rituals Documentation

Release 0.3.1

Jürgen Hermann

2019-01-24

Contents

CHAPTER 1

Overview

"Rituals" is a task library for Invoke that keeps the most common tasks you always need out of your project, and makes them centrally maintained. This leaves your `tasks.py` small and to the point, with only things specific to the project at hand.

For more information, continue with the *User's Manual*.

CHAPTER 2

Important Links

- GitHub Project
- Issue Tracker
- PyPI
- Latest Documentation
- Google Group

CHAPTER 3

Documentation Contents

3.1 User's Manual

3.1.1 Introduction

"Rituals" is a task library for Invoke that keeps the most common tasks you always need out of your project, and makes them centrally maintained. This leaves your `tasks.py` small and to the point, with only things specific to the project at hand.

The following lists the common task implementations that the `rituals.easy` module offers. See *below* on how to integrate them into your `tasks.py`.

- `help` – Default task, when invoked with no task names.
- `clean` – Perform house-cleaning.
- `build` – Build the project.
- `docs` – Build the documentation.
- `test` – Perform standard unittests.
- `check` – Perform source code checks.
- `release.bump` – Bump a development version.
- `release.dist` – Distribute the project.
- `release.prep` – Prepare for a release.
- . . . and *many* more, see `inv -l` for a complete list.

The guiding principle for these tasks is to strictly separate low-level tasks for building and installing (via `setup.py`) from high-level convenience tasks a developer uses (via `invoke`). *Invoke* tasks can use *Setuptools* ones as building blocks, but never the other way 'round – this avoids any bootstrapping headaches during package installations.

Use `inv -h ‹task›` as usual to get details on the options of these tasks. The *Tasks Reference* explains them in more detail. Look at the modules in rituals.acts if you want to know every nuance of what these tasks do.

Note:

The easiest way to get a working project using `rituals` is the py-generic-project cookiecutter archetype, which is tightly integrated with the tasks defined here.

That way you have a working project skeleton within minutes that is fully equipped, with all aspects of building, testing, quality checks, continuous integration, documentation, and releasing covered.

3.1.2 Adding Rituals to Your Project

First of all, include `rituals` as a dependency in your `dev-requirements.txt` or a similar file, to get a release from PyPI. To refer to the current GitHub `master` branch instead, use a `pip` requirement like this:

```
-e git+https://github.com/jhermann/rituals.git#egg=rituals
```

Then at the start of your `tasks.py`, use the following statement to define *all* tasks that are considered standard:

```
from rituals.easy import *
```

This works by defining the `namespace` identifier containing Ritual's default tasks. Note that it also defines Invoke's `Collection` and `task` identifiers, and some other common helpers assembled in *`rituals.easy`*. Rituals' own tasks.py can serve as an example.

Of course, you may also do more selective imports, or build your own *Invoke* namespaces with the specific tasks you need.

Warning: These tasks expect an importable `setup.py` that defines a `project` dict with the setup parameters, see rudiments and py-generic-project for examples. The needed changes are minimal:

```
project = dict(  # this would usually be a setup(...) call
    name='...',
    ...
)
if __name__ == '__main__':
    setup(**project)
```

3.1.3 Task Namespaces

3.1.3.1 The Root Namespace

The tasks useful for any (Python) project are organized in a root namespace. When you use the `from rituals.easy import *` statement, that also imports this root namespace. By convention of *Invoke*, when the identifier `namespace` is defined, that one is taken instead of constructing one automatically from all defined tasks.

It contains some fundamentals like `clean`, and nested namespaces handling specific topics. Examples of nested namespaces are `test`, `check`, `docs`, and `release`. See *Tasks Reference* for a complete list.

The root namespace has `help` as the default task, and most nested namespaces also have a default with the most commonly performed action. These default tasks are automatically aliased to the name of the namespace, so for example `docs.sphinx` can also be called as `docs`.

3.1.3.2 Adding Local Task Definitions

Having an explicit root namespace means that within `tasks.py`, you need to register your own tasks using its `add_task` method, if you want them to be available as top-level names:

```
@task
def my_own_task(ctx):
    """Something project-specific."""
    ...

namespace.add_task(my_own_task)
```

Rituals' own tasks.py uses this to add some local tasks.

Another strategy is to add them in bulk, so when you write a new task you cannot forget to make it visible:

```
# Register local tasks in root namespace
from invoke import Task
for _task in globals().values():
    if isinstance(_task, Task) and _task.body.__module__ == __name__:
        namespace.add_task(_task)
```

Add the above snippet to the *end* of your `tasks.py`, and every *local* task definition gets added to the root namespace.

3.1.3.3 Constructing Your Own Namespace

When you want to have more control, you can exclude the `namespace` identifier from the import and instead define your own. This example taken from the tasks.py of py-generic-project shows how it's done:

```
from rituals.easy import task, Collection
from rituals.acts.documentation import namespace as _docs

...

namespace = Collection.from_module(sys.modules[__name__], name='')
namespace.add_collection(_docs)
```

Note that the `name=''` makes this a root namespace. If you need to be even more selective, import individual tasks from modules in `rituals.acts` and add them to your namespaces.

