
Fields

Release 5.0.0

Aug 26, 2021

Contents

1	Overview	1
1.1	Installation	1
1.2	Usage & examples	1
1.3	Documentation	3
1.4	Development	3
1.5	FAQ	3
1.6	Testimonials	6
1.7	Apologies	6
2	Installation	7
3	Usage	9
4	Reference	11
4.1	fields	11
4.2	fields.extras	13
5	Contributing	15
5.1	Bug reports	15
5.2	Documentation improvements	15
5.3	Feature requests and feedback	15
5.4	Development	16
6	Authors	17
7	Changelog	19
7.1	5.0.0 (2016-04-13)	19
7.2	4.0.0 (2016-01-28)	19
7.3	3.0.0 (2015-10-04)	19
7.4	2.4.0 (2015-06-13)	19
7.5	2.3.0 (2015-01-20)	20
7.6	2.2.0 (2015-01-19)	20
7.7	2.1.1 (2015-01-19)	20
7.8	2.1.0 (2015-01-09)	20
7.9	2.0.0 (2014-10-16)	20
7.10	1.0.0 (2014-10-05)	20
7.11	0.3.0 (2014-07-19)	21

7.12	0.2.0 (2014-06-28)	21
7.13	0.1.0 (2014-06-27)	21
8	Indices and tables	23
	Python Module Index	25
	Index	27

CHAPTER 1

Overview

docs	
tests	
package	

Container class boilerplate killer.

- Free software: BSD 2-Clause License

1.1 Installation

```
pip install fields
```

1.2 Usage & examples

A class that has 2 attributes, name and size:

```
>>> from fields import Fields
>>> class Pizza(Fields.name.size):
...     pass
...
>>> p = Pizza("Pepperoni", "large")
```

(continues on next page)

(continued from previous page)

```
>>> p
Pizza(name='Pepperoni', size='large')
>>> p.size
'large'
>>> p.name
'Pepperoni'
```

You can also use keyword arguments:

```
>>> Pizza(size="large", name="Pepperoni")
Pizza(name='Pepperoni', size='large')
```

You can have as many attributes as you want:

```
>>> class Pizza(Fields.name.ingredients.crust.size):
...     pass
...
>>> Pizza("Funghi", ["mushrooms", "mozzarella"], "thin", "large")
Pizza(name='Funghi', ingredients=['mushrooms', 'mozzarella'], crust='thin', size='large'
↳')
```

A class that has one required attribute value and two attributes (left and right) with default value None:

```
>>> class Node(Fields.value.left[None].right[None]):
...     pass
...
>>> Node(1, Node(2), Node(3, Node(4)))
Node(value=1, left=Node(value=2, left=None, right=None), right=Node(value=3,
↳left=Node(value=4, left=None, right=None), right=None))
>>> Node(1, right=Node(2))
Node(value=1, left=None, right=Node(value=2, left=None, right=None))
```

You can also use it *inline*:

```
>>> Fields.name.size("Pepperoni", "large")
FieldsBase(name='Pepperoni', size='large')
```

1.2.1 Want tuples?

An alternative to namedtuple:

```
>>> from fields import Tuple
>>> class Pair(Tuple.a.b):
...     pass
...
>>> issubclass(Pair, tuple)
True
>>> p = Pair(1, 2)
>>> p.a
1
>>> p.b
2
>>> tuple(p)
(1, 2)
>>> a, b = p
```

(continues on next page)

(continued from previous page)

```
>>> a
1
>>> b
2
```

Tuples are *fast*!

```
benchmark: 9 tests, min 5 rounds (of min 25.00us), 1.00s max time, timer: time.perf_
↳ counter
```

Name (time in us)	Min	Max	Mean	StdDev	Rounds	Iterations
test_characteristic	6.0100	1218.4800	11.7102	34.3158	15899	10
test_fields	6.8000	1850.5250	9.8448	33.8487	5535	4
test_slots_fields	6.3500	721.0300	8.6120	14.8090	15198	10
test_super_dumb	7.0111	1289.6667	11.6881	31.6012	15244	9
test_dumb	3.7556	673.8444	5.8010	15.0514	14246	18
test_tuple	3.1750	478.7750	5.1974	9.1878	14642	12
test_namedtuple	3.2778	538.1111	5.0403	9.9177	14105	9
test_attrs_decorated_class	4.2062	540.5125	5.3618	11.6708	14266	16
test_attrs_class	3.7889	316.1056	4.7731	6.0656	14026	18

1.3 Documentation

<https://python-fields.readthedocs.org/>

1.4 Development

To run all the tests run `tox` in your shell (`pip install tox` if you don't have it):

```
tox
```

1.5 FAQ

1.5.1 Why should I use this?

It's less to type, why have quotes around when the names need to be valid symbols anyway. In fact, this is one of the shortest forms possible to specify a container with fields.

