# attrs Documentation

***Release 15.2.0***

**Hynek Schlawack**

**Sep 25, 2017**

# Contents

Release v15.2.0 (*What's new?*).

attrs is an MIT-licensed Python package with class decorators that ease the chores of implementing the most common attribute-related object protocols:

```
>>> import attr
>>> @attr.s
... class C(object):
...     x = attr.ib(default=42)
...     y = attr.ib(default=attr.Factory(list))
>>> i = C(x=1, y=2)
>>> i
C(x=1, y=2)
>>> i == C(1, 2)
True
>>> i != C(2, 1)
True
>>> attr.asdict(i)
{'y': 2, 'x': 1}
>>> C()
C(x=42, y=[])
>>> C2 = attr.make_class("C2", ["a", "b"])
>>> C2("foo", "bar")
C2(a='foo', b='bar')
```

(If you don't like the playful attr.s and attr.ib, you can also use their no-nonsense aliases attr.attributes and attr.attr).

You just specify the attributes to work with and attrs gives you:

- a nice human-readable __repr__,
- a complete set of comparison methods,
- an initializer,
- and much more

*without* writing dull boilerplate code again and again.

This gives you the power to use actual classes with actual types in your code instead of confusing tuples or confusingly behaving namedtuples.

So put down that type-less data structures and welcome some class into your life!

---

**Note:** I wrote an explanation on why I forked my own characteristic. It's not dead but attrs will have more new features.

---

attrs's documentation lives at Read the Docs, the code on GitHub. It's rigorously tested on Python 2.6, 2.7, 3.3+, and PyPy.

CHAPTER 1

# User's Guide

## Why not...

### ...tuples?

#### Readability

What makes more sense while debugging:

```
Point(x=1, x=2)
```

or:

```
(1, 2)
```

?

Let's add even more ambiguity:

```
Customer(id=42, reseller=23, first_name="Jane", last_name="John")
```

or:

```
(42, 23, "Jane", "John")
```

?

Why would you want to write `customer[2]` instead of `customer.first_name`?

Don't get me started when you add nesting. If you've never ran into mysterious tuples you had no idea what the hell they meant while debugging, you're much smarter then I am.

Using proper classes with names and types makes program code much more readable and comprehensible. Especially when trying to grok a new piece of software or returning to old code after several months.

### Extendability

Imagine you have a function that takes or returns a tuple. Especially if you use tuple unpacking (eg. `x, y = get_point()`), adding additional data means that you have to change the invocation of that function *everywhere*.

Adding an attribute to a class concerns only those who actually care about that attribute.

## . . . namedtuples?

The difference between `collections.namedtuple()`s and classes decorated by `attrs` is that the latter are type-sensitive and less typing aside regular classes:

```
>>> import attr
>>> @attr.s
... class C1(object):
...     a = attr.ib()
...     def print_a(self):
...         print self.a
>>> @attr.s
... class C2(object):
...     a = attr.ib()
>>> c1 = C1(a=1)
>>> c2 = C2(a=1)
>>> c1.a == c2.a
True
>>> c1 == c2
False
>>> c1.print_a()
1
```

. . . while namedtuple's purpose is *explicitly* to behave like tuples:

```
>>> from collections import namedtuple
>>> NT1 = namedtuple("NT1", "a")
>>> NT2 = namedtuple("NT2", "b")
>>> t1 = NT1._make([1,])
>>> t2 = NT2._make([1,])
>>> t1 == t2 == (1,)
True
```

This can easily lead to surprising and unintended behaviors.

Other than that, `attrs` also adds nifty features like validators or default values.

## . . . hand-written classes?

While I'm a fan of all things artisanal, writing the same nine methods all over again doesn't qualify for me. I usually manage to get some typos inside and there's simply more code that can break and thus has to be tested.

To bring it into perspective, the equivalent of

```
>>> @attr.s
... class SmartClass(object):
...     a = attr.ib()
...     b = attr.ib()
>>> SmartClass(1, 2)
SmartClass(a=1, b=2)
```

