` is run (last line of the example), although `B` has not been called directly, `B.b()` is called as well, as a registered hook. The output of this example is:

```
Before running A.a()...
Running A.a()...
After running A.a()...
```

6.2.2 Get functions registered as hooks

6.2.3 Method hooks

Methods can be marked to accept hooks using the following function.

evariste.hooks.**setmethodhook**(*, *getter: None | Callable = None*) → Callable

Decorator to mark that a method can accept method and context *Hooks* (page 35).

Parameters

getter (*function*) – Function that, given the instance object as argument, returns a `plugins.Loader` object. If `None`, the default `self.shared.builder.plugins` is used (`self` is supposed to have this attribute).

6.2.4 Context hooks

Context hooks cannot be directly defined: every method hook is also a context hook.

6.2.5 Iteration hooks

Iteration hooks can be executed using *`applyiterhook()`* (page 47).

6.2.6 Register functions as hooks

evariste.hooks.**hook**(*hooktype: str*, *name: str*) → Callable

Decorator to register a function or method as a hook.

Parameters

- **hooktype** (*str*) – Type of hook (`"methodhook"` or `"contexthook"`, or whatever string you want).
- **name** (*str*) – Name of the target hook, of the form `Class.methodname` (or `Class` only for the `__init__` method).

evariste.hooks.**contexthook**(*name: str*) → Callable

Decorator to register a function or method as a context hook.

For any string `name`, `contexthook(name)` is a shortcut for `hook("contexthook", name)`.

evariste.hooks.**methodhook**(*name: str*) → Callable

Decorator to register a function or method as a method hook.

For any string `name`, `methodhook(name)` is a shortcut for `hook("methodhook", name)`.

evariste.hooks.**iterhook**(*name: str*) → Callable

Decorator to register a function or method as an iter hook.

For any string `name`, `iterhook(name)` is a shortcut for `hook("iterhook", name)`.

6.3 evariste.plugins

Plugin base class and plugin loader

Every plugin is a subclass of *Plugin* (page 46) (see *Write your own plugin* (page 32) for more information).

The *Loader* (page 47) class finds and loads the plugins.

6.3.1 Constants

evariste.plugins.**MANDATORY_PLUGINS = {'action.cached', 'action.directory', 'action.noplugin', 'action.raw', 'changed', 'logging', 'tree'}**

Set of mandatory plugins: plugins that are loaded by default, and cannot be disabled.

6.3.2 Plugin

class evariste.plugins.**Plugin**(*shared*)

Plugin base: all imported plugins must be subclasses of this class.

See *Write your own plugin* (page 32) to see how to write a new plugin.

Parameters
shared (Shared (page 56)) – The object *shared* (page 33) among plugins.

default_setup: Dict[str, str] = {}

Default value for section `self.keyword` in the *setup file* (page 10). It may be overwritten by data provided by user in the *setup file* (page 10). See *Plugin.default_setup and Plugin.global_default_setup* (page 34).

depends: Iterator[str] = ()

Iterable of plugins this plugin depends on. When this plugin is enabled, those plugins are enabled as well.

classmethod depends_dynamic(*shared*) → Iterator[str]

Iterator of plugins this plugin depends on (as an iterator of `str`)

When this plugin is enabled, those plugins are enabled as well.

Parameters
shared (Shared (page 56)) – Shared object of the current builder.

Warning: called before everything is settled down

global_default_setup: Dict[str, Dict[str, str]] = {}

Default values for setup file. See *Plugin.default_setup and Plugin.global_default_setup* (page 34).

keyword: None | str = None

Keyword plugin, used to reference it: it is used to *enable plugins in the setup file* (page 12), to name its section in the *setup file* (page 10), etc.

local

Same as *Plugin.shared* (page 47), but from this plugin point of view: see *evariste.shared.Shared.get_plugin_view()* (page 56) and *Plugin.local* (page 33).

match(*value*, **args*, ***kwargs*) → bool

Return `True` iff `value` matches `self`.

Default is keyword match. This method can be overloaded by subclasses.

plugin_type: str = ''

Type of the plugin. Plugins of the same type gather some common behaviour.

priority: int = 0

When Évariste has to choose *one* plugin among several one, it chooses the one with higher priority.

shared: *Shared* (page 56)

Common data shared with every `Tree` and *Plugin* (page 46) of this *Builder* (page 43).