3.1.4 How-Tos

3.1.4.1 Change default project layout

By default, sources are expected in `src/‹packagename›` and tests in `src/tests`.

You can change this by calling one of the following functions, directly after the import from `rituals.invoke_tasks`.

- `config.set_maven_layout()` – Changes locations to `src/main/python/‹packagename›` and `src/test/python`.

- `config.set_flat_layout()` – Changes locations to ‹ `packagename` › and `tests`.

3.1.4.2 Change default project configuration

If you want to override the configuration defaults of various tasks, without using environment variables, add an `invoke.yaml` file in the same directory where your `tasks.py` is located – usually the project root directory.

This example makes *Sphinx* (as called by the default `docs` task) place generated files in the top-level `build` directory instead of a sub-directory in `docs`.

Listing 1: invoke.yaml

```
rituals:
    docs:
        build: ../build/_html
```

See *Configuration Reference* for a list of possible options.

3.2 Tasks Reference

Please make sure you also read the section on *Task Namespaces*.

3.2.1 Fundamental Tasks

3.2.1.1 Getting Help

`help` is the default task in the root namespace, so that just calling `inv` gives a reasonable response. It combines the global help (`inv -h`) and the task listing (`inv -l`).

3.2.1.2 Project Cleanup

The `clean` task gets rid of various groups of generated files, selectable by options. These options are:

```
-a, --all         The same as --backups --bytecode --dist --docs
-b, --backups     Also clean '*~' files etc.
-d, --docs        Also clean the documentation build area
-e, --extra       Any extra patterns, space-separated and possibly quoted
-i, --dist        Also clean the 'dist' dir
-t, --tox         Include '.tox' directory
-v, --venv        Include an existing virtualenv (in '.' or in '.venv')
-y, --bytecode    Also clean '.pyc', '.pyo', and package metadata
```

Note that `--all` is selective and only cleans out ‘cheap’ files; it especially excludes a local virtualenv that carries state (installed packages) you might not have recorded safely in requirements files yet, and the tree generated by `tox` that can take a while to reproduce.

The `--extra` options allows you to add any custom glob patterns to clean out.

3.2.1.3 Building the Project

Invoking `build` just delegates to `setup.py` right now. In the future, automatic detection of other project components like a Maven POM or Javascript build files might trigger additional build tools.

You can also include the `docs` task by adding the `--docs` option.

3.2.1.4 Freezing Requirements

Calling `freeze` writes the frozen requirements as found in the current environment into the file `frozen-requirements.txt`, by calling `pip freeze`.

3.2.2 Executing Tests

The `test.pytest` and `test.tox` tasks call the related testing tools with appropriate parameters. Coverage configuration is added to the `py.test` call, and if you pass the `--coverage` option, the generated report is loaded into your web browser. In case the `py.test` command isn't available, calling the test runner is delegated to `setup.py`.

For `test.tox`, `PATH` is extended according to the directories in the `rituals.snakepits` configuration value, which defaults to `/opt/pyenv/bin:/opt/pyrun/bin`. That way, you can provide the *Python* interpreter versions to run multi-environment tests locally.

3.2.3 Documentation Tasks

3.2.3.1 Building Sphinx Documentation

Rituals provides automatic process management of a `sphinx-autobuild` daemon, which means you easily get a live-reload preview in your browser. To start the build watchdog, use `inv docs -w -b`. The `-b` means to open a new browser tab, after the server process is ready. To kill the server, call the `inv docs -k` command. You can check on the status of a running daemon with `inv docs -s`.

Note that sometimes you have to manually trigger a full rebuild via `inv docs --clean`, especially when you make structural changes (e.g. adding new chapters to the main toc-tree). Your browser will change the view to an empty canvas, just initiate a reload (`Ctrl-R`) when the build is done. Typically this is needed when the sidebar TOC is out of sync, which happens due to the optimizations in `sphinx-autobuild` that make it so responsive.

3.2.3.2 Publishing Documentation

To upload documentation to either PyPI or a WebDAV server (like *Artifactory*), you can use the `docs.upload` tasks after using `docs.sphinx`. By default, the documentation hosting of PyPI is used.

To enable a local documentation server, set the following environment variables (e.g. in your `~/.bashrc`):

```
export INVOKE_RITUALS_DOCS_UPLOAD_METHOD=webdav
export INVOKE_RITUALS_DOCS_UPLOAD_TARGETS_WEBDAV_URL='http://repo.example.com/
↪artifactory/wwwdata-local/python/{name}/{version}/{name}-{version}.zip;kind=docs'
```

This example shows a configuration for an Artifactory server, and as you can see the `name` and `version` of the project can be used to generate appropriate URLs. The version defaults to `latest`, unless you provide a specific version number via the `--release` option. To use the upload in a browser, add `!/index.html` to the URL of the ZIP file, and make sure the configuration of your Artifactory installation correctly handles image and other relevant MIME types.

Note that you can override the default upload method using the `target` option, i.e. adding `--target pypi` to your task invocation will upload the docs to PyPI no matter what.

Note: The WebDAV upload is tested with Artifactory 4.x, if you encounter problems with other repository servers, open a ticket so support for them can be added.