1.5.2 But you're abusing a very well known syntax. You're using attribute access instead of a list of strings. Why?

Symbols should be symbols. Why validate strings so they are valid symbols when you can avoid that? Just use symbols. Save on both typing and validation code.

The use of language constructs is not that surprising or confusing in the sense that semantics precede conventional syntax use. For example, if we have `class Person(Fields.first_name.last_name.height.weight):`

pass then it's going to be clear we're talking about a *Person* object with *first_name*, *last_name*, *height* and *width* fields: the words have clear meaning.

Again, you should not name your variables as *f1*, *f2* or any other non-semantic symbols anyway.

Semantics precede syntax: it's like looking at a cake resembling a dog, you won't expect the cake to bark and run around.

1.5.3 Is this stable? Is it tested?

Yes. Mercilessly tested on [Travis](#) and [AppVeyor](#).

1.5.4 Is the API stable?

Yes, ofcourse.

1.5.5 Why not namedtuple?

It's ugly, repetitive and unflexible. Compare this:

```
>>> from collections import namedtuple
>>> class MyContainer(namedtuple("MyContainer", ["field1", "field2"])):
...     pass
>>> MyContainer(1, 2)
MyContainer(field1=1, field2=2)
```

To this:

```
>>> class MyContainer(Tuple.field1.field2):
...     pass
>>> MyContainer(1, 2)
MyContainer(field1=1, field2=2)
```

1.5.6 Why not characteristic?

Ugly, inconsistent - you don't own the class:

Lets try this:

```
>>> import characteristic
>>> @characteristic.attributes(["field1", "field2"])
... class MyContainer(object):
...     def __init__(self, a, b):
...         if a > b:
...             raise ValueError("Expected %s < %s" % (a, b))
>>> MyContainer(1, 2)
Traceback (most recent call last):
...
ValueError: Missing keyword value for 'field1'.
```

WHAT !? Ok, lets write some more code:


```
>>> MyContainer(field1=1, field2=2)
Traceback (most recent call last):
...
TypeError: __init__() ... arguments...
```

This is bananas. You have to write your class *around* these quirks.

Lets try this:

```
>>> class MyContainer(Fields.field1.field2):
...     def __init__(self, a, b):
...         if a > b:
...             raise ValueError("Expected %s < %s" % (a, b))
...         super(MyContainer, self).__init__(a, b)
```

Just like a normal class, works as expected:

```
>>> MyContainer(1, 2)
MyContainer(field1=1, field2=2)
```

1.5.7 Why not attrs?

Now this is a very difficult question.

Consider this typical use-case:

```
>>> import attr
>>> @attr.s
... class Point(object):
...     x = attr.ib()
...     y = attr.ib()
```

Worth noting:

- `attrs` is faster because it doesn't allow your class to be used as a mixin (it doesn't do any `super(cls, self).__init__(...)` for you).
- The typical use-case doesn't allow you to have a custom `__init__`. You can use `@attr.s(init=False)` that will allow you to implement your own `__init__`. However, you can't have your own `__init__` that calls `attrs` provided `__init__` (like in a subclassing scenario).
- It works better with IDEs and source code analysis tools because of the attributes defined on the class.
- It's more composable if you only use `@attr.s` decorated hierarchies. Example:

```
>>> @attr.s
... class A(object):
...     a = attr.ib()
...     def get_a(self):
...         return self.a
>>> @attr.s
... class B(object):
...     b = attr.ib()
>>> @attr.s
... class C(B, A):
...     c = attr.ib()
>>> C(1, 2, 3)
C(a=1, b=2, c=3)
```

All in all, `attrs` is a fast and minimal container library that does support subclasses, but quite differently than `fields`. Definitely worth considering.

Also, nowadays it has more features than `fields`. See [#6](#).

1.5.8 Won't this confuse `pylint`?

Normaly it would, but there's a plugin that makes `pylint` understand it, just like any other class: `pylint-fields`.

1.6 Testimonials

Diabolical. Can't be unseen.

—David Beazley

I think that's the saddest a single line of python has ever made me.

—Someone on IRC (#python)

Don't speak around saying that I like it.

—A PyPy contributor

Fields is completey bat-shit insane, but kind of cool.

—Someone on IRC (#python)

WHAT?!?!

