ecdysis Documentation

Release ???

Antoine Beaupré

Jan 17, 2024

CONTENTS

This project is a set of templates and code snippets I reuse by copy-pasting in other projects. They are generally not worth creating a full module or project for, since they are often small one-liners. Or they are too complex and overlap with existing functionalities. Ideally, those would be merged in the standard library.

This here is chaos and may make sense only to me.

Contents:

CHAPTER

ONE

USAGE

1.1 Quick start

Copy the code you need from the *ecdysis/* module. Search and replace the `ecdysis` string with your own module name. Backport to Python 2 if you need: the code targets Python 3, but should be easily backportable. Make sure you also check the `setup.py`, `setup.cfg`, `tox.ini`, `.gitignore` and similar files from the top directory to setup tests properly.

Copy the `doc/` directory into your project. Review and edit the *Contribution guide* guidelines and run `make html` to generate a HTML rendering of the documentation. Read the *API documentation* and copy-paste the code you need. Consider adding a changelog.

We use the doctest module to perform tests on the various functions because it allows us to quickly copy the tests along with the functions when we copy code around. Tests are discovered with pytest.

1.2 Documentation structure

Code is not everything. Documentation is really important too. This base package also features extensive self-documentation, but also documentation templates that can be reused.

The documentation is made of this `README` file, but is also rendered as a ReST (REStructured Text) document that is rendered into various formats (HTML, ePUB, PDF, etc) through the Sphinx documentation system. Special includes in the `index.rst` file do some magic to distribute parts of this file in the right sections of the online documentation.

1.3 Community guidelines

The community guidelines are described in the *Contribution guide* document, which provides a nice template that I reuse in other projects. It includes:

- a code of conduct
- how to send patches
- how documentation works
- how to report bugs
- how to make a release

It seems critical to me that every project should have such documentation.

1.4 Why the name?

The name comes from what snakes and other animals do to “create a new snake”: they shed their skin. This is not so appropriate for snakes, as it’s just a way to rejuvenate their skin, but is especially relevant for anthropods since the ecdysis may be associated with a metamorphosis:

> Ecdysis is the moulting of the cuticle in many invertebrates of the clade Ecdysozoa. Since the cuticle of these animals typically forms a largely inelastic exoskeleton, it is shed during growth and a new, larger covering is formed. The remnants of the old, empty exoskeleton are called exuviae.
>
> —Wikipedia

So this project is *metamorphosed* into others when the documentation templates, code examples and so on are reused elsewhere. For that reason, the license is an unusally liberal (for me) MIT/Expat license.

The name also has the nice property of being absolutely unpronounceable, which makes it unlikely to be copied but easy to search online.

CHAPTER

TWO

CODE SNIPPETS

This is Python code snippets I often reuse between different software. One day, maybe parts of this could be merged in the standard library or at least shipped in a reusable library?

2.1 Code documentation

This is the automatically generated documentation for the Python code.

2.1.1 ecdysis.argparse

improvements to the standard **:module:`argparse`** module

class `ecdysis.argparse.`**`NegateAction`**(*option_strings*, **args*, ***kwargs*)

add a toggle flag to argparse

this is similar to 'store_true' or 'store_false', but allows arguments prefixed with –no to disable the default. the default is set depending on the first argument - if it starts with the negative form (defined by default as '–no'), the default is False, otherwise True.

originally written for the stressant project.

@deprecated use the BooleanOptionalAction from Python 3.9 instead, although it doesn't have the default override we implemented here.

`negative = '--no'`

class `ecdysis.argparse.`**`ConfigAction`**(**args*, ***kwargs*)

add configuration file to current defaults.

a *list* of default config files can be specified and will be parsed when added by ConfigArgumentParser.

it was reported this might not work well with subparsers, patches to fix that are welcome.

