libfuncpy

Release 0.0.4

Joao M.C. Teixeira

Jul 19, 2021

CONTENTS

1 libfuncpy **1**
- 1.1 Motivation . . . 1

2 Installation **3**

3 Usage **5**

4 Contributing **7**
- 4.1 Fork this repository . . . 7
- 4.2 Install for developers . . . 7
- 4.3 Make a new branch . . . 8
- 4.4 Uniformed Tests with tox . . . 9

5 libfuncpy API **11**

6 Changelog **17**
- 6.1 v0.0.4 (2021-07-19) . . . 17
- 6.2 v0.0.3 (2021-06-29) . . . 17
- 6.3 v0.0.2 (2021-06-29) . . . 17
- 6.4 v0.0.1 (2021-06-29) . . . 17
- 6.5 v0.0.0 (2021-06-29) . . . 17

7 Authors **19**

8 Indices and tables **21**

Python Module Index **23**

Index **25**

CHAPTER

ONE

LIBFUNCPY

1.1 Motivation

Functional Programming tools in Python - extending beyond map, filter, reduce, and partial.

CHAPTER

TWO

INSTALLATION

Install `libfuncpy` from PyPI with:

```
pip install libfuncpy
```

Install it from GitHub with:

```
# clone the repository
git clone https://github.com/joaomcteixeira/libfuncpy

# move to the folder
cd libfuncpy

# install in develop mode
python setup.py develop
```

CHAPTER

THREE

USAGE

This is an example page for a real project. Describe here examples on how to use your software!

To use `libfuncpy`:

```
import libfuncpy
```

Here we explain how to contribute to a project that adopted this template. Actually, you can use this same scheme when contributing to this template. If you are completely new to `git` this might not be the best beginner tutorial, but will be very good still ;-)

You will notice that the text that appears is a mirror of the `CONTRIBUTING.rst` file. You can also point your community to that file (or the docs) to guide them in the steps required to interact with you project.

CHAPTER

FOUR

CONTRIBUTING

How to contribute to this project.

4.1 Fork this repository

Fork this repository before contributing.

4.1.1 Clone your fork

Next, clone your fork to your local machine, keep it up to date with the upstream, and update the online fork with those updates.

```
git clone https://github.com/YOUR-USERNAME/libfuncpy.git
cd libfuncpy
git remote add upstream git://github.com/joaomcteixeira/libfuncpy.git
git fetch upstream
git merge upstream/main
git pull origin main
```

4.2 Install for developers

Create a dedicated Python environment where to develop the project.

If you are using `pip` follow the official instructions on Installing packages using pip and virtual environments, most likely what you want is:

```
python3 -m venv libfuncpy
source libfuncpy/bin/activate
```

If you are using Anaconda go for:

```
conda create --name libfuncpy python=3.7
conda activate libfuncpy
```

Where `libfuncpy` is the name you wish to give to the environment dedicated to this project.

Either under *pip* or *conda*, install the package in `develop` mode, and also *tox*. **Note**, here I assume our project has **no** dependencies.

```
python setup.py develop
pip install tox
```

This configuration, together with the use of the `src` folder layer, guarantee that you will always run the code after installation. Also, thanks to the `develop` flag, any changes in the code will be automatically reflected in the installed version.

4.3 Make a new branch

From the `main` branch create a new branch where to develop the new code.

```
git checkout main
git checkout -b new_branch
```

Develop the feature and keep regular pushes to your fork with comprehensible commit messages.

```
git status
git add (the files you want)
git commit -m (add a nice commit message)
git push origin new_branch
```

While you are developing, you can execute `tox` as needed to run your unittests or inspect lint, etc. See the last section of this page.

4.3.1 Update CHANGELOG

Update the changelog file under `docs/CHANGELOG.rst` with an explanatory bullet list of your contribution. Add that list right after the main title and before the last version subtitle:

```
Changelog
=========

* here goes my new additions
* explain them shortly and well

vX.X.X (1900-01-01)
-------------------
```

Also add your name to the authors list at `docs/AUTHORS.rst`.

4.3.2 Pull Request

Once you are finished, you can Pull Request you additions to the main repository, and engage with the community. Please read the `PULLREQUEST.rst` guidelines first, you will see them when you open a PR.

Before submitting a Pull Request, verify your development branch passes all tests as *described bellow* **. If you are developing new code you should also implement new test cases.**

4.4 Uniformed Tests with tox

Thanks to Tox we can have a unified testing platform where all developers are forced to follow the same rules and, above all, all tests occur in a controlled Python environment.