3.2.4 Release Workflow

3.2.4.1 Managing Development Versions

For intermediate test releases, you can automatically generate a dev version compliant to PEP-440. The `release.bump` task generates this version, and rewrites the `tag_build` value in `setup_cfg`. For this to work, the config file must already exists, and contain a line with an existing `tag_build` setting, like this:

```
[egg_info]
tag_build = .dev
tag_date = false
```

The created version strives to uniquely describe the code being packaged, so it can get quite lengthy. The "worst case scenario" when it comes to length looks like this: `1.2.2.dev4+1.2.1.g07c5047.20170309t1247.ci.42`.

Let's dissect this:

- `1.2.2` is the next-release version as reported by `setup.py`.
- `.dev4` means we are 4 commits beyond the last release version.
- `1.2.1` is that last release version, found via checking the annotated `git` tags.
- `g07c5047` is the `git` commit hash of the `HEAD` ref.
- Having the `20170309t1247` timestamp means the working directory at the time of the task execution was dirty (had uncommitted changes).
- `ci.42` is appended when the environment contains a `BUILD_ID` variable (in this case, set to `42`).

Use the `--pypi` option to prevent creation of the local part of the version info (anything after the `+`). This allows you to push development versions to PyPI for open beta release testing. In those cases, you should commit `setup.cfg` with the specific `tag_build` setting, and then `git tag` that commit.

Adding `-v` for verbosity shows a few more details of assembling the version information. Right now, only `git` is really supported regarding SCM metadata. Anything else will give you `Unsupported SCM` warnings when using this and some other tasks.

3.2.4.2 Preparing a Release

`release.prep` performs QA checks, and switches to non-dev versioning.

TODO

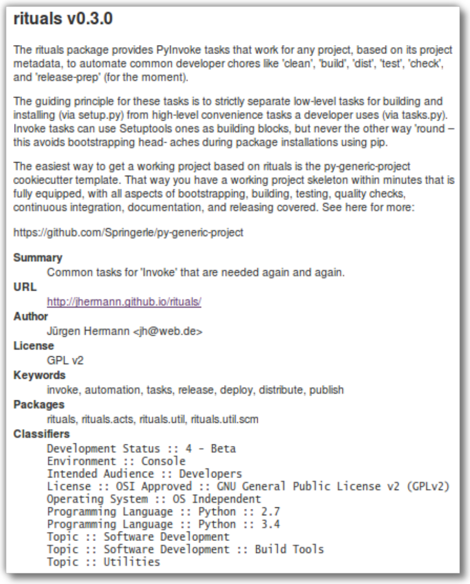
3.2.4.3 Building a PEX Distribution

TODO

3.2.5 Continuous Integration

3.2.5.1 Jenkins

Note that tasks related to *Jenkins* are only available by default when the environment variable `JENKINS_URL` is defined.



rituals v0.3.0

The rituals package provides PyInvoke tasks that work for any project, based on its project metadata, to automate common developer chores like 'clean', 'build', 'dist', 'test', 'check', and 'release-prep' (for the moment).

The guiding principle for these tasks is to strictly separate low-level tasks for building and installing (via setup.py) from high-level convenience tasks a developer uses (via tasks.py). Invoke tasks can use Setuptools ones as building blocks, but never the other way 'round – this avoids bootstrapping head- aches during package installations using pip.

The easiest way to get a working project based on rituals is the py-generic-project cookiecutter template. That way you have a working project skeleton within minutes that is fully equipped, with all aspects of bootstrapping, building, testing, quality checks, continuous integration, documentation, and releasing covered. See here for more:

https://github.com/Springerle/py-generic-project

Summary
Common tasks for 'Invoke' that are needed again and again.
URL
http://jhermann.github.io/rituals/
Author
Jürgen Hermann <jh@web.de>
License
GPL v2
Keywords
invoke, automation, tasks, release, deploy, distribute, publish
Packages
rituals, rituals.acts, rituals.util, rituals.util.scm
Classifiers

```
Development Status :: 4 - Beta
Environment :: Console
Intended Audience :: Developers
License :: OSI Approved :: GNU General Public License v2 (GPLv2)
Operating System :: OS Independent
Programming Language :: Python :: 2.7
Programming Language :: Python :: 3.4
Topic :: Software Development
Topic :: Software Development :: Build Tools
Topic :: Utilities
```

The `jenkins.description` task creates a text file (by default `build/project.html`) that can be used via the Jenkins Description Setter plugin to dynamically fill the Jenkins job description from already available metadata. The resulting description looks like the example on the right.

3.3 Configuration Reference

Please read Invoke's Configuration Guide on the concepts and basic mechanisms of its hierarchy of configuration files, environment variables, task namespaces and CLI flags. This reference guide lists the configuration options specific to tasks provided by *Rituals*.

Note: In the following tables of configurations settings, the root namespace of 'rituals' is implied, so to access them in a task you'd use `ctx.rituals.‹ name ›`, and `INVOKE_RITUALS_‹ NAME ›` to define an environment variable.

3.3.1 General Options

To make Python versions available that are not part of the host's default installation, `rituals.snakepits` is used, e.g. when performing multi-environment testing. The default is `/opt/pyenv/bin:/opt/pyrun/bin`.