`parse_config`(*path: str*) → dict

abstract implementation of config file parsing, should be overridden in subclasses

class `ecdysis.argparse.`**`YamlConfigAction`**(**args*, ***kwargs*)

YAML config file parser action

`parse_config`(*path: str*) → dict

This doesn't handle errors around open() and others, callers should probably catch FileNotFoundError at least.

class ecdysis.argparse.**ConfigArgumentParser**(**args*, ***kwargs*)

argument parser which supports parsing extra config files

Config files specified on the commandline through the YamlConfigAction arguments modify the default values on the spot. If a default is specified when adding an argument, it also gets immediately loaded.

This will typically be used in a subclass, like this:

> self.add_argument('–config', action=YamlConfigAction, default=self.default_config())

This shows how the configuration file overrides the default value for an option:

```
>>> from tempfile import NamedTemporaryFile
>>> c = NamedTemporaryFile()
>>> c.write(b"foo: delayed\n")
13
>>> c.flush()
>>> parser = ConfigArgumentParser()
>>> a = parser.add_argument('--foo', default='bar')
>>> a = parser.add_argument('--config', action=YamlConfigAction, default=[c.name])
>>> args = parser.parse_args([])
>>> args.config == [c.name]
True
>>> args.foo
'delayed'
>>> args = parser.parse_args(['--foo', 'quux'])
>>> args.foo
'quux'
```

This is the same test, but with *–config* called earlier, which should still work:

```
>>> from tempfile import NamedTemporaryFile
>>> c = NamedTemporaryFile()
>>> c.write(b"foo: quux\n")
10
>>> c.flush()
>>> parser = ConfigArgumentParser()
>>> a = parser.add_argument('--config', action=YamlConfigAction, default=[c.name])
>>> a = parser.add_argument('--foo', default='bar')
>>> args = parser.parse_args([])
>>> args.config == [c.name]
True
>>> args.foo
'quux'
>>> args = parser.parse_args(['--foo', 'baz'])
>>> args.foo
'baz'
```

This tests that you can override the config file defaults altogether:

```
>>> parser = ConfigArgumentParser()
>>> a = parser.add_argument('--config', action=YamlConfigAction, default=[c.name])
>>> a = parser.add_argument('--foo', default='bar')
>>> args = parser.parse_args(['--config', '/dev/null'])
>>> args.foo
'bar'
```

(continues on next page)

(continued from previous page)

```
>>> args = parser.parse_args(['--config', '/dev/null', '--foo', 'baz'])
>>> args.foo
'baz'
```

This tests multiple search paths, first one should be loaded:

```
>>> from tempfile import NamedTemporaryFile
>>> d = NamedTemporaryFile()
>>> d.write(b"foo: argh\n")
10
>>> d.flush()
>>> parser = ConfigArgumentParser()
>>> a = parser.add_argument('--config', action=YamlConfigAction, default=[d.name, c.
↪name])
>>> a = parser.add_argument('--foo', default='bar')
>>> args = parser.parse_args([])
>>> args.foo
'argh'
>>> c.close()
>>> d.close()
```

There are actually many other implementations of this we might want to consider instead of maintaining our own:

https://github.com/omni-us/jsonargparse https://github.com/bw2/ConfigArgParse https://github.com/omry/omegaconf

See this comment for a quick review:

https://github.com/borgbackup/borg/issues/6551#issuecomment-1094104453

parse_args(*args: Sequence[str] | None = None*, *namespace: Namespace | None = None*) → Namespace

default_config() → Iterable[str]

handy shortcut to detect commonly used config paths

This list is processed as a FIFO: if a file is found in there, it will be parsed and the remaining ones will be ignored.

class ecdysis.argparse.**LoggingAction**(**args*, ***kwargs*)

change log level on the fly

The logging system should be initialized before this, using *basicConfig*.

