cl-conditions

Release

January 16, 2017

Contents

CHAPTER 1

Overview

Implementation of the Common Lisp's conditions system in Python.

Free software: BSD license.

docs	
tests	
package	

1.1 Rationale

Common Lisp (CL) has a very rich condition system. Conditions in CL is a some sort of signals, and used not only for exception handling but also in some other patterns. There is a very good explanation of how they works – a chapter from the book Practical Common Lisp by Peter Seibel: Beyond Exception Handling: Conditions and Restarts.

Python's exceptions cover only one scenerio from this book, but Common Lisp's conditions allows more interesting usage, particlarly "restarts". Restart is a way to continue code execution after the exception was signaled, without unwinding a call stack. I'll repeat: without unwinding a call stack.

Moreover, conditions allows to the author of the library to define varios cases to be choosen to take over the exception.

1.1.1 Example

Here is example from the book, but implemented in python using conditions library:

```
def parse_log_entry(text):
    """This function does all real job on log line parsing.
    it setup two cases for restart parsing if a line
    with wrong format was found.

    Restarts:
    - use_value: just retuns an object it was passed. This can
      be any value.
    - reparse: calls `parse_log_entry` again with other text value.
      Beware, this call can lead to infinite recursion.
    """
    text = text.strip()
```

```
    if well_formed_log_entry_p(text):
        return LogEntry(text)
    else:
        def use_value(obj):
            return obj
        def reparse(text):
            return parse_log_entry(text)

        with restarts(use_value,
                      reparse) as call:
            return call(signal, MalformedLogEntryError(text))


def log_analyzer(path):
    """This procedure replaces every line which can't be parsed
    with special object MalformedLogEntry.
    """
    with handle(MalformedLogEntryError,
                lambda (c):
                    invoke_restart('use_value',
                                   MalformedLogEntry(c.text))):
        for filename in find_all_logs(path):
            analyze_log(filename)


def log_analyzer2(path):
    """This procedure considers every line which can't be parsed
    as a line with ERROR level.
    """
    with handle(MalformedLogEntryError,
                lambda (c):
                    invoke_restart('reparse',
                                   'ERROR: ' + c.text)):
        for filename in find_all_logs(path):
            analyze_log(filename)
```

What we have here is a function `parse_log_entry` which defines two ways of handling an exceptional situation: `use_value` and `reparse`. But decision how bad lines should be handled is made by high level function `log_analyser`. Original book's chapter have only one version of the `log_analyser`, but I've added an alternative `log_analyser2` to illustrate a why restarts is a useful pattern. The value of this pattern is in the ability to move dicision making code from low level library functions into the higher level business logic.

Full version of this example can be found in example/example.py file.

1.2 Installation

```
pip install conditions
```

1.3 Documentation

https://python-cl-conditions.readthedocs.org/

1.4 Development

To run the all tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

1.5 Related projects

There is also withrestart python library, created with the same intent as conditions. But it have clunky API and weird name seems abandoned since 2010.

CHAPTER 2

Installation

At the command line:

```
pip install conditions
```

CHAPTER 3

Usage

To use cl-conditions in a project:

```
import conditions
```

CHAPTER 4

Reference

4.1 conditions

conditions.signals.**signal**(*e*)
Some docstrings.

conditions.handlers.**find_handler**(*e*)

conditions.handlers.**handle**(**args*, ***kwds*)

conditions.restarts.**find_restart**(*name*)

conditions.restarts.**invoke_restart**(*name*, **args*, ***kwargs*)

conditions.restarts.**restart**(*callback*)

class conditions.restarts.**restarts**(**callbacks*)

exception conditions.exceptions.**InvokeRestart**(*callback*, **args*, ***kwargs*)

exception conditions.exceptions.**RestartNotFoundError**

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

cl-conditions could always use more documentation, whether as part of the official cl-conditions docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/svetlyak40wt/python-cl-conditions/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.4 Development

To set up *python-cl-conditions* for local development:

1. Fork python-cl-conditions (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/python-cl-conditions.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`) [1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 6

Authors

- Alexander Artemenko - http://dev.svetlyak.ru

CHAPTER 7

Changelog

7.1 0.2.0 (2016-04-05)

- Added context manager `restarts` and manager `restart` now gets only a function and returns a function like to call code to be restarted.

7.2 0.1.0 (2016-03-29)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

C

C

F

H

I

R

S