6.3.3 Loader

class evariste.plugins.**Loader**(*, *shared*)

Load plugins

Parameters
shared (evariste.shared.Shared (page 56)) – The *shared* (page 33) object among plugins.

The constructor (`Loader.__init__()`):

- reads the *setup file* (page 10) (looking for the *libdirs* (page 40) option);
- search all plugins (subclasses of *Plugin* (page 46));
- instanciate:
  - the *mandatory plugins* (page 46),
  - those enabled in the *setup file* (page 10),
  - and their dependencies;
- store them in some attribute, so that they can be accessed later.

applyiterhook(*hookname: str*, **args*, ***kwargs*) → Iterable

Apply an *iteration hook* (page 36).

Run every fonction hooked to this name (they should be iterators), and iterate over the chain of those iterators.

Parameters

- **hookname** (*str*) – Name of the hook to apply.
- **arg** (*list*) – Positional arguments passed to the hooks.
- **kwargs** (*dict*) – Named arguments passed to the hooks.

get_plugin(*keyword: str*) → *Plugin* (page 46)

Return the plugin with the given keyword.

Raises

NoMatch (page 49) – If no (loaded) plugin was found with this keyword.

items(*plugin_type: str | None = None*) → Iterable[Tuple[str, *Plugin* (page 46)]]

Iterate over plugin keywords.

Parameters

plugin_type (*Optional[str]*) – See *Loader.iter()* (page 48).

iter(*plugin_type: str | None = None*) → Iterable[*Plugin* (page 46)]

Iterate over keywords.

Parameters

plugin_type (*Optional[str]*) – Type of the plugins to iterate over.

- if `None`, iterate over keywords of every (loaded) plugins;
- else, iterate over keywords of plugins of this given type only.

match(*plugin_type: str | None*, *value*) → *Plugin* (page 46)

Return the first plugin matching `value`.

A plugin `Foo` matches `value` if `Foo.match(value)` returns `True`.

Parameters

plugin_type (*Optional[str]*) – See *Loader.iter()* (page 48).

values(*plugin_type: str | None = None*) → Iterable[*Plugin* (page 46)]

Iterate over plugins.

Parameters

plugin_type (*Optional[str]*) – See *Loader.iter()* (page 48).

6.3.4 Functions

evariste.plugins.**find_plugins**(*libdirs: Iterable[str] | None = None*) → Iterator[*Plugin* (page 46)]

Iterate over available plugins.

Parameters

libdirs (*Iterable[str]*) – Additional iterable of directories where plugins can be found.

6.3.5 Exceptions

class evariste.plugins.**NoMatch**(*value*, *available*)

No plugin found matching `value`.

The *Plugin* (page 46) class has a handful of subclasses.

6.3.6 evariste.plugins.action

Actions performed to compile files.

The result of an *action* (page 49) (Was it sucessful? Which files were used? What is the log? etc.) is stored as a *report* (page 50).

If you plan to write your own action plugin, see *Write action plugins* (page 39).

Action

class evariste.plugins.action.**Action**(*shared*)

Generic action

Subclass this to create a new action (see *Write action plugins* (page 39)).

abstract compile(*path: Tree (page 57)*) → *Report* (page 50)

Compile `path`.

This function *must* be thread-safe. It can use *Action.lock* (page 49) if necessary.

lock: threading.Lock = <unlocked _thread.lock object>

A lock shared by every action. Can be used for parts of the compilation which are not thread-safe.

match(*value: Tree (page 57)*) → bool

Return `True` if `value` can be compiled by this action.

plugin_type: str = 'action'

Type of the plugin. Plugins of the same type gather some common behaviour.

Report

class evariste.plugins.action.**Report**(*path*, *targets=None*, *success=False*, *log=None*, *depends=None*)

Report of an action. Mainly a namespace with very few methods.

property **full_depends**: Set[Path]

Set of files this action depends on, including `self.path`.

property **success**: bool

Was compilation sucessful?