Example usage:

logging.basicConfig(level="WARNING", format="%(message)s") parser.add_argument(

"-v", "–verbose", action=LoggingAction, const="INFO", help="enable verbose messages",

) parser.add_argument(

"-d", "–debug", action=LoggingAction, const="DEBUG", help="enable debugging messages",

)

Or, if you want to default to "verbose" (AKA "INFO"):

logging.basicConfig(format="%(message)s") # INFO is default parser.add_argument(

“-q”, “–quiet”, action=LoggingAction, const=”WARNING”, help=”silence messages except warnings and errors”,

) parser.add_argument(

“-d”, “–debug”, action=LoggingAction, const=”DEBUG”, help=”enable debugging messages”,

)

2.1.2 ecdysis.cli

various commandline tools

`class ecdysis.cli.throbber(factor=0, stream=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>, symbol='.', fmt='{}', i=1)`

weird logarithmic “progress bar”

when a throbber object is called, will display progress using the provided “symbol”

the throbber will print the symbol every time it’s called until it crosses a logarithmic threshold (the “factor”), at which point the factor is increased.

this is useful to display progress on large datasets that have an unknown size (so we can’t guess completion time *and* we can’t reasonably guess the progress/display ratio).

originally from the code I wrote for the Euler project

this function requires Python 3.3 at least, because it uses print(flush=True)

Other progress bars include:

Rich: https://rich.readthedocs.io/en/stable/progress.html tqdm: https://github.com/tqdm/tqdm progress: https://pypi.org/project/progress/ progressbar: https://pypi.org/project/progressbar/

`class ecdysis.cli.Prompter`

Set of prompt utilities.

This is untested. It mostly comes from Monkeysign, but was rewritten for notmuch-sync-flagged and in doing so, was significantly refactored without further tests.

This could possibly be replaced with:

https://github.com/prompt-toolkit/python-prompt-toolkit https://github.com/Mckinsey666/bullet

`yes_no(prompt, default='y', choices=['y', 'n'])`

This will show the given prompt, check if it matches the given choices, and return True if it matches the first choice provided. If some “false” string (e.g. empty string which happens when you just hit “enter”) is provided, the default value (which should be a boolean) is returned.

For unit testing, the input function can be overridden with input_func.

```
>>> prompter = Prompter()
>>> prompter.input = lambda x: 'y'
>>> prompter.yes_no('foo')
True
>>> prompter.input = lambda x: 'n'
>>> prompter.yes_no('foo')
False
>>> prompter.input = lambda x: ''
>>> prompter.yes_no('foo', default='y')
```

(continues on next page)

(continued from previous page)

```
True
>>> prompter.yes_no('foo', default='n')
False
```

pick(*prompt*, *default*, *choices*)

acknowledge(*prompt=None*)

Just wait for the user to hit enter and return.

input(*prompt*)

Wrapper around python's input function, to ease testing.

input_pass(*prompt*)

Input without showing the typed characters on the terminal.

2.1.3 ecdysis.logging

- similarly, bup-cron has this GlobalLogger and a Singleton concept that may be useful elsewhere? it certainly does a nice job at setting up all sorts of handlers and stuff. stressant also has a *setup_logging* function that also supports colors and SMTP mailers. debmans has a neat log_warnings hook as well.
- monkeysign also has facilities to (ab)use the logging handlers to send stuff to the GTK framework (GTKLoggingHandler) and a error handler in GTK (in msg_exception.py)

ecdysis.logging.**logging_args**(*parser*)

```
>>> from pprint import pprint
>>> parser = argparse.ArgumentParser()
>>> logging_args(parser)
>>> pprint(sorted(parser.parse_args(['--verbose']).__dict__.items()))
[('email', None),
 ('logfile', None),
 ('loglevel', 'INFO'),
 ('smtppass', None),
 ('smtpserver', None),
 ('smtpuser', None),
 ('syslog', None)]
>>> pprint(sorted(parser.parse_args(['--verbose', '--debug']).__dict__.items()))
[('email', None),
 ('logfile', None),
 ('loglevel', 'DEBUG'),
 ('smtppass', None),
 ('smtpserver', None),
 ('smtpuser', None),
 ('syslog', None)]
>>> pprint(sorted(parser.parse_args(['--verbose', '--syslog']).__dict__.items()))
[('email', None),
 ('logfile', None),
 ('loglevel', 'INFO'),
 ('smtppass', None),
 ('smtpserver', None),
 ('smtpuser', None),
 ('syslog', 'INFO')]
```

