syntactic

Release 0.1.3

Jan 11, 2020

Contents

docs	
tests	
package	

https://syntactic.readthedocs.io/

Customizable syntax for Python.

CHAPTER 1

Possible uses

- Experimenting with possible language features.
- Boilerplate reduction.

CHAPTER 2

Examples

2.1 Unicode lambdas

```
from __syntax__ import unicode_lambdas

increment = λx: x + 1
```

is equivalent to

```
increment = lambda x: x + 1
```

2.2 SQL template literals

Embedded sql:

```
from __syntax__ import sql_literals

engine.query(sql`SELECT author FROM books WHERE name = {book} AND author = {author}`)
```

is equivalent to:

```
engine.query('SELECT author FROM books WHERE name = ? AND author = ?', [book, author])
```

CHAPTER 3

Limitations

The example transformers are written in a fragile way. They are intended only as inspiration rather than production-ready transformers. If you want to add some production-ready ones, pull-requests are welcome.

CHAPTER 4

Related work

Several projects have explored manipulating Python syntax.

- MacroPy
- future-fstrings
- experimental

CHAPTER 5

Contents

5.1 Overview

docs	
tests	
package	

https://syntactic.readthedocs.io/

Customizable syntax for Python.

5.1.1 Possible uses

- Experimenting with possible language features.
- Boilerplate reduction.

5.1.2 Examples

Unicode lambdas

```
from __syntax__ import unicode_lambdas

increment = λx: x + 1
```

is equivalent to

```
increment = lambda x: x + 1
```

SQL template literals

Embedded sql:

```
from __syntax__ import sql_literals

engine.query(sql`SELECT author FROM books WHERE name = {book} AND author = {author}`)
```

is equivalent to:

```
engine.query('SELECT author FROM books WHERE name = ? AND author = ?', [book, author])
```

5.1.3 Limitations

The example transformers are written in a fragile way. They are intended only as inspiration rather than production-ready transformers. If you want to add some production-ready ones, pull-requests are welcome.

5.1.4 Related work

Several projects have explored manipulating Python syntax.

- MacroPy
- future-fstrings
- experimental

5.2 Installation

5.2.1 Basic

With pip:

```
pip install syntactic
```

With Poetry:

```
poetry add syntactic
```

5.2.2 With optional command-line tool

With pip:

```
pip install 'syntactic[cli]'
```

With Poetry:

```
poetry add 'syntactic[cli]'
```

5.3 Usage

5.3.1 Create a new custom syntax

1. Make a transformer

Create a function that takes the original unicode source string and returns a new unicode source string.

```
def unicode_lambdas(source: str) -> str:
    """Convert unicode lambdas into regular lambdas."""
    return source.replace("λ", "lambda ")
```

2. Put that function in a module named `__syntax__.py`. It may be in a package.

5.3.2 Use a custom syntax

0. Install `syntactic`.
1. Install a module that provides a custom syntax plugin.
2. In the module where you want to use the syntax, put the `syntactic` coding declaration at the top of the file.

```
# coding: syntactic
```

3. In the module where you want to use the syntax, import the desired syntax.

```
from __syntax__ import unicode_lambdas
```

If the module is in a package, namespace the import as normal. For example:

```
from syntactic.examples.__syntax__ import unicode_lambdas
```

4. Write code using the custom syntax. The full module should look like this:

```
# coding: syntactic

from __syntax__ import unicode_lambdas

add_one = λx: x+1

print(add_one(1))
```

5. Run the module using the python environment where syntactic is installed. The output should be:

```
2
```

5.3.3 View transformed syntax

View the expanded form of a Python file by using the optional command-line tool.

1. Ensure Syntactic's `cli` extra is installed.
2. Use `python -m syntactic show <filename>`.

5.4 Reference

5.4.1 syntactic package

Submodules

syntactic.app module

Support for custom syntax.

class `syntactic.app.`**IncrementalDecoder**(*errors='strict'*)
Bases: `codecs.BufferedIncrementalDecoder`

A buffered incremental decoder for custom syntax.

class `syntactic.app.`**StreamReader**(*stream*, *errors='strict'*)
Bases: `encodings.utf_8.StreamReader`

decode is deferred to support better error messages

stream
Get the stream.

`syntactic.app.`**decode**(*source_bytes*, *errors='strict'*)
Decode the utf-8 input and transform it with the named transformers.

`syntactic.app.`**get_transformer_pairs**(*source*)
Return the module and function names of requested transformers.

Searches for `from __syntax__ import ...`.

Return type `List[Tuple[str, str]]`

`syntactic.app.`**main**()
Register the codec with Python.

syntactic.cli module

syntactic.examples module

Module contents

Syntactic provides custom syntax for Python.

5.5 Contributing

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

5.5.1 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.

5.5.2 Documentation improvements

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

5.5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/metatooling/syntactic/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 :)

5.5.4 Development

To set up *syntactic* for local development:

1. Fork syntactic (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/syntactic.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.

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 file in `changelog.d/` describing the changes. The filename should be `{id}.{type}.rst`, where `{id}` is the number of the GitHub issue or pull request and `{type}` is one of `breaking` (for breaking changes), `deprecation` (for deprecations), or `change` (for non-breaking changes). For example, to add a new feature requested in GitHub issue #1234, add a file called `changelog.d/1234.change.rst` describing the change.
4. Add yourself to `AUTHORS.rst`.

Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

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

```
detox
```

5.6 Authors

- Metatooling - https://github.com/metatooling/

5.7 Changelog

5.7.1 0.1.0 (2019-12-30)

Changes

- First release on PyPI.

—

5.8 Indices and tables

- genindex
- modindex
- search

[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 ...

Python Module Index

S

Index