6.3.7 evariste.plugins.renderer

This plugin does not define anything directly, but is interesting because of its submodules.

evariste.plugins.renderer.jinja2

Abstract class for jinja2 renderers.

See also *renderer.jinja2 — jinja2 renderer* (page 26).

class evariste.plugins.renderer.jinja2.**Jinja2Renderer**(*shared: Shared (page 56)*)

Abstract class for jinja2 renderers.

To write your own renderer:

- subclass this class;
- define a default template name: *Jinja2Renderer.template* (page 51);
- write such a template file, and place it in one of the *templatedirs* (page 26). The following template variables are defined and can be used in the template: *Template* (page 26);
- you can also overwrite the methods defined here.

You might also have a look at the implementation of the *HTML renderer* (page 53).

- Each file can be rendered in its own way: see *Jinja2FileRenderer* (page 51) (for instance, you might want to add a nice thumbnail to files that are images);
- To define how files are annotated, see *Jinja2ReadmeRenderer* (page 52).

default_setup: Dict[str, str] = {'destfile': 'output'}

Default value for section `self.keyword` in the *setup file* (page 10). It may be overwritten by data provided by user in the *setup file* (page 10). See *Plugin.default_setup and Plugin.global_default_setup* (page 34).

get_readme(*tree:* Tree *(page 57)*) → *Tree* (page 57)

Iterate the only README file for `tree`.

If there is such a README file, iterate over it (a single value); otherwise, iterate nothing.

Side effect: Store a (partial) function in `self.readmes[tree.from_source]` to render this README file.

iter_subplugins(*subtype: str*) → Iterable[*Plugin* (page 46)]

Iterate over subplugins of type `subtype`.

This method iterates plugins (as their keywords) `{keyword}.{subtype}`, where `keyword` is the attribute of this class, or its subclasses.

For instance, given that:

- the correct plugins are loaded;
- plugin *renderer.html* (page 53) is a subclass of *renderer.jinja2* (page 50),

call to `Jinja2Renderer.iter_subplugins(HtmlRenderer(), "readme")` will yield: `renderer.html.readme`, `renderer.html.readme.mdwn`...

render(*builder:* Builder *(page 43)*) → None

Render the tree as a file, and write result into the destination file.

render_tree(*tree:* Tree *(page 57)*) → str

Render the tree using templates, and return the string.

template: `str` **= None**

Name of the default template.

evariste.plugins.renderer.jinja2.file

Abstract utilities for file renderers using Jinja2.

class evariste.plugins.renderer.jinja2.file.**Jinja2FileRenderer**(*shared*)

Renderer of file using jinja2.

This is an abstract class that defines a default renderer for files.

From within a template, the macro `render_file` can be called, which:

- looks for the first plugin that matches this file (that is, the first plugin where *Jinja2FileRenderer.match()* (page 52) returns `True`;
- calls *Jinja2FileRenderer.render()* (page 52), and returns its return value.

To implement such a renderer, you can:

- write a `file/default` template that defines a `file()` macro;
- set the *Jinja2FileRenderer.extension* (page 52), and write a `file/default.`*extension* template, that defines a `file()` macro;

- overwrite the *Jinja2FileRenderer.render()* (page 52) method, if the default implementation does not pleas you.

You can also overwrite *Jinja2FileRenderer.match()* (page 52), so that your subplugin cannot be applied to *any* file, but only to some of them.

extension: Optional[str] = None

Extension that is automatically added at the end of the template name when searching them.

keyword: Union[None, str] = None

Keyword plugin, used to reference it: it is used to *enable plugins in the setup file* (page 12), to name its section in the *setup file* (page 10), etc.

match(*filename:* Tree *(page 57)*) → bool

This is the default renderer, that matches everything.

priority: int = -inf

When Évariste has to choose *one* plugin among several one, it chooses the one with higher priority.

render(*filename:* Tree *(page 57)*, *context: jinja2.runtime.Context*) → str

Render `tree`, which is a *File* (page 60)

By default, call the `file()` macro, with `filename` as argument, and returns its value.

template: str = 'default'

Name of the template rendering files.

evariste.plugins.renderer.jinja2.readme

Common utilities for readme renderers using Jinja2.

class evariste.plugins.renderer.jinja2.readme.**Jinja2ReadmeRenderer**(*shared*)

Default readme renderer using jinja2.