`ecdysis.logging.`**`advancedConfig`**`(`*level='warning'*, *stream=None*, *syslog=False*, *prog=None*, *email=False*, *smtpparams=None*, *logfile=None*, *logFormat='%(levelname)s: %(message)s'*, ***kwargs*`)`

setup standard Python logging facilities

this was taken from the debmans and stressant loggers, although it lacks stressant's color support

Parameters

- **`level`** (*`str`*) – logging level, usually one of *levels*
- **`stream`** (*`file`*) – stream to send logging events to, or None to

use the logging default (usually stderr)

Parameters
`syslog` (*`str`*) – send log events to syslog at the specified

level. defaults to False, which doesn't send syslog events

Parameters
`prog` (*`str`*) – the program name to use in syslog lines, defaults

to .__prog__

Parameters
`email` (*`str`*) – send logs by email to the given email address

using the BufferedSMTPHandler

Parameters
`smtpparams` (*`dict`*) – parameters to use when sending

email. expected fields are:

- fromaddr (defaults to $USER@$FQDN)
- subject (defaults to '')
- mailhost (defaults to the last part of the destination email)
- user (to authenticate against the SMTP server, defaults to no auth)
- pass (password to use, prompted using getpass otherwise)

Parameters
`logfile` (*`str`*) – filename to pass to the FileHandler to log

directly to a file

Parameters
`logFormat` (*`str`*) – logformat to use for the FileHandler and

BufferedSMTPHandler

`class ecdysis.logging.`**`BufferedSMTPHandler`**`(`*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *capacity=5000*, *flushLevel=40*, *retries=1*`)`

A handler class which sends records only when the buffer reaches capacity. The object is constructed with the arguments from SMTPHandler and MemoryHandler and basically behaves as a merge between the two classes.

The SMTPHandler.emit() implementation was copy-pasted here because it is not flexible enough to be overridden. We could possibly override the format() function to instead look at the internal buffer, but that would have possibly undesirable side-effects.

emit(*record*)

buffer the record in the MemoryHandler

flush()

Flush all records.

Format the records and send it to the specified addressees.

The only change from SMTPHandler here is the way the email body is created.

2.1.4 ecdysis.os

various overrides to the builtin os library

ecdysis.os.**make_dirs_helper**(*path*)

Create the directory if it does not exist

Return True if the directory was created, false if it was already present, throw an OSError exception if it cannot be created

```
>>> import tempfile
>>> import os
>>> import os.path as p
>>> d = tempfile.mkdtemp()
>>> make_dirs_helper(p.join(d, 'foo'))
True
>>> make_dirs_helper(p.join(d, 'foo'))
False
>>> make_dirs_helper(p.join('/dev/null', 'foo'))
Traceback (most recent call last):
    ...
NotADirectoryError: [Errno 20] Not a directory: ...
>>> os.rmdir(p.join(d, 'foo'))
>>> os.rmdir(d)
>>>
```

2.1.5 ecdysis.packaging

ecdysis.packaging.**find_parent_module**()

find the name of a the first module calling this module

if we cannot find it, we return the current module's name (__name__) instead.

ecdysis.packaging.**find_static_file**(*path*, *module=None*)

locate a file in the distribution

this will look in the shipped files in the package

this assumes the files are at the root of the package or the source tree (if not packaged)

this does not check if the file actually exists.

Parameters

- **path** (*str*) – path for the file, relative to the source tree root

- **module** (*str*) – name of the module to find the find in. if None, guessed with *find_parent_module()*

Returns
the absolute path to the file

2.1.6 ecdysis.strings

2.1.7 ecdysis.time

2.2 Indices and tables

- genindex
- modindex
- search

CHAPTER

THREE

SUPPORT

If you have problems or question with this project, there are several options at your disposal:

- Try to troubleshoot the issue yourself
- Chat on IRC
- File bug reports

We of course welcome other contributions like documentation, translations and patches, see the *Contribution guide* guide for more information on how to contribute to the project.