—Unsuspecting victim at EuroPython 2015

I don't think it should work ...

—Unsuspecting victim at EuroPython 2015

Is it some Ruby thing?

—Unsuspecting victim at EuroPython 2015

Are Python programmers that lazy?

—Some Java developer

I'm going to use this in my next project. You're a terrible person.

—Isaac Dickinson

It's so bad you had to write a `pylint` plugin :)

—Colin Dunklau on IRC (#python)

1.7 Apologies

I tried my best at [EuroPython](#) ...

CHAPTER 2

Installation

At the command line:

```
pip install fields
```


CHAPTER 3

Usage

To use Fields in a project:

```
import fields
```


4.1 fields

How it works: the library is composed of 2 major parts:

- The *sealers*. They return a class that implements a container according the given specification (list of field names and default values).
- The *factory*. A metaclass that implements attribute/item access, so you can do `Fields.a.b.c`. On each `getattr/getitem` it returns a new instance with the new state. Its `__new__` method takes extra arguments to store the construction state and it works in two ways:
 - Construction phase (there are no bases). Make new instances of the *Factory* with new state.
 - Usage phase. When subclassed (there are bases) it will use the sealer to return the final class.

`fields.factory(sealer, **sealer_options)`

Create a factory that will produce a class using the given `sealer`.

Parameters

- **sealer** (*func*) – A function that takes `fields`, `defaults` as arguments, where:
 - `fields` (list): A list with all the field names in the declared order.
 - `defaults` (dict): A dict with all the defaults.
- **sealer_options** – Optional keyword arguments passed to `sealer`.

Returns

A class on which you can do `.field1.field2.field3...`. When it's subclassed it “seals”, and whatever the `sealer` returned for the given fields is used as the baseclass.

Example:

```
>>> def sealer(fields, defaults):  
...     print("Creating class with:")
```

(continues on next page)

(continued from previous page)

```

...     print("  fields = {0}".format(fields))
...     print("  defaults = {0}".format(defaults))
...     return object
...
>>> Fields = factory(sealer)
>>> class Foo(Fields.foo.bar.lorem[1].ipsum[2]):
...     pass
...
Creating class with:
  fields = ['foo', 'bar', 'lorem', 'ipsum']
  defaults = OrderedDict([('lorem', 1), ('ipsum', 2)])
>>> Foo
<class '...Foo'>
>>> Foo.__bases__
(<... 'object'>,)

```

`fields.class_sealer` (*fields*, *defaults*, *base*=<class 'fields.__base__'>, *make_init_func*=<function *make_init_func*>, *initializer*=True, *comparable*=True, *printable*=True, *convertible*=False, *pass_kwargs*=False)

This sealer makes a normal container class. It's mutable and supports arguments with default values.

`fields.slots_class_sealer` (*fields*, *defaults*)

This sealer makes a container class that uses `__slots__` (it uses `class_sealer()` internally).

The resulting class has a metaclass that forcibly sets `__slots__` on subclasses.

`fields.tuple_sealer` (*fields*, *defaults*)

This sealer returns an equivalent of a namedtuple.

`class fields.Namespace` (***kwargs*)

A backport of Python 3.3's `types.SimpleNamespace`.

`class fields.Fields`

Container class generator. The resulting class will implement `__repr__`, `__init__`, `__eq__`, `__ne__`, `__lt__`, `__gt__`, `__le__`, `__ge__` and `__hash__`.

Usage:

```

class Foo(Fields.foo.bar):
    pass

```

`class fields.BareFields`

Container class generator. The resulting class will implement `__init__`.

Usage:

```

class Foo(BareFields.foo.bar):
    pass

```

`class fields.PrintableMixin`

Container class generator. The resulting class will implement `__repr__`.

Usage:

```

class Foo(PrintableMixin.foo.bar):
    # we need to have the `foo` and `bar` attributes
    foo = None
    bar = None

```


class fields.ComparableMixin

Container class generator. The resulting class will implement `__eq__`, `__ne__`, `__lt__`, `__gt__`, `__le__`, `__ge__` and `__hash__`.

Usage:

```
class Foobar(BareFields.name.extra, ComparableMixin.name):  
    """  
    A class that only compares on `name` but has an `extra` field.  
    """  
    pass
```

4.2 fields.extras

exception fields.extras.ValidationError

fields.extras.regex_validation_sealer (*fields, defaults, RegexType=<class 're.Pattern'>*)

Example sealer that just does regex-based validation.