Name	Description
snakepits	Lookup path for Python interpreters

3.3.2 Options for 'test'

If one of the directories in `rituals.snakepits` exists, it's added to the `PATH` of `tox`.

3.3.3 Options for 'docs'

The defaults for the `docs` task should almost always fit, but if you need to change them, you can.

Name	Description
docs.sources	Documentation source folder (`docs`)
docs.build	Build area within the source folder (`_build`)
docs.watchdog.host	IP to bind `sphinx-autobuild` to (`127.0.0.1`)
docs.watchdog.port	Port to bind `sphinx-autobuild` to (`8840`)

3.3.4 Options for 'release'

When `release.prep` changes the project configuration for a release and then tags the resulting changeset, the values from the following table are used for messages and names.

Name	Description
release.commit.message	Message used (`:package: Release v{version}`)
release.tag.name	Release tag (`v{version}`)
release.tag.message	Tag annotation (`Release v{version}`)

The `release.pex` task has an `--upload` option to upload the created archive to a WebDAV repository, e.g. a local Artifactory server or to Bintray. The best way to make this usable in each of your projects is to insert the base URL of your Python repository into your shell environment:

```
export INVOKE_RITUALS_RELEASE_UPLOAD_BASE_URL=\
"http://repo.example.com/artifactory/pypi-releases-local/"
```

Name	Description
release.upload.base_url	WebDAV server end-point
release.upload.path	WebDAV path (`{name}/{version}/{filename}`)

The following settings are used when building self-contained releases that integrate eGenix PyRun.

Name	Description
pyrun.version	The version of *PyRun* to use (e.g. `2.1.0`)
pyrun.python	The version of *Python* to use (`2.6`, `2.7`, or `3.4`)
pyrun.ucs	Unicode code points size (`ucs2` or `ucs4`)
pyrun.platform	The platform ID (e.g. `linux-x86_64`, `macosx-10.5-x86_64`)
pyrun.base_url	Download location base URL pattern
pyrun.archive	Download location file name pattern

The `rituals.pyrun.base_url` value can be a local `http[s]` URL of an Artifactory repository or some similar webserver, or else a `file://` URL of a file system cache. Note that you should keep the unchanged name of the original download location, i.e. do not change `rituals.pyrun.archive`. The location patterns can contain the `pyrun` settings as placeholders, e.g. `{version}`.

This sets a local download cache:

```
export INVOKE_RITUALS_PYRUN_BASE_URL="file://$HOME/Downloads"
```

You have to download the *PyRun* releases you plan to use to that directory, using your browser or `curl`.

3.3.5 Options for 'devpi'

When you call the `devpi.refresh` task without any option, the value of `rituals.devpi.requirements` is the name of the file parsed for the list of packages to refresh in the active `devpi` server. It defaults to `dev-requirements.txt`.

Name	Description
devpi.requirements	Name of requirements file to use for refreshing

3.4 Complete API Reference

The following is a complete API reference generated from source.

3.4.1 rituals package

Common tasks for 'Invoke' that are needed again and again.

The `rituals` package provides PyInvoke tasks that work for any project, based on its project metadata, to automate common developer chores like 'clean', 'build', 'dist', 'test', 'check', and 'release-prep' (for the moment).

The guiding principle for these tasks is to strictly separate low-level tasks for building and installing (via `setup.py`) from high-level convenience tasks a developer uses (via `tasks.py`). Invoke tasks can use Setuptools ones as building blocks, but never the other way 'round – this avoids bootstrapping head- aches during package installations using `pip`.

The easiest way to get a working project based on `rituals` is the `py-generic-project` cookiecutter template. That way you have a working project skeleton within minutes that is fully equipped, with all aspects of bootstrapping, building, testing, quality checks, continuous integration, documentation, and releasing covered. See here for more:

> https://github.com/Springerle/py-generic-project

3.4.1.1 Subpackages

rituals.acts package

rituals.acts – Task building blocks.

Submodules

rituals.acts.basic module

Basic tasks.

rituals.acts.devpi module

'devpi' tasks.

`rituals.acts.devpi.`**`get_devpi_url`**(*ctx*)
  Get currently used 'devpi' base URL.

rituals.acts.documentation module

'docs' tasks.

`class` `rituals.acts.documentation.`**`DocsUploader`**(*ctx*, *cfg*, *target*)
  Bases: `object`

  Helper to perform an upload of pre-built docs.

  `upload`(*docs_base*, *release*)
    Upload docs in `docs_base` to the target of this uploader.

`rituals.acts.documentation.`**`get_pypi_auth`**(*configfile=u'~/.pypirc'*)
  Read auth from pip config.

`rituals.acts.documentation.`**`watchdogctl`**(*ctx*, *kill=False*, *verbose=True*)
  Control / check a running Sphinx autobuild process.

rituals.acts.github module

GitHub automation.

rituals.acts.inspection module

'docs' tasks.

rituals.acts.jenkins module

Tasks specific to Jenkins.

rituals.acts.pkgdeb module

'deb' tasks.

rituals.acts.releasing module

Release tasks.