This is an abstract class that defines a default README renderer for files. From within a template, the macro `render_readme` can be called to annotate a file, which:

- looks for the first plugin that matches this file (that is, the first plugin where *Jinja2ReadmeRenderer.match()* (page 53) returns `True`);
- calls *Jinja2ReadmeRenderer.render()* (page 53), and returns its return value.

To implement such a renderer, in a subclass:

- **do one of:**
  - set *Jinja2ReadmeRenderer.extensions* (page 53) as a list of extensions: the README of any file `foo` is `foo.{ext}`, the README of any directory is `directory/README.{ext}`, where `ext` is one of the extensions listed here.

  - implement *Jinja2ReadmeRenderer.render()* (page 53);

- (optional) implement *Jinja2ReadmeRenderer.match()* (page 53) and *Jinja2ReadmeRenderer.get_readme()* (page 53) if the default implementation does not please you.

extensions: List[str] = []

List of extensions of the READMEs (see *Jinja2ReadmeRenderer* (page 52)).

get_readme(*tree:* Tree *(page 57)*) → Optional[*Tree* (page 57)]

Return readme file for `tree`, or `None` if there is no such README file.

match(*tree:* Tree *(page 57)*) → bool

Return `True` if this plugin can handle the README of the argument.

static render(*tree:* Tree *(page 57)*) → str

Render argument as README.

Return a string to be included when rendering the template. The functions and variables available in the template are described in *renderer.jinja2 — jinja2 renderer* (page 26).

evariste.plugins.renderer.html

Render tree as an HTML (body) page.

class evariste.plugins.renderer.html.**HTMLRenderer**(*shared:* Shared *(page 56)*)

Render tree as an HTML div (without the tags).

The default template name is `tree.html`, and such a default template is found in one of the template directories.

default_setup: Dict[str, str] = {'destfile': 'index.html', 'href_prefix': ''}

Default value for section `self.keyword` in the *setup file* (page 10). It may be overwritten by data provided by user in the *setup file* (page 10). See *Plugin.default_setup and Plugin.global_default_setup* (page 34).

depends: Iterator[str] = ['renderer.html.readme.html', 'renderer.html.file.default']

Iterable of plugins this plugin depends on. When this plugin is enabled, those plugins are enabled as well.

keyword: Union[None, str] = 'renderer.html'

Keyword plugin, used to reference it: it is used to *enable plugins in the setup file* (page 12), to name its section in the *setup file* (page 10), etc.

template: str = 'tree.html'

Default template. This can be overloaded in the setup file. The template is looked for in any of the *templatedirs* (page 26).

evariste.plugins.renderer.html.file

Default HTML file renderer

class evariste.plugins.renderer.html.file.**HtmlFileRenderer**(*shared*)

Default HTML file renderer.

This displays the file name together with the file source.

To write another file renderer, you can:

- define a new *template* (page 52) to use:
- overwrite the default *match()* (page 52) method;
- overwrite the default *render()* (page 52) method.

extension: Optional[str] = 'html'

Extension that is automatically added at the end of the template name when searching them.

keyword: Union[None, str] = 'renderer.html.file.default'

Keyword plugin, used to reference it: it is used to *enable plugins in the setup file* (page 12), to name its section in the *setup file* (page 10), etc.

plugin_type: str = 'renderer.html.file'

Type of the plugin. Plugins of the same type gather some common behaviour.

evariste.plugins.renderer.html.readme

Raw README plugin for html renderer.

class evariste.plugins.renderer.html.readme.**HtmlReadmeRenderer**(*shared*)

Html renderer for readme files, using jinja2 template engine.

It uses the row content of the `html` README.

extensions: List[str] = ['html', 'htm']

List of extensions of the READMEs (see `Jinja2ReadmeRenderer`).

keyword: Union[None, str] = 'renderer.html.readme.html'

Keyword plugin, used to reference it: it is used to *enable plugins in the setup file* (page 12), to name its section in the *setup file* (page 10), etc.

plugin_type: str = 'renderer.html.readme'

Type of the plugin. Plugins of the same type gather some common behaviour.

