autosig Documentation

Release __version__ = '0.10.0'

Antonio Piccolboni

Jul 01, 2020

Contents:

CHAPTER 1

Introduction to autosig

Go straight to the documentation. Install with `pip install autosig`. Python 3 only.

1.1 Motivation

When I look at a great API I always observe a great level of consistency: similarly named and ordered arguments at a syntactic level; similar defaults, range of allowable values etc. on the semantic side. When looking at the code, one doesn't see these regularities represented very explicitly.

Imagine we are starting to develop a library with three entry points, `map`, `reduce` and `filter`:

```
from collections import Iterable


def map(function, iterable):
    assert callable(function)
    assert isinstance(iterable, Iterable)
    return (function(x) for x in iterable)


def reduce(function, iterable):
    total = next(iterable)
    for x in iterable:
        total = function(total, x)
    return total
```

(continues on next page)

(continued from previous page)

```
def filter(iterable, fun):
    if not isinstance(iterable, Iterable):
        iterable = [iterable]
    if isinstance(fun, set):
        fun = lambda x: x in fun
    return (x for x in iterable if fun(x))
```

But this is hardly well crafted. The order and naming of arguments isn't consistent. One function checks its argument right away. The next doesn't. The third attempts certain conversions to try and work with arguments that are not iterables or functions. There are reasons to build strict or tolerant APIs, but it's unlikely that mixing the two within the same API is a good idea, unless it's done deliberately (for instance offering a strict and tolerant version of every function). It wouldn't be difficult to fix these problems in this small API but we would end up with duplicated logic that we need to keep aligned for the foreseeable future. Let's do it instead the `autosig` way:

```
from autosig import param, Signature, autosig, check
from collections import Iterable


def to_callable(x):
    return (lambda y: y in x) if isinstance(x, set) else x


def to_iterable(x):
    return x if isinstance(x, Iterable) else [x]


API_signature = Signature(
    function=param(converter=to_callable, validator=callable),
    iterable=param(converter=to_iterable, validator=Iterable))


@autosig(API_signature)
def map(function, iterable):
    return (function(x) for x in iterable)


@autosig(API_signature)
def reduce(function, iterable):
    total = next(iterable)
    for x in iterable:
        total = function(total, x)
    return total


@autosig(API_signature)
def filter(function, iterable):
    return (x for x in iterable if function(x))
```

Let's go through it step by step. First we defined 2 simple conversion functions. This is a good first step independent of `autosig`. Next we create a signature object, with two parameters. These are intialized with objects that define the checking and conversion that need to be performed on those parameters, independent of which function is going to use that signature. A type works as a validator, as does any callable that returns *True* when a value is valid, returns *False* or raises an exception otherwise. Finally, we repeat the definition of our three API function, attaching the signature just defined with a decorator and then skipping all the checking and conversion logic and going straight to the meat of

the function!

At the cost of a little code we have gained a lot:

- Explicit definition of the desired API signature, in a single place — DRY principle;
- association of that signature with API functions, checked at load time — no room for error;
- uniform application of conversion and validation logic without repeating it;

`autosig` is the pro tool for the API designer! If you want to take a look at a real package that uses `autosig`, check out altair_recipes.

1.2 Features

- Define reusable parameters with defaults, conversion and validation logic, documentation, preferred position in the signature and whether keyword-only.
- Define reusable return values with conversion and validation logic and documentation.
- Define reusable signatures as ordered maps from names to parameters with optional return value definition.
- Combine signatures to create complex ones on top of simple ones.
- Decorate functions and methods with their signatures. Enforced at load time. Conversion and validation logic executed at call time.
- Not hot about signatures? You can just use parameters as in:

  ```
  @autosig
  def reduce(function = param(...), iterable=param(...)):
  ```

  for more free-form APIs.
- Open source (BSD license)
- Extensive property-based testing, excellent coverage

1.3 Credits

This package is heavily based on attrs. While that may change in the future, for now it must be said this is a thin layer over that, with a bit of reflection sprinkled over. It is, I suppose, a quite original direction to take `attrs` into.

CHAPTER 2

Installation

2.1 Stable release

To install autosig, run this command in your terminal:

```
$ pip install autosig
```

This is the preferred method to install autosig, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for autosig can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/piccolbo/autosig
```

Or download the tarball:

```
$ curl  -OL https://github.com/piccolbo/autosig/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use autosig in a project:

```
from autosig import *
```

To define a signature:

```
api_sig = Signature(x = param(default=0, converter=int))
```

To associate that signature with a function:

```
@autosig(api_sig)
def entry_point(x=0)
    # signature executed here, in this case int conversion
    return x
```

The same works with methods, just leave the self argument out:

```
class C:
    @autosig(api_sig)
    def entry_point(self, x=0)
        # signature executed here, in this case int conversion
        return x
```

Simple signatures can be combined to for more complex ones:

```
sig = Signature(x=param())+Signature(y=param())
```

Signatures can now include return values:

```
api_sig = Signature(Retval(validator=int), x = param(default=0, converter=int))
```

You can skip signatures altogehter and just capture commonalities between arguments with the argumentless form of the decorator:

```
x_arg = param(...)
y_arg = param(...)

@autosig
def entry_point(x = x_arg, y = y_arg):
    return x + y
```

`param` allows you to define a number of properties or behaviours of function arguments: validator, converter, docstring, default value, position,

CHAPTER 4

autosig

4.1 autosig package

Top-level package for autosig.

class autosig.**Signature**(**params*, ***kwparams*)

Bases: object

Class to represent signatures.