`rituals.acts.releasing.`**`get_egg_info`**`(`*cfg*, *verbose=False*`)`
Call 'setup egg_info' and return the parsed meta-data.

rituals.acts.testing module

Testing tasks.

rituals.util package

rituals.util – Helper modules.

`rituals.util.`**`add_dir2pypath`**`(`*path*`)`
Add given directory to PYTHONPATH, e.g. for pylint.

`rituals.util.`**`search_file_upwards`**`(`*name*, *base=None*`)`
Search for a file named *name* from cwd or given directory to root. Return None if nothing's found.

Subpackages

rituals.util.scm package

rituals.util.scm – Source Code Management support.

`rituals.util.scm.`**`auto_detect`**`(`*workdir*`)`
Return string signifying the SCM used in the given directory.

Currently, 'git' is supported. Anything else returns 'unknown'.

`rituals.util.scm.`**`provider`**`(`*workdir*, *commit=True*, ***kwargs*`)`
Factory for the correct SCM provider in *workdir*.

Submodules

rituals.util.scm.base module

Provider base class.

`class` `rituals.util.scm.base.`**`ProviderBase`**`(`*workdir*, *commit=True*, ***kwargs*`)`
Bases: `object`

Base class for SCM providers.

`run``(`*cmd*, **args*, ***kwargs*`)`
Run a command.

run_elective(*cmd*, **args*, ***kwargs*)
Run a command, or just echo it, depending on *commit*.

rituals.util.scm.git module

git SCM provider.

class rituals.util.scm.git.**GitProvider**(*workdir*, *commit=True*, ***kwargs*)
Bases: *rituals.util.scm.base.ProviderBase*

git SCM provider.

Expects a working *git* executable in the path, having a reasonably current version.

add_file(*filename*)
Stage a file for committing.

commit(*message*)
Commit pending changes.

key = u'git'

pep440_dev_version(*verbose=False*, *non_local=False*)
Return a PEP-440 dev version appendix to the main version number.

Result is `None` if the workdir is in a release-ready state (i.e. clean and properly tagged).

tag(*label*, *message=None*)
Tag the current workdir state.

workdir_is_clean(*quiet=False*)
Check for uncommitted changes, return *True* if everything is clean.

Inspired by http://stackoverflow.com/questions/3878624/.

rituals.util.scm.null module

Provider for unknown SCM systems.

class rituals.util.scm.null.**NullProvider**(*workdir*, *commit=True*, ***kwargs*)
Bases: *rituals.util.scm.base.ProviderBase*

Stub provider for unknown SCM systems.

This implements the provider interface, mostly emitting warnings.

add_file(*filename*)
Stage a file for committing, or commit it directly (depending on the SCM).

commit(*message*)
Commit pending changes.

key = u'unknown'

pep440_dev_version(*verbose=False*, *non_local=False*)
Return a PEP-440 dev version appendix to the main version number.

tag(*label*, *message=None*)
Tag the current workdir state.

workdir_is_clean(*quiet=False*)
Check for uncommitted changes, return *True* if everything is clean.

Submodules

rituals.util.antglob module

Recursive globbing with ant-style syntax.

class `rituals.util.antglob.`**`FileSet`**(*root*, *patterns*)

Bases: `object`

Ant-style file and directory matching.

Produces an iterator of all of the files that match the provided patterns. Note that directory matches must end with a slash, and if they're exclusions, they won't be scanned (which prunes anything in that directory that would otherwise match).

Directory specifiers: ** matches zero or more directories. / path separator.

File specifiers:

- glob style wildcard.

[chars] inclusive character sets. [^chars] exclusive character sets.

Examples

/*.py recursively match all python files. foo//*.py recursively match all python files in the 'foo' directory. *.py match all the python files in the current directory.* */.txt match all the text files in top-level directories. foo/**/* all files under directory 'foo'. */ top-level directories. foo/ the directory 'foo' itself.* */foo/ any directory named 'foo'. **/.* hidden files. **/.*/ hidden directories.

`included`(*path*, *is_dir=False*)

Check patterns in order, last match that includes or excludes *path* wins. Return *None* on undecided.

`walk`(***kwargs*)

Like *os.walk* and taking the same keyword arguments, but generating paths relative to the root.

Starts in the fileset's root and filters based on its patterns. If `with_root=True` is passed in, the generated paths include the root path.

`rituals.util.antglob.`**`includes`**(*pattern*)

A single inclusive glob pattern.

`rituals.util.antglob.`**`excludes`**(*pattern*)

A single exclusive glob pattern.

rituals.util.filesys module

File system helpers.

`rituals.util.filesys.`**`pretty_path`**(*path*, *_home_re=<_sre.SRE_Pattern object>*)

Prettify path for humans, and make it Unicode.

`rituals.util.filesys.`**`pushd`**(**args*, ***kwds*)

A context that enters a given directory and restores the old state on exit.

The original directory is returned as the context variable.

`rituals.util.filesys.`**`url_as_file`**(**args*, ***kwds*)

Context manager that GETs a given *url* and provides it as a local file.

The file is in a closed state upon entering the context, and removed when leaving it, if still there.

To give the file name a specific extension, use *ext*; the extension can optionally include a separating dot, otherwise it will be added.