3.1 Troubleshooting

The basic way to troubleshoot this program is to run the same command as you did when you had an error with the `--verbose` or, if that doesn't yield satisfactory results, with the `--debug` output.

Note: The debug output outputs a lot of information and may be confusing for new users.

If you suspect there is a bug specific to your environment, you can also try to see if it is reproducible within the testsuite. From there, you can either file a bug report or try to fix the issue yourself, see the *Contribution guide* section for more information.

Otherwise, see below for more options to get support.

3.2 Chat

We are often present in realtime in the `#anarcat` channel of the Freenode network. You can join the channel using a normal IRC client or using this web interface.

3.3 Bug reports

We want you to report bugs you find in this project. It's an important part of contributing to a project, and all bug reports will be read and replied to politely and professionally.

We are using an issue tracker to manage issues, and this is where bug reports should be sent.

Tip: A few tips on how to make good bug reports:

- Before you report a new bug, review the existing issues in the online issue tracker to make sure the bug has not already been reported elsewhere.
- The first aim of a bug report is to tell the developers exactly how to reproduce the failure, so try to reproduce the issue yourself and describe how you did that.
- If that is not possible, just try to describe what went wrong in detail. Write down the error messages, especially if they have numbers.
- Take the necessary time to write clearly and precisely. Say what you mean, and make sure it cannot be misinterpreted.
- Include the output of `--version` and `--debug` in your bug reports. See the issue template for more details about what to include in bug reports.

If you wish to read more about issues regarding communication in bug reports, you can read How to Report Bugs Effectively which takes about 30 minutes.

Warning: The output of the `--debug` may show information you may want to keep private. Do review the output before sending it in bug reports.

3.4 Commercial support

The project maintainers are available for commercial support for this software. If you have a feature you want to see prioritized or have a bug you absolutely need fixed, you can sponsor this development. Special licensing requirements may also be negociated if necessary. See *Contact* for more information on how to reach the maintainers.

CHAPTER

FOUR

CONTRIBUTION GUIDE

This document outlines how to contribute to this project. It details instructions on how to submit issues, bug reports and patches.

Before you participate in the community, you should agree to respect the *Code of conduct*.

Note:

Before you reuse this document in your own project, you will need to at least read the whole thing and make a few changes. Concretely, you will at least need to do the following changes:

- change the references at the top of the file to point to your project
- change the release process to follow your workflow (or remove it if releases are against your religion, which would be sad)
- also consider using a tool like DCO to assign copyright ownership
- obviously, remove or comment out this note when done

4.1 Positive feedback

Even if you have no changes, suggestions, documentation or bug reports to submit, even just positive feedback like "it works" goes a long way. It shows the project is being used and gives instant gratification to contributors. So we welcome emails that tell us of your positive experiences with the project or just thank you notes. Head out to contact for contact informations or submit a closed issue with your story.

You can also send your "thanks" through saythanks.io.

4.2 Documentation

We love documentation!

The documentation resides in various Sphinx documentations and in the README file. Those can can be edited online once you register and changes are welcome through the normal patch and merge request system.

Issues found in the documentation are also welcome, see below to file issues in our tracker.

4.3 Issues and bug reports

We want you to report issues you find in the software. It is a recognized and important part of contributing to this project. All issues will be read and replied to politely and professionnally. Issues and bug reports should be filed on the issue tracker.

4.3.1 Issue triage

Issue triage is a useful contribution as well. You can review the issues in the project page and, for each issue:

- try to reproduce the issue, if it is not reproducible, label it with `more-info` and explain the steps taken to reproduce
- if information is missing, label it with `more-info` and request specific information
- if the feature request is not within the scope of the project or should be refused for other reasons, use the `wontfix` label and close the issue
- mark feature requests with the `enhancement` label, bugs with `bug`, duplicates with `duplicate` and so on…

Note that some of those operations are available only to project maintainers, see below for the different statuses.