With *Tox*, the testing setup can be defined in a configuration file, the tox.ini, which contains all the operations that are performed during the test phase. Therefore, to run the unified test suite, developers just need to execute `tox`, provided tox is installed in the Python environment in use.

```
pip install tox
# or
conda install tox -c conda-forge
```

Before creating a Pull Request from your branch, certify that all the tests pass correctly by running:

```
tox
```

These are exactly the same tests that will be performed online in the Github Actions.

Also, you can run individual environments if you wish to test only specific functionalities, for example:

```
tox -e lint  # code style
tox -e build  # packaging
tox -e docs  # only builds the documentation
tox -e prreqs  # special requirements before Pull Request
tox -e py37  # performs pytest in Python 3.7 environment (it should
be installed)
```

CHAPTER

FIVE

LIBFUNCPY API

Contain functions.

libfuncpy.lib.**ITE**(*iflogic*, *assertion*, *elselogic*)
Apply ternary operator logic executing functions.

Functions should be preconfigured and accept no arguments.

Better if you see the code:

return iflogic() if assertion() else elselogic()

libfuncpy.lib.**ITEX**(*x*, *iflogic*, *assertion*, *elselogic*)
Apply ternary operator logic executing functions.

Functions should receive a single value: *x*.

Better if you see the code:

return iflogic(x) if assertion(x) else elselogic(x)

Parameters **x** – The value to pass to each function.

libfuncpy.lib.**chainf**(*init*, **funcs*)
Run functions in sequence starting from an initial value.

Example

```
>>> chainf(2, [str, int, float])
2.0
```

libfuncpy.lib.**chainfs**(**funcs*)
Store functions be executed on a value.

Example

```
>>> do = chainfs(str, int, float)
>>> do(2)
2.0
```

libfuncpy.lib.**consume**(*gen*)
Consume generator in a single statement.

Example

```
>>> consume(generator)
```

libfuncpy.lib.**context_engine**(*func*, *exceptions*, *doerror*, *doelse*, *dofinally*, **args*, ***kwargs*)
Make a context engine.

libfuncpy.lib.**f1f2**(*f1*, *f2*, **a*, ***k*)
Apply one function after the other.

Call *f1* on the return value of *f2*.

Args and kwargs apply to *f2*.

Example

```
>>> f1f2(str, int, 2)
"2"
```

libfuncpy.lib.**f2f1**(*f1*, *f2*, **a*, ***k*)
Apply the second function after the first.

Call *f2* on the return value of *f1*.

Args and kwargs apply to *f1*.

Example

```
>>> f2f1(str, int, 2)
2
```

libfuncpy.lib.**flatlist**(*list_*)
Flat a list recursively.

This is a generator.

libfuncpy.lib.**give**(*value*)
Preare a function to return a value when called.

Ignore **args* and ***kwargs*.

Example

```
>>> true = give(True)
>>> true()
True
```

```
>>> five = give(5)
>>> five(4, 6, 7, 8, some_args='some string')
5
```

libfuncpy.lib.**if_elif_else**(*value*, *condition_function_pair*)
Apply logic if condition is True.

Parameters

- **value** (*anything*) – The initial value
- **condition_function_pair** (*tuple*) – First element is the assertion function, second element is the logic function to execute if assertion is true.

Returns *The result of the first function for which assertion is true.*

libfuncpy.lib.**ite**(*iflogic*, *assertion*, *elselogic*)
Apply ternary operator logic executing functions.

Functions should be preconfigured and accept no arguments.

Better if you see the code:

return iflogic() if assertion() else elselogic()

libfuncpy.lib.**itev**(*x*, *iflogic*, *assertion*, *elselogic*)
Apply ternary operator logic executing functions.

Functions should receive a single value: *x*.

Better if you see the code:

return iflogic(x) if assertion(x) else elselogic(x)

Parameters **x** – The value to pass to each function.

libfuncpy.lib.**make_iterable**(*value*)
Transform into an iterable.

Transforms a given *value* into an iterable if it is not. Else, return the value itself.

Example

```
>>> make_iterable(1)
[1]
```

```
>>> make_iterable([1])
[1]
```

libfuncpy.lib.**mapc**(*f*, **iterables*)
Consume map function.

Like *map()* but it is not a generator; *map* is consumed immediately.

libfuncpy.lib.**pass_**(*value*)
Do nothing, just pass the value.