Parameters

- **`url`** (`str`) – URL to retrieve.
- **`ext`** (`str, optional`) – Extension for the generated filename.

Yields *str* – The path to a temporary file with the content of the URL.

Raises `requests.RequestException` – Base exception of `requests`, see its docs for more detailed ones.

Example

```
>>> import io, re, json
>>> with url_as_file('https://api.github.com/meta', ext='json') as meta:
...     meta, json.load(io.open(meta, encoding='ascii'))['hooks']
(u'/tmp/www-api.github.com-Ba5OhD.json', [u'192.30.252.0/22'])
```

rituals.util.notify module

Log notification messages to console.

`rituals.util.notify.`**`banner`**(*msg*)

Emit a banner just like Invoke's *run(…, echo=True)*.

`rituals.util.notify.`**`error`**(*msg*)

Emit an error message to stderr.

`rituals.util.notify.`**`failure`**(*msg*)

Emit a fatal message and exit.

`rituals.util.notify.`**`info`**(*msg*)

Emit a normal message.

`rituals.util.notify.`**`warning`**(*msg*)

Emit a warning message.

rituals.util.shell module

Shell command calls.

`rituals.util.shell.`**`capture`**(*cmd*, ***kw*)

Run a command and return its stripped captured output.

`rituals.util.shell.`**`run`**(*cmd*, ***kw*)

Run a command and flush its output.

rituals.util.which module

Find the full path to commands.

which(command, path=None, verbose=0, exts=None) Return the full path to the first match of the given command on the path.

whichall(command, path=None, verbose=0, exts=None) Return a list of full paths to all matches of the given command on the path.

whichgen(command, path=None, verbose=0, exts=None) Return a generator which will yield full paths to all matches of the given command on the path.

By default the PATH environment variable is searched (as well as, on Windows, the AppPaths key in the registry), but a specific 'path' list to search may be specified as well. On Windows, the PATHEXT environment variable is applied as appropriate.

If "verbose" is true then a tuple of the form (<fullpath>, <matched-where-description>)

is returned for each match. The latter element is a textual description of where the match was found. For example:

from PATH element 0 from HKLMSOFTWARE. . . perl.exe

exception `rituals.util.which.`**`WhichError`**

Bases: `exceptions.Exception`

Executable not found by *which* module.

`rituals.util.which.`**`which`**(*command*, *path=None*, *verbose=0*, *exts=None*)

Return the full path to the first match of the given command on the path.

"command" is a the name of the executable to search for. "path" is an optional alternate path list to search. The default it

to use the PATH environment variable.

"verbose", if true, will cause a 2-tuple to be returned. The second element is a textual description of where the match was found.

"exts" optionally allows one to specify a list of extensions to use instead of the standard list for this system. This can effectively be used as an optimization to, for example, avoid stat's of "foo.vbs" when searching for "foo" and you know it is not a VisualBasic script but ".vbs" is on PATHEXT. This option is only supported on Windows.

If no match is found for the command, a WhichError is raised.

`rituals.util.which.`**`whichall`**(*command*, *path=None*, *verbose=0*, *exts=None*)

Return a list of full paths to all matches of the given command on the path.

"command" is a the name of the executable to search for. "path" is an optional alternate path list to search. The default it

to use the PATH environment variable.

"verbose", if true, will cause a 2-tuple to be returned for each match. The second element is a textual description of where the match was found.

"exts" optionally allows one to specify a list of extensions to use instead of the standard list for this system. This can effectively be used as an optimization to, for example, avoid stat's of "foo.vbs" when searching for "foo" and you know it is not a VisualBasic script but ".vbs" is on PATHEXT. This option is only supported on Windows.

`rituals.util.which.`**`whichgen`**`(`*command*, *path=None*, *verbose=0*, *exts=None*`)`

Return a generator of full paths to the given command.

"command" is a the name of the executable to search for. "path" is an optional alternate path list to search. The default it

to use the PATH environment variable.

"verbose", if true, will cause a 2-tuple to be returned for each match. The second element is a textual description of where the match was found.

"exts" optionally allows one to specify a list of extensions to use instead of the standard list for this system. This can effectively be used as an optimization to, for example, avoid stat's of "foo.vbs" when searching for "foo" and you know it is not a VisualBasic script but ".vbs" is on PATHEXT. This option is only supported on Windows.

This method returns a generator which yields either full paths to the given command or, if verbose, tuples of the form (<path to command>, <where path found>).

3.4.1.2 Submodules

3.4.1.3 rituals.config module

Project configuration and layout.

`rituals.config.`**`get_project_root`**`()`

Determine location of *tasks.py*.

`rituals.config.`**`load`**`()`

Load and return configuration as a `Bunch`.

Values are based on `DEFAULTS`, and metadata from `setup.py`.

`rituals.config.`**`set_flat_layout`**`()`

Switch default project layout to everything top-level.

`rituals.config.`**`set_maven_layout`**`()`

Switch default project layout to Maven-like.

3.4.1.4 rituals.easy module

Default namespace for convenient wildcard import in task definition modules.