6.3.8 evariste.plugins.vcs

evariste.plugins.vcs

Access to VCS (git, etc.) versionned files.

Every path processed here is a `pathlib.Path` object.

class evariste.plugins.vcs.**VCS**(*shared*)

Generic class to access to versionned files.

To write a new VCS plugin, one has to subclass this class, and implement every abstract method (see for instance the implementation of `evariste.plugin.vcs.git.Git`).

abstract __contains__(*path: Path*) → bool

Return `True` iff `path` is versionned.

from_repo(*path: Path*) → Path

Return `path`, relative to the repository root.

global_default_setup: Dict[str, Dict[str, str]] = {'setup': {'source': '.'}}

Default values for setup file. See *Plugin.default_setup and Plugin.global_default_setup* (page 34).

last_modified(*path: Path*) → datetime

Return the datetime of last modification.

plugin_type: str = 'vcs'

Type of the plugin. Plugins of the same type gather some common behaviour.

property source: Path

Return an absolute version of source setup option.

abstract walk() → Iterable[Path]

Iterate versionned files, descendants of source (as defined by setup file).

abstract property workdir: Path

Return path of the root of the repository.

evariste.plugins.none

Dummy, do-nothing vcs. Used for tests.

class evariste.plugins.vcs.none.**NoneVCS**(*shared*)

Dummy vcs: Does not access any file.

__contains__(*path: Path*) → bool

Return `True` iff `path` is versionned.

keyword: Union[None, str] = 'vcs.none'

Keyword plugin, used to reference it: it is used to *enable plugins in the setup file* (page 12), to name its section in the *setup file* (page 10), etc.

walk()

Iterate versionned files, descendants of source (as defined by setup file).

property workdir: Path

Return path of the root of the repository.

evariste.plugins.git

Access to git-versionned files.

class evariste.plugins.vcs.git.**Git**(*shared*)

Access git-versionned files

__contains__(*path: Path*) → bool

Return `True` iff `path` is versionned.

keyword: Union[None, str] = 'vcs.git'

Keyword plugin, used to reference it: it is used to *enable plugins in the setup file* (page 12), to name its section in the *setup file* (page 10), etc.

last_modified(*path: Path*) → datetime

Return the datetime of last modification.

walk() → Iterable[Path]

Iterate versionned files, descendants of source (as defined by setup file).

property workdir: Path

Return path of the root of the repository.

6.4 evariste.shared

Share global data between evariste objects.

More information in *Plugin.shared* (page 33).

class evariste.shared.**Shared**(*builder*, ***kwargs*)

Shared data

get_plugin_view(*keyword: str*) → _SharedView

Get this data, from the point of view of a plugin.

Let's define a `shared` object, and its "plugin view":

```
shared = Shared(...)
view = self.get_plugin_view(foo)
```

Now, when both getting and setting data:

- `view.plugin` is equivalent to `shared.plugin[foo]`;
- `view.tree[bar]` is equivalent to `shared.tree[bar][foo]`;
- `view.setup` is equivalent to `shared.setup[foo]`.

get_tree_view(*path: str*) → _SharedView

Get this data, from the point of view of a tree.

Let's define a `shared` object, and its "plugin view":

```
shared = Shared(...)
view = self.get_tree_view(foo)
```

Now, when both getting and setting data:

- `view.tree[bar]` is equivalent to `shared.tree[foo][bar]`.

6.5 evariste.tree

Directory representation and compilation.

A *Tree* (page 57) is an abstract class representing a directory structure (a directory with files and nested directories). Its implementations are:

- *File* (page 60): a file;
- *Directory* (page 60): a directory;
- *Root* (page 61): the root directory being processed.

6.5.1 Tree

class evariste.tree.**Tree**(*path: Path*, *, *parent:* Directory *(page 60)* | *None* = *None*)

A file system tree.

A directory, that contains files and has subdirectories.

Parameters

- **path** (*pathlib.Path*) – Relative path (relative to the root of this tree).
- **parent** (*Optional*[Directory (page 60)]) – Directory containing this file or directory.

basename: pathlib.Path

Name of the tree (path, relative to its *Tree.parent* (page 59)).

config: Union[*utils.DeepDict* (page 62), None]

Computed configuration for this file. See *Per-file and per-directory configuration files* (page 13). Note: This attribute is None until Tree.set_config() has been called.

count(*dirs: bool = False*, *files: bool = True*) → int

Count the number of files or directories in this tree.

property depth: int

Return the depth of the path.