4.3.2 Security issues

Security issues should first be disclosed privately to the project maintainers (see *Contact*), which support receiving encrypted emails through the usual OpenPGP key discovery mechanisms.

This project cannot currently afford bounties for security issues. We would still ask that you coordinate disclosure, giving the project a reasonable delay to produce a fix and prepare a release before public disclosure.

Public recognition will be given to reporters security issues if desired. We otherwise agree with the Disclosure Guidelines of the HackerOne project, at the time of writing.

4.4 Patches

Patches can be submitted through merge requests on the project page.

Some guidelines for patches:

- A patch should be a minimal and accurate answer to exactly one identified and agreed problem.
- A patch must compile cleanly and pass project self-tests on all target platforms.
- A patch commit message must consist of a single short (less than 50 characters) line stating a summary of the change, followed by a blank line and then a description of the problem being solved and its solution, or a reason for the change. Write more information, not less, in the commit log.
- Patches should be reviewed by at least one maintainer before being merged.

Project maintainers should merge their own patches only when they have been approved by other maintainers, unless there is no response within a reasonable timeframe (roughly one week) or there is an urgent change to be done (e.g. security or data loss issue).

As an exception to this rule, this specific document cannot be changed without the consensus of all administrators of the project.

Note: Those guidelines were inspired by the Collective Code Construct Contract. The document was found to be a little too complex and hard to read and wasn't adopted in its entirety. See this discussion for more information.

4.4.1 Patch triage

You can also review existing pull requests, by cloning the contributor's repository and testing it. If the tests do not pass (either locally or in the online Continuous Integration (CI) system), if the patch is incomplete or otherwise does not respect the above guidelines, submit a review with "changes requested" with reasoning.

4.5 Membership

There are three levels of membership in the project, Administrator (also known as "Owner" in GitHub or GitLab), Maintainer (also known as "Member" on GitHub or "Developer" on GitLab), or regular users (everyone with or without an account). Anyone is welcome to contribute to the project within the guidelines outlined in this document, regardless of their status, and that includes regular users.

Maintainers can:

- do everything regular users can
- review, push and merge pull requests
- edit and close issues

Administrators can:

- do everything maintainers can
- add new maintainers
- promote maintainers to administrators

Regular users can be promoted to maintainers if they contribute to the project, either by participating in issues, documentation or pull requests.

Maintainers can be promoted to administrators when they have given significant contributions for a sustained timeframe, by consensus of the current administrators. This process should be open and decided as any other issue.

4.6 Release process

Note: This is just an example. There is no official release process for the ecdysis project right now, as the module is not publicly released or versioned.

To make a release:

1. generate release notes with:

   ```
   gbp dch
   ```

   the file header will need to be moved back up to the beginning of the file. also make sure to add a summary and choose a proper version according to Semantic Versioning

2. tag the release according to Semantic Versioning rules:

   ```
   git tag -s x.y.z
   ```

3. build and test the Python package:

   ```
   python setup.py bdist_wheel &&
   python3 -m venv ~/.venvs/ecdsysis --system-site-packages &&
   ~/.venvs/ecdsysis/bin/pip3 install $(ls -1tr dist/*.whl | tail -1) &&
   ~/.venvs/ecdsysis/bin/ecdsysis --version &&
   rm -rf ~/.venvs/feed2exec
   ```

4. build and test the debian package:

   ```
   git-buildpackage &&
   sudo dpkg -i $(ls -tr1 ../build-area/ecdysis_*.deb | tail -1) &&
   ecdysis --version &&
   sudo dpkg --purge ecdysis
   ```

5. push changes:

   ```
   git push
   git push --tags
   twine upload dist/*
   dput ../ecdysis*.changes
   ```

6. edit the tag, copy-paste the changelog entry and attach the signed binaries

CHAPTER

FIVE

CODE OF CONDUCT

Note:

Before you reuse this document in your own project, you will need to at least read the whole thing and make a few changes. Read more about community guidelines, code of conducts and intersectionality (also on geek feminism wiki) before adopting this: it is not just a rubber stamp, a badge to add to your project, but a real commitment with complex ethical ramification. Concretely, you will at least need to make sure there is a *Contact* section that details who can handle complaints and remove or comment out this note.