Example

```
>>> pass_(1)
1
```

libfuncpy.lib.**raise_**(*exception*, **ignore*, ***everything*)
Raise exception.

libfuncpy.lib.**reduce_helper**(*value*, *f*, **a*, ***k*)
Help in *reduce*.

Helper function when applying *reduce* to a list of functions.

Parameters

- **value** (*anything*)
- **f** (*callable*) – The function to call. This function receives *value* as first positional argument.
- ***a, ****k** – Args and kwargs passed to *f*.

libfuncpy.lib.**ternary_operator**(*iflogic*, *assertion*, *elselogic*)
Apply ternary operator logic executing functions.

Functions should be preconfigured and accept no arguments.

Better if you see the code:

return iflogic() if assertion() else elselogic()

libfuncpy.lib.**ternary_operator_v**(*x*, *iflogic*, *assertion*, *elselogic*)
Apply ternary operator logic executing functions.

Functions should receive a single value: *x*.

Better if you see the code:

return iflogic(x) if assertion(x) else elselogic(x)

Parameters **x** – The value to pass to each function.

libfuncpy.lib.**ternary_operator_x**(*x*, *iflogic*, *assertion*, *elselogic*)
Apply ternary operator logic executing functions.

Functions should receive a single value: *x*.

Better if you see the code:

return iflogic(x) if assertion(x) else elselogic(x)

Parameters **x** – The value to pass to each function.

libfuncpy.lib.**vartial**(*func*, **args*, ***keywords*)
Prepare a function with args and kwargs except for the first arg.

Functions like *functools.partial* except that the resulting preprepared function expects the first positional argument.

Example

```
>>> pow2 = vartial(math.pow, 2)
>>> pow2(3)
9
>>> pow2(4)
16
```

This is different from: >>> pow_base_3 = partial(math.pow, 3) >>> pow_base_3(2) 9 >>> pow_base_3(4) 81

libfuncpy.lib.**whileloop**(*cond*, *func*, *do_stopiteration=<function give.<locals>.newfunc>*, *do_exhaust=<function give.<locals>.newfunc>*)
Execute while loop.

All function accept no arguments. If state needs to be evaluated, *cond* and *func* need to be synchronized.

Parameters

- **cond** (*callable*) – The *while* loop condition.

- **func** (*callable*) – The function to call on each while loop iteration.
- **do_stopiteration** (*callable*) – The function to execute when *func* raises StopIteration error.
- **do_exhaust** (*callable*) – The function to execute when while loop exhausts.

Returns *None*

CHAPTER

SIX

CHANGELOG

6.1 v0.0.4 (2021-07-19)

- Add docstrings
- add *mapc*
- add new names and list of deprecates
- disable *isort* checks

6.2 v0.0.3 (2021-06-29)

- implemented *make_iterable*

6.3 v0.0.2 (2021-06-29)

- corrected git actions

6.4 v0.0.1 (2021-06-29)

- PyPI badges

6.5 v0.0.0 (2021-06-29)

- Initial library commit

CHAPTER

SEVEN

AUTHORS

- Joao M. C. Teixeira (webpage, github)

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

libfuncpy.lib, 11

INDEX

C

chainf() (*in module libfuncpy.lib*), 11
chainfs() (*in module libfuncpy.lib*), 11
consume() (*in module libfuncpy.lib*), 11
context_engine() (*in module libfuncpy.lib*), 12

F

f1f2() (*in module libfuncpy.lib*), 12
f2f1() (*in module libfuncpy.lib*), 12
flatlist() (*in module libfuncpy.lib*), 12

G

give() (*in module libfuncpy.lib*), 12

I

if_elif_else() (*in module libfuncpy.lib*), 12
ITE() (*in module libfuncpy.lib*), 11
ite() (*in module libfuncpy.lib*), 13
itev() (*in module libfuncpy.lib*), 13
ITEX() (*in module libfuncpy.lib*), 11

L

libfuncpy.lib
    module, 11

M

make_iterable() (*in module libfuncpy.lib*), 13
mapc() (*in module libfuncpy.lib*), 13
module
    libfuncpy.lib, 11

P

pass_() (*in module libfuncpy.lib*), 13

R

raise_() (*in module libfuncpy.lib*), 13
reduce_helper() (*in module libfuncpy.lib*), 13

T

ternary_operator() (*in module libfuncpy.lib*), 14
ternary_operator_v() (*in module libfuncpy.lib*), 14
ternary_operator_x() (*in module libfuncpy.lib*), 14

V

vartial() (*in module libfuncpy.lib*), 14

W

whileloop() (*in module libfuncpy.lib*), 14