The root has depth 0, and depth of each path is one more than the depth of its parent.

find(*path: str | Path | Tuple[str]*) → *Tree* (page 57) | False

Return the tree object corresponding to path if it exists; False otherwise.

Argument can be:

- a string (str);
- a pathlib.Path object;
- a tuple of strings, as a list of directories and (optional) final file.

format(*string: str*) → str

Format given string, with several variables related to self.

Here are the replacements (with example /home/louis/repo/foo/bar.txt):

- {dirname} (/home/louis/repo/foo): the name of the directory. Note that most of the time, this is useless, since when compiling a file, the working directory is the directory of this file (i.e. {dirname}).
- {filename} (bar.txt): The file name (without directory).
- {fullname} (/home/louis/repo/foo/bar.txt): The file name (with directory).
- {extension} (txt): The extension (without the dot). If the file has several extensions (e.g. foo.tar.gz), this is only the last one gz.
- {basename} (bar): The file name, without directory and extension.

from_fs: pathlib.Path

Absolute path

from_source: pathlib.Path

Path, relative to the *Root* (page 61).

abstract full_depends() → Iterable[Path]

Iterate over all dependencies of this tree (recursively for directories).

is_dir() → bool

Return *True* iff *self* is a directory.

is_file() → bool

Return *True* iff *self* is a file.

static is_root()

Return `True` iff `self` is the root.

local

Same as *Tree.shared* (page 59), but from a tree point of view: see *get_tree_view()* (page 57).

parent: Union[*Tree* (page 57), None]

Parent directory (copied from constructor argument).

prune(*path: Path | str | Tuple[str]*)

Remove a file.

Argument can be either:

- a `pathlib.Path`,
- a `tuple`,
- or a `str` (which would be converted to a `pathlib.Path`.

If called with a non-existing path, does nothing.

property relativename: Path

Return a *relative* name.

- For root, return path relative to file system (or directory of setup file).
- For non-root, return path relative to parent (i.e. basename of path).

report: Union[*plugins.action.Report* (page 50), None]

Once the file has been *compiled* (page 60), the report (compilation log, if any) is saved here.

property root: *Root* (page 61)

Return the root of the tree.

shared: *Shared* (page 56)

Common data shared with every *Tree* (page 57) and *Plugin* (page 46) of this *Builder* (page 43).

vcs: *plugins.vcs.VCS* (page 55)

VCS plugin

walk(*dirs: bool = False*, *files: bool = True*) → Iterable[*Tree* (page 57)]

Iterator over itself.

6.5.2 File

class evariste.tree.**File**(*path: Path*, *, *parent:* Directory *(page 60)* | *None* = *None*)

A file

compile()

Compile file.

depends() → Iterable[Path]

Iterator over dependencies of this file (but not the file itself).

full_depends() → Iterable[Path]

Iterate over all dependencies of this tree (recursively for directories).

last_modified() → datetime

Return the last modified date and time of `self`.

make_archive(*destdir: Path*) → Path

Make an archive of `self` and its dependency.

Steps are:

- build the archive;
- copy it to `destdir`;
- return the path of the archive, relative to `destdir`.

If `self` has no dependencies, consider the file as an archive.

It can be called several times: the archive will be built only once.

6.5.3 Directory

class evariste.tree.**Directory**(**args*, ***kwargs*)

__contains__(*key: str*) → bool

Return `True` if key (a single file name or directory) is in this directory.

__delitem__(*item: str*)

Remove a subfile or subdirectory.

If, after deletion, `self` is an empty directory (and is not root), `self` is remove from its parent.

__getitem__(*key: str*) → *Tree* (page 57)

Return subfile or subdirectory `self.from_fs / key`.

If it does not exist, it is created first.