5.1 Contributor Covenant Code of Conduct

5.1.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

5.1.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

5.1.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

5.1.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

5.1.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting one of the persons listed in *Contact*. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project maintainers is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Project maintainers are encouraged to follow the spirit of the Django Code of Conduct Enforcement Manual when receiving reports.

5.1.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at http://contributor-covenant.org/version/1/4.

5.1.7 Changes

The Code of Conduct was modified to refer to *project maintainers* instead of *project team* and small paragraph was added to refer to the Django enforcement manual.

> Note: We have so far determined that writing an explicit enforcement policy is not necessary, considering the available literature already available online and the relatively small size of the community. This may change in the future if the community grows larger.

CHAPTER

SIX

TODO

6.1 README improvements

For now, I have only included here the *Contribution guide* document, but more should be added. There are a lot of templates around for `README` files which we could inspire from:

- the art of README - has a nice checklist, from the common-readme node.js module guy
- NPM has another standard readme spec
- Perl has standards, which are basically derived from manpages
- manpages, themselves, have a standard set of headings, well described in the mdoc(7) manpage
- Drupal has its own set of guidelines for README files
- yet another “kickass” README template
- Ben Ford has an article named Writing a great README which recommends the 5/30/90 rule: 5 seconds to decide if the README is worth a read, 30 seconds for the main pitch, 90 seconds for architecture, performance and more in-depth questions, then “the rest” (links to the rest of the docs, how to install, etc)
- finally, this list wouldn’t be complete without a good discussion on stackoverflow

I did a quick review of the Art of README checklist, and we’re not too bad for a template. Ironically, I should review my own writing in the bug reporting blog post.

I wonder if I should adopt the Semantic Line Breaks standard.

6.2 Changelog and index

The `index.rst` file should link to a template design document as well, along with an example of how to build a manpage. A changelog may be a good addition as well.

6.3 Man pages

There's also this whole thing about syncing the inline `--help` with documentation and the manpage. There's help2man that can be useful for simpler programs, and I have used it to bootstrap the manpage for undertime (I think). Otherwise, there's a whole slew of half-broken stuff to turn `argparse` output directly into a manpage in the build system. This is how Monkeysign works. Finally, what I actually prefer (and I do in stressant) is to write the manpage by hand, in RST, and convert it to a manpage at build time.

See also my other projects (e.g. monkeysign, linkchecker, wallabako, stressant, debmans) for more examples of the documentation layout.

6.4 Commit messages

The contribution guidelines could benefit from improvements regarding commit messages. People often write fairly bad commit messages in patches and commits on projects I participate in. It's also the case with bug reports, but we have fairly good instructions in the *Support* template here. Patches are specifically painful as there are no templates that can force users to do the right thing. There are some notes in the *Contribution guide* document, but they could be expanded. Some documents I need to review:

- How to Write a Git Commit Message - establishes “seven rules”:
  1. Separate subject from body with a blank line
  2. Limit the subject line to 50 characters
  3. Capitalize the subject line
  4. Do not end the subject line with a period
  5. Use the imperative mood in the subject line
  6. Wrap the body at 72 characters
  7. Use the body to explain what and why vs. how
- Linux kernel's SubmittingPatches documentation:
  - one patch should fix only one thing
  - describe the problem
  - describe the impact
  - quantify optimizations and tradeoffs (benchmarks!)
  - describe what is being done
  - use the imperative form (e.g. “make foo” instead of “this makes foo”)
  - refer to bug reports, URL, emails if relevant
  - refer to a commit hash if the commit fixes a regression
  - if you refer to a commit, mention the SHA-1 and the short description
- ProGit commit guidelines - formatting tips and some of the above
- Romulo Oliveira's guide is also interesting
- A great commit message and many more in this Hacker News discussion
- Conventional commits - a stricter approach?