Parameters

- ***params** (*(str, attr.Attribute)*) – Optional first non-pair argument describes the return value. Each following argument is a pair with the name of an argument in the signature and a description of it generated with a call to param.
- ****kwparams** (*attr.Attribute*) – Each keyword argument becomes an argument named after the key in the signature of a function and must be initialized with a param call. Requires python >=3.6. If both *param and **params are provided the first will be concatenated with items of the second, in this order.

Returns The object created.

Return type *Signature*

set_late_init(*init*)

Set a function to be called immediately after all arguments have been initialized.

Use this function to perform initialization logic that involves multiple arguments in the signature.

Parameters **init** (*FunctionType*) – The init function is called after the initialization of all arguments in the signature but before the execution of the body of a function with that signature and is passed as an argument a dictionary with all arguments of the function. Returns None and acts exclusively by side effects.

Returns Returns self.

Return type *Signature*

autosig.**autosig**(*sig_or_f*)

Decorate functions or methods to attach signatures.

Use with (W) or without (WO) an argument:

```
@autosig(Signature(a = param(), b=param()))
def fun(a, b)
```

or, equivlently (WO):

```
@autosig
def fun(a=param(), b=param())
```

Do not include the self argument in the signature when decorating methods

Parameters **sig_or_f** (*Signature or function*) – An instance of class Signature (W) or a function or method (WO) whose arguments are intialized with a call to param.

Returns A decorator (W) or an already decorated function (WO) The decorated function, will initialize, convert, and validate its arguments and will include argument docstrings in its docstring.

Return type function

autosig.**param**(*default=NOTHING*, *validator=<function always_valid>*, *converter=<function identity>*, *docstring=''*, *position=-1*, *kw_only=False*)

Define parameters in a signature class.

Parameters

- **default** (*Any*) – The default value for the parameter (defaults to no default, that is, mandatory).
- **validator** (*callable or type*) – If a callable, it takes the actual parameter as an argument, raising an exception or returning False if invalid; returning True otherwise. If a type, the actual parameter must be instance of that type.
- **converter** (*callable*) – The callable is executed with the parameter as an argument and its value assigned to the parameter itself. Useful for type conversions, but not only (e.g. truncate range of parameter).
- **docstring** (*string*) – The docstring fragment for this parameter.
- **position** (*int*) – Desired position of the param in the signature. Negative values start from the end.
- **kw_only** (*bool*) – Whether to make this parameter keyword-only.

Returns Object describing all the properties of the parameter. Can be reused in multiple signature definitions to enforce consistency.

Return type attr.Attribute

class autosig.**Retval**(*validator=<function always_valid>*, *converter=<function identity>*, *docstring=''*)

Bases: object

Define return values in a Signature class.

Parameters

- **validator** (*callable or type*) – If a callable, it takes the return value as an argument, raising an exception or returning False if invalid; returning True otherwise. If a type, the return value must be an instance of that type.

- **converter** (*callable*) – The callable is executed with the return value as an argument and its return value is returned instead. Useful to enforce properties of return values, e.g. type, but not only.
- **docstring** (*string*) – The content for the docstring Returns section.

CHAPTER 5

Contributing

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

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/piccolbo/autosig/issues.

If you are 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.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

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

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/piccolbo/autosig/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 contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *autosig* for local development.

1. Fork the *autosig* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/autosig.git
```

3. Install your local copy into a virtualenv. This is how you set up your fork for local development:

```
$ curl -sSL https://raw.githubusercontent.com/sdispater/poetry/master/get-poetry.
↪py | python #if needed, or other method to install poetry
$ cd autosig
$ poetry install
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 autosig tests
$ make test
$ tox # in the works
```

6. 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
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/piccolbo/autosig/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ py.test tests.test_autosig
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 6

Credits

6.1 Development Lead

- Antonio Piccolboni <autosig@piccolboni.info>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

History

7.1 0.10.0 (2020-7-1)

- Support for return values in signatures

7.2 0.9.2 (2019-10-1)

- Single argument validators.
- Late init feature for signatures.

7.3 0.8.2 (2019-09-18)

- Switch from pipenv to poetry for development

7.4 0.8.0 (2019-08-27)

- autosig can decorate methods (exclude self from signature)

7.5 0.7.0 (2018-09-25)

- Argumentless autosig decorator for the use case of reusable parameter definitions but no reusable signatures.

7.6 0.6.0 (2018-09-24)

- Added `check` the quick validator generator. `check(int)` checks an argument is integer. `check(\lambda x: x>0)` checks an argument is positive. Behind the scenes it creates uses an assert statement which hopefully prints a useful message.

7.7 0.5.0 (2018-09-21)

- All new API, many breaking changes (sorry)
- signature decorator is gone
- create signatures directly withe the Signature constructor (it is no longer a base class to inherit from)
- do not use inheritance to define new signatures form old ones. It was a dead end as far as controlling the order of arguments. Use instead the + operator to combine two signatures, analogous to inheriting from one while adding new attributes.
- the new approach gives control over order of arguments, allows to mix mandatory and default arguments in one signature yet allow to reuse it ("stick" new mandatory arguments in between the arguments of the old signature)

7.8 0.4.1 (2018-09-05)

- Close abstraction holes revealing dependency on attr (which is gratefully acknowledged, but could be confusing).

7.9 0.3.1 (2018-08-30)

- Improved docstring generation

7.10 0.3.0 (2018-08-30)

- Compose docstring from param docstrings

7.11 0.2.3 (2018-08-28)

- Better and passing tests.

7.12 0.2.2 (2018-08-27)

- More stringent enforcement of signatures including defaults. Fixed build.

7.13 0.1.0 (2018-04-25)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index