__iter__() → Iterable[str]

Iterate over subpaths (this function is not recursive).

add_subpath(*sub: List[Path]*)

Add a path to the tree (relative to `self`).

compile()

Compile directory.

full_depends() → Iterable[Path]

Iterate over all dependencies of this tree (recursively for directories).

keys() → Iterable[str]

Iterator over subpaths (as `str` objects).

values() → Iterable[*Tree* (page 57)]

Iterator over subpaths (as *Tree* (page 57) objects).

walk(*dirs: bool = False*, *files: bool = True*) → Iterable[*Tree* (page 57)]

Iterator over files or directories of `self`.

Parameters

- **dirs** (*bool*) – If *False*, do not yield directories.
- **files** (*bool*) – If *False*, do not yield files.

Directories are yielded *before* subfiles and subdirectories.

6.5.4 Root

class evariste.tree.**Root**(*path*, *, *vcs=None*, *shared=None*)

Root object (directory with no parents).

classmethod from_vcs(*repository: VCS (page 55)*) → *Root* (page 61)

Return a directory, fully set.

static is_root() → bool

Return `True` iff `self` is the root.

root_compile()

Recursively compile files..

set_config()

Compute the configuration of each file of the tree.

That is:

- look for the file that configure it (typically `foo.evsconfig` is the configuration for file `foo`),
- load it,
- and complete it using the recursive configuration of parent directories.

6.6 evariste.utils

A rag-bag of utility functions that did not fit anywhere else.

class evariste.utils.**DeepDict**(*depth*, *dictionary=None*)

Dictionary of dictionary of ... of dictionaries.

All the dictionaries (expeted the last one) are `collections.defaultdict` objects.

copy()

Return a copy of `self`.

fill_blanks(*other: dict*)

Recursively copy values of `other` into `self`.

The values are copied only if those of `self` are not defined.

classmethod from_configparser(*config: ConfigParser*) → *DeepDict* (page 62)

Create a *DeepDict* (page 62) object from a `configparser.ConfigParser` object.

get_subkey(*subkey*)

Return the first `self[ANY][subkey]`, when `ANY` is any dictionary key.

evariste.utils.**smart_open**(*filename*, *mode='w'*, *encoding='utf8'*)

Open filename, standard output, or nothing.

Parameters

- **filename** (*str*) – If `filename` is:
  - **"" (the empty string): return a fake file object,** which is empty if file is open for reading, and can be written in if file is open for writing (but content is then discarded);
  - **"-" (a dash):** read from standard input, or write to standard output (depending on mode);
  - any other: open the given file.
- **mode** (*str*) – Same as the `mode` parameter of `open()`.
- **encoding** (*str*) – Same as the `encoding` parameter of `open()`.

evariste.utils.**yesno**(*arg: bool | str | int | None*) → bool

Interpret some (mostly `str`) variable as a boolean.

```
>>> yesno("y")
True
>>> yesno("0")
False
>>> yesno("1")
True
```

(continues on next page)

(continued from previous page)

```
>>> yesno("Yes")
True
>>> yesno("True")
True
>>> yesno("something senseless")
False
>>> yesno(None)
False
```

evariste.utils.**expand_path**(*path*)

Return path where environment variables and user directory have been expanded.

class evariste.utils.**ChangeDir**(*directory*)

Context manager to change and restore current directory.

evariste.utils.**cached_iterator**(*func*)

Like functools.cache(), but for iterators.

That is, the first time the function is run, the returned iterable is stored (as a tuple), and next calls to the function return this tuple.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

EIGHT

JE NE SAIS PAS LE RESTE

C'est que malheureusement on ne se doute pas que le livre le plus précieux du plus savant serait celui où il dirait tout ce qu'il ne sait pas, c'est qu'on ne se doute pas qu'un auteur ne nuit jamais tant à ses lecteurs que quand il dissimule une difficulté. Quand la concurrence, c'est-à-dire l'égoïsme, ne règnera plus dans la science, quand on s'associera pour étudier, au lieu d'envoyer aux Académies des paquets cachetés, on s'empressera de publier ses moindres observations pour peu qu'elles soient nouvelles et on ajoutera : « Je ne sais pas le reste. »

—Évariste Galois, Préface aux « Deux mémoires d'Analyse pure », décembre 1831

PYTHON MODULE INDEX

e

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

P

R

S

T

V

W

Y