- Patterns for writing better git commit messages - has good directives too
- **`Git Best Practices – How to Write Meaningful Commits, Effective Pull Requests, and Code Reviews https://www.freecodecamp.org/news/git-best-practices-commits-and-code-reviews/>`_** - "imperative", "brief", "helpful", examples of bad commits, small, meaningful commits

6.5 Funding

Another thing I'm looking at is donations for financing software projects. I don't like donations too much because that is charity-based, which skews social dynamics away from the republic and towards capital, but that's another political discussion that cannot be resolved in the short term. We still need to find ways of feeding developers and the options are rare. Here are a few reviews worth mentioning:

- review of funding approaches from Tyil
- GitHub's Open Source funding guide
- Snowdrift has an excellent market research about all this, including payment gateways and crowdfunding platforms
- Other ideas from a random GitHub project
- License zero
- Liberal Software and Percival's paid support
- GNOME apps funding

Whether any of this will be implemented in my projects remains an open question, for which I am still looking for feedback. One of the concerns is that launching a funding campaign that eventually fails could have a net negative reputation and psychological impacts. Furthermore, we may want to avoid supporting certain platforms that ban political speech… This is a minefield.

6.6 Code

I still don't know what to do with that code. Let loose, this could become like Stackoverflow: a huge dump of random code. Ideally, the following steps should be taken:

1. 100% documentation coverage
2. 100% test coverage
3. parts or everything published as (a?) module(s?)
4. parts or everything merged in the standard library
5. type-checking (mypy, pyright, feed2exec uses the former)

Stuff like the logging handlers, in particular, should especially be considered for merging. On the other hand, I also like the idea of simply copy-pasting small bits of code as needed. There *is* already a slugify module - yet my `ecdysis.slug()` function is still useful because it's much simpler and it's a one-liner that can be copy-pasted in your code without adding another dependency…

Note that code is nevertheless split up in modules that match the upstream module names where they could possibly end up, when relevant.

There are other code snippets that are not included here yet, because I'm not sure they're good enough or that I would actually reuse them:

- for pid files, I wrote my own PidFile class in bup-cron, but should look at lockfile

- to run commands, stressant has this useful *collectCmd* function to collect output of running commands. bup-cron also has shit like that.
- for `setup.py`, monkeysign has things to generate manpages (I used Sphinx instead in stressant), automatically call sphinx from the build chain, and translation stuff. debmans also has a neat __main__ hook. openstack's pbr project may be relevant here as well.
- monkeysign also has a UI abstraction layer that well… works more or less well, but at least works.
- gameclock also has some neat ideas that may be reused

Finally, it looks like Python is moving away from `setup.py` to build packages. Some tools have started using pyproject.toml instead, like flit and poetry. Unfortunately, neither supports reading the version number from git: flit reads it from the package's `__version__` variable (flit bug 257) and poetry hardcodes it in the `pyproject.toml` file, neither of which seem like the right solution as it duplicates information from the source of truth: git. So I'm still using setuptools, but I should probably consider moving the metadata to setup.cfg for the static ones (like trove classifiers) that do not need to be present at runtime.

CHAPTER

SEVEN

LICENSE

Unless otherwise noted, the content here is distributed under an Expat license since code snippets are small and we want to encourage code reuse.

CHAPTER EIGHT

CONTACT

This program was written by Antoine Beaupré. Please do not send email to maintainers privately unless you are looking for paid consulting or support. See *Contribution guide* for more information about how to collaborate on this project.

As a special exception, security issues can be reported privately using this contact information, where OpenPGP key material is also available.

The following people have volunteered to be available to respond to Code of Conduct reports. They have reviewed existing literature and agree to follow the aforementioned process in good faith. They also accept OpenPGP-encrypted email:

- Antoine Beaupré

Todo: https://github.com/kentcdodds/all-contributors?

PYTHON MODULE INDEX

INDEX