`class` `rituals.easy.`**`Collection`**`(`**args*, ***kwargs*`)`

Bases: `object`

A collection of executable tasks. See /concepts/namespaces.

New in version 1.0.

`add_collection``(`*coll*, *name=None*`)`

Add .*Collection* `coll` as a sub-collection of this one.

Parameters

- **`coll`** – The .*Collection* to add.
- **`name`** (*`str`*) – The name to attach the collection as. Defaults to the collection's own internal name.

New in version 1.0.

add_task(*task*, *name=None*, *aliases=None*, *default=None*)
Add *.Task* `task` to this collection.

Parameters

- **task** – The *.Task* object to add to this collection.
- **name** – Optional string name to bind to (overrides the task's own self-defined `name` attribute and/or any Python identifier (i.e. `.func_name`.)
- **aliases** – Optional iterable of additional names to bind the task as, on top of the primary name. These will be used in addition to any aliases the task itself declares internally.
- **default** – Whether this task should be the collection default.

New in version 1.0.

configuration(*taskpath=None*)
Obtain merged configuration values from collection & children.

Parameters **taskpath** – (Optional) Task name/path, identical to that used for *~.Collection.__getitem__* (e.g. may be dotted for nested tasks, etc.) Used to decide which path to follow in the collection tree when merging config values.

Returns A *dict* containing configuration values.

New in version 1.0.

configure(*options*)
(Recursively) merge `options` into the current *.configuration*.

Options configured this way will be available to all tasks. It is recommended to use unique keys to avoid potential clashes with other config options

For example, if you were configuring a Sphinx docs build target directory, it's better to use a key like `'sphinx.target'` than simply `'target'`.

Parameters **options** – An object implementing the dictionary protocol.

Returns `None`.

New in version 1.0.

classmethod from_module(*module*, *name=None*, *config=None*, *loaded_from=None*, *auto_dash_names=None*)
Return a new *.Collection* created from `module`.

Inspects `module` for any *.Task* instances and adds them to a new *.Collection*, returning it. If any explicit namespace collections exist (named `ns` or `namespace`) a copy of that collection object is preferentially loaded instead.

When the implicit/default collection is generated, it will be named after the module's `__name__` attribute, or its last dotted section if it's a submodule. (I.e. it should usually map to the actual `.py` filename.)

Explicitly given collections will only be given that module-derived name if they don't already have a valid `.name` attribute.

If the module has a docstring (`__doc__`) it is copied onto the resulting *.Collection* (and used for display in help, list etc output.)

Parameters

- **name** (*str*) – A string, which if given will override any automatically derived collection name (or name set on the module's root namespace, if it has one.)

- **config** (*dict*) – Used to set config options on the newly created *.Collection* before returning it (saving you a call to *.configure*.)

  If the imported module had a root namespace object, `config` is merged on top of it (i.e. overriding any conflicts.)
- **loaded_from** (*str*) – Identical to the same-named kwarg from the regular class constructor - should be the path where the module was found.
- **auto_dash_names** (*bool*) – Identical to the same-named kwarg from the regular class constructor - determines whether emitted names are auto-dashed.

New in version 1.0.

serialized()
Return an appropriate-for-serialization version of this object.

See the documentation for *.Program* and its `json` task listing format; this method is the driver for that functionality.

New in version 1.0.

subcollection_from_path(*path*)
Given a `path` to a subcollection, return that subcollection.

New in version 1.0.

subtask_name(*collection_name*, *task_name*)

task_names
Return all task identifiers for this collection as a one-level dict.

Specifically, a dict with the primary/"real" task names as the key, and any aliases as a list value.

It basically collapses the namespace tree into a single easily-scannable collection of invocation strings, and is thus suitable for things like flat-style task listings or transformation into parser contexts.

New in version 1.0.

task_with_config(*name*)
Return task named `name` plus its configuration dict.

E.g. in a deeply nested tree, this method returns the *.Task*, and a configuration dict created by merging that of this *.Collection* and any nested *Collections <.Collection>*, up through the one actually holding the *.Task*.

See *~.Collection.__getitem__* for semantics of the `name` argument.

Returns Two-tuple of (*.Task*, *dict*).

New in version 1.0.

to_contexts()
Returns all contained tasks and subtasks as a list of parser contexts.

New in version 1.0.

transform(*name*)
Transform `name` with the configured auto-dashes behavior.

If the collection's `auto_dash_names` attribute is `True` (default), all non leading/trailing underscores are turned into dashes. (Leading/trailing underscores tend to get stripped elsewhere in the stack.)

If it is `False`, the inverse is applied - all dashes are turned into underscores.

New in version 1.0.

`rituals.easy.`**`task`**`(`**args*, ***kwargs*`)`

Marks wrapped callable object as a valid Invoke task.