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

Fields could always use more documentation, whether as part of the official Fields 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/ionelmc/python-fields/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-fields* for local development:

1. Fork [python-fields](#) (look for the “Fork” button).
2. Clone your fork locally:

```
git clone git@github.com:YOURGITHUBNAME/python-fields.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 and docs builder 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 -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

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

- Ionel Cristian Mărieș - <https://blog.ionelmc.ro>

7.1 5.0.0 (2016-04-13)

- Added the `fields.InheritableFields` base. It allows subclassing and it's intended for multiple inheritance scenarios. Yes, yes, this enables lots pain and suffering, but some people just like it that way.
- Simplified the interfaces for the builtin sealers (the required argument is gone as it was redundant, and the remaining arguments are swapped). Now they must be a function that take just two arguments: `fields`, `defaults`.

7.2 4.0.0 (2016-01-28)

- Added `__all__` and `factory` conveniences. Removed `fields.Factory` from the public API since it need some special care with it's use (it's a damn metaclass after all).
- Added `make_init_func` into public API for advanced uses (combine with `factory` and `class_sealer`).

7.3 3.0.0 (2015-10-04)

- Disallowed creating containers with fields with “dunder” names. E.g.: `class Foo(Fields.__foo__):` is disallowed.

7.4 2.4.0 (2015-06-13)

- Similarly to `fields.Fields`, added three new bases:
 - `fields.BareFields` (implements `__init__`).

- `fields.ComparableMixin` (implements `__eq__`, `__ne__`, `__lt__`, `__gt__`, `__le__`, `__ge__` and `__hash__`).
- `fields.PrintableMixin` (implements `__repr__`).
- Improved reference section in the docs.
- Added `fields.ConvertibleFields` and `fields.ConvertibleMixin`. They have two convenience properties: `as_dict` and `as_tuple`.

7.5 2.3.0 (2015-01-20)

- Allowed overriding `__slots__` in `SlotsFields` subclasses.

7.6 2.2.0 (2015-01-19)

- Added `make_init_func` as an optional argument to `class_sealer`. Rename the `__base__` option to just `base`.

7.7 2.1.1 (2015-01-19)

- Removed bogus `console_scripts` entrypoint.

7.8 2.1.0 (2015-01-09)

- Added `SlotsFields` (same as `Fields` but automatically adds `__slots__` for memory efficiency on CPython).
- Added support for default argument to `Tuple`.

7.9 2.0.0 (2014-10-16)

- Made the `__init__` in the `FieldsBase` way faster (used for `fields.Fields`).
- Moved `RegexValidate` in `fields.extras`.

7.10 1.0.0 (2014-10-05)

- Lots of internal changes, the metaclass is not created in a closure anymore. No more closures.
- Added `RegexValidate` container creator (should be taken as an example on using the `Factory` metaclass).
- Added support for using multiple containers as baseclasses.
- Added a `super()` *sink* so that `super().__init__(*args, **kwargs)` always works. Everything inherits from a baseclass that has an `__init__` that can take any argument (unlike `object.__init__`). This allows for flexible usage.

- Added validation so that you can't use conflicting field layout when using multiple containers as the baseclass.
- Changed the `__init__` function in the class container so it works like a python function w.r.t. positional and keyword arguments. Example: `class MyContainer(Fields.a.b.c[1].d[2])` will function the same way as `def func(a, b, c=1, d=2)` would when arguments are passed in. You can now use `MyContainer(1, 2, 3, 4)` (everything positional) or `MyContainer(1, 2, 3, d=4)` (mixed).

7.11 0.3.0 (2014-07-19)

- Corrected string repr.

7.12 0.2.0 (2014-06-28)

- Lots of breaking changes. Switched from `__call__` to `__getitem__` for default value assignment.

7.13 0.1.0 (2014-06-27)

- Alpha release.

CHAPTER 8

Indices and tables

- `genindex`
- `modindex`
- `search`

f

`fields`, [11](#)

`fields.extras`, [13](#)

C

`class_sealer()` (*in module fields*), 12

F

`factory()` (*in module fields*), 11

`fields` (*module*), 11

`fields.BareFields` (*class in fields*), 12

`fields.ComparableMixin` (*class in fields*), 12

`fields.extras` (*module*), 13

`fields.Fields` (*class in fields*), 12

`fields.PrintableMixin` (*class in fields*), 12

N

`Namespace` (*class in fields*), 12

R

`regex_validation_sealer()` (*in module fields.extras*), 13

S

`slots_class_sealer()` (*in module fields*), 12

T

`tuple_sealer()` (*in module fields*), 12

V

`ValidationError`, 13