May be called without any parentheses if no extra options need to be specified. Otherwise, the following keyword arguments are allowed in the parenthese'd form:

- `name`: Default name to use when binding to a *.Collection*. Useful for avoiding Python namespace issues (i.e. when the desired CLI level name can't or shouldn't be used as the Python level name.)
- `aliases`: Specify one or more aliases for this task, allowing it to be invoked as multiple different names. For example, a task named `mytask` with a simple `@task` wrapper may only be invoked as `"mytask"`. Changing the decorator to be `@task(aliases=['myothertask'])` allows invocation as `"mytask"` *or* `"myothertask"`.
- `positional`: Iterable overriding the parser's automatic "args with no default value are considered positional" behavior. If a list of arg names, no args besides those named in this iterable will be considered positional. (This means that an empty list will force all arguments to be given as explicit flags.)
- `optional`: Iterable of argument names, declaring those args to have optional values. Such arguments may be given as value-taking options (e.g. `--my-arg=myvalue`, wherein the task is given `"myvalue"`) or as Boolean flags (`--my-arg`, resulting in `True`).
- `iterable`: Iterable of argument names, declaring them to build iterable values.
- `incrementable`: Iterable of argument names, declaring them to increment their values.
- `default`: Boolean option specifying whether this task should be its collection's default task (i.e. called if the collection's own name is given.)
- `auto_shortflags`: Whether or not to automatically create short flags from task options; defaults to True.
- `help`: Dict mapping argument names to their help strings. Will be displayed in `--help` output.
- `pre`, `post`: Lists of task objects to execute prior to, or after, the wrapped task whenever it is executed.
- `autoprint`: Boolean determining whether to automatically print this task's return value to standard output when invoked directly via the CLI. Defaults to False.
- `klass`: Class to instantiate/return. Defaults to *.Task*.

If any non-keyword arguments are given, they are taken as the value of the `pre` kwarg for convenience's sake. (It is an error to give both `*args` and `pre` at the same time.)

New in version 1.0.

Changed in version 1.1: Added the `klass` keyword argument.

`rituals.easy.`**`pushd`**`(`**args*, ***kwds*`)`

A context that enters a given directory and restores the old state on exit.

The original directory is returned as the context variable.

`rituals.easy.`**`fail`**`(`*message*, *exitcode=1*`)`

Exit with error code and message.

3.5 Contribution Guidelines

3.5.1 Overview

Contributing to this project is easy, and reporting an issue or adding to the documentation also improves things for every user. You don't need to be a developer to contribute.

3.5.1.1 Reporting issues

Please use the *GitHub issue tracker*, and describe your problem so that it can be easily reproduced. Providing relevant version information on the project itself and your environment helps with that.

3.5.1.2 Improving documentation

The easiest way to provide examples or related documentation that helps other users is the *GitHub wiki*.

If you are comfortable with the Sphinx documentation tool, you can also prepare a pull request with changes to the core documentation. GitHub's built-in text editor makes this especially easy, when you choose the *"Create a new branch for this commit and start a pull request"* option on saving. Small fixes for typos and the like are a matter of minutes when using that tool.

3.5.1.3 Code contributions

Here's a quick guide to improve the code:

1. Fork the repo, and clone the fork to your machine.
2. Add your improvements, the technical details are further below.
3. Run the tests and make sure they're passing (`invoke test`).
4. Check for violations of code conventions (`invoke check`).
5. Make sure the documentation builds without errors (`invoke build --docs`).
6. Push to your fork and submit a pull request.

Please be patient while waiting for a review. Life & work tend to interfere.

3.5.2 Details on contributing code

This project is written in Python, and the documentation is generated using Sphinx. setuptools and Invoke are used to build and manage the project. Tests are written and executed using pytest and tox.

3.5.2.1 Set up a working development environment

To set up a working directory from your own fork, follow these steps, but replace the repository `https` URLs with SSH ones that point to your fork.

For that to work on Debian type systems, you need the `git`, `python`, and `python-virtualenv` packages installed. Other distributions are similar.

3.5.2.2 Add your changes to a feature branch

For any cohesive set of changes, create a *new* branch based on the current upstream `master`, with a name reflecting the essence of your improvement.

```
git branch "name-for-my-fixes" origin/master
git checkout "name-for-my-fixes"
... make changes...
invoke ci # check output for broken tests, or PEP8 violations and the like
... commit changes...
git push origin "name-for-my-fixes"
```

Please don't create large lumps of unrelated changes in a single pull request. Also take extra care to avoid spurious changes, like mass whitespace diffs. All Python sources use spaces to indent, not TABs.

3.5.2.3 Make sure your changes work

Some things that will increase the chance that your pull request is accepted:

- Follow style conventions you see used in the source already (and read PEP8).
- Include tests that fail *without* your code, and pass *with* it. Only minor refactoring and documentation changes require no new tests. If you are adding functionality or fixing a bug, please also add a test for it!
- Update any documentation or examples impacted by your change.
- Styling conventions and code quality are checked with `invoke check`, tests are run using `invoke test`, and the docs can be built locally using `invoke build --docs`.

Following these hints also expedites the whole procedure, since it avoids unnecessary feedback cycles.

3.6 Software License

The full LICENSE file and source are available at

https://github.com/jhermann/rituals

3.6.1 Full License Text

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc., <http://fsf.org/>
 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

CHAPTER 4

References

4.1 Tools

- Cookiecutter
- PyInvoke
- pytest
- tox
- Pylint
- twine
- bpython
- yolk3k

CHAPTER 5

Indices and Tables

- genindex
- modindex
- search

Python Module Index

Index