is

```
>>> class ArtisanalClass(object):
...     def __init__(self, a, b):
...         self.a = a
...         self.b = b
...
...     def __repr__(self):
...         return "ArtisanalClass(a={}, b={})".format(self.a, self.b)
...
...     def __eq__(self, other):
...         if other.__class__ is self.__class__:
...             return (self.a, self.b) == (other.a, other.b)
...         else:
...             return NotImplemented
...
...     def __ne__(self, other):
...         result = self.__eq__(other)
...         if result is NotImplemented:
...             return NotImplemented
...         else:
...             return not result
...
...     def __lt__(self, other):
...         if other.__class__ is self.__class__:
...             return (self.a, self.b) < (other.a, other.b)
...         else:
...             return NotImplemented
...
...     def __le__(self, other):
...         if other.__class__ is self.__class__:
...             return (self.a, self.b) <= (other.a, other.b)
...         else:
...             return NotImplemented
...
...     def __gt__(self, other):
...         if other.__class__ is self.__class__:
...             return (self.a, self.b) > (other.a, other.b)
...         else:
...             return NotImplemented
...
...     def __ge__(self, other):
...         if other.__class__ is self.__class__:
...             return (self.a, self.b) >= (other.a, other.b)
...         else:
...             return NotImplemented
...
...     def __hash__(self):
...         return hash((self.a, self.b))
>>> ArtisanalClass(a=1, b=2)
ArtisanalClass(a=1, b=2)
```

which is quite a mouthful and it doesn't even use any of `attrs`'s more advanced features like validators or defaults values. Also: no tests whatsoever. And who will guarantee you, that you don't accidentally flip the < in your tenth implementation of `__gt__`?

If you don't care and like typing, I'm not gonna stop you. But if you ever get sick of the repetitiveness, `attrs` will

be waiting for you. :)

### . . . characteristic

characteristic is a very similar and fairly popular project of mine. So why the self-fork? Basically after nearly a year of usage I ran into annoyances and regretted certain decisions I made early-on to make too many people happy. In the end, *I* wasn't happy using it anymore.

So I learned my lesson and `attrs` is the result of that.

---

**Note:** Nevertheless, `characteristic` is **not** dead. A lot of software uses it and I will keep maintaining it.

---

#### Reasons For Forking

- Fixing those aforementioned annoyances would introduce more complexity. More complexity means more bugs.
- Certain unused features make other common features complicated or impossible. Prime example is the ability write your own initializers and make the generated one cooperate with it. The new logic is much simpler allowing for writing optimal initializers.
- I want it to be possible to gradually move from `characteristic` to `attrs`. A peaceful co-existence is much easier if it's separate packages altogether.
- My libraries have very strict backward-compatibility policies and it would take years to get rid of those annoyances while they shape the implementation of other features.
- The name is tooo looong.

#### Main Differences

- The attributes are defined *within* the class definition such that code analyzers know about their existence. This is useful in IDEs like PyCharm or linters like PyLint. `attrs`'s classes look much more idiomatic than `characteristic`'s. Since it's useful to use `attrs` with classes you don't control (e.g. Django models), a similar way to `characteristic`'s is still supported.
- The names are held shorter and easy to both type and read.
- It is generally more opinionated towards typical uses. This ensures I'll not wake up in a year hating to use it.
- The generated `__init__` methods are faster because of certain features that have been left out intentionally. The generated code should be as fast as hand-written one.

## Examples

### Basics

The simplest possible usage would be:

```
>>> import attr
>>> @attr.s
... class Empty(object):
...     pass
>>> Empty()
```

```
Empty()
>>> Empty() == Empty()
True
>>> Empty() is Empty()
False
```

So in other words: `attrs` useful even without actual attributes!

But you'll usually want some data on your classes, so let's add some:

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

These by default, all features are added, so you have immediately a fully functional data class with a nice `repr` string and comparison methods.

```
>>> c1 = Coordinates(1, 2)
>>> c1
Coordinates(x=1, y=2)
>>> c2 = Coordinates(x=2, y=1)
>>> c2
Coordinates(x=2, y=1)
>>> c1 == c2
False
```

As shown, the generated `__init__` method allows both for positional and keyword arguments.

If playful naming turns you off, `attrs` comes with no-nonsense aliases:

```
>>> @attr.attributes
... class SeriousCoordinates(object):
...     x = attr.attr()
...     y = attr.attr()
>>> SeriousCoordinates(1, 2)
SeriousCoordinates(x=1, y=2)
>>> attr.fields(Coordinates) == attr.fields(SeriousCoordinates)
True
```

For private attributes, `attrs` will strip the leading underscores for keyword arguments:

```
>>> @attr.s
... class C(object):
...     _x = attr.ib()
>>> C(x=1)
C(_x=1)
```

An additional way (not unlike `characteristic`) of defining attributes is supported too. This is useful in times when you want to enhance classes that are not yours (nice `__repr__` for Django models anyone?):

```
>>> class SomethingFromSomeoneElse(object):
...     def __init__(self, x):
...         self.x = x
>>> SomethingFromSomeoneElse = attr.s(these={"x": attr.ib()},␣
↪init=False)(SomethingFromSomeoneElse)
>>> SomethingFromSomeoneElse(1)
SomethingFromSomeoneElse(x=1)
```

Or if you want to use properties:

```
>>> @attr.s(these={"_x": attr.ib()})
... class ReadOnlyXSquared(object):
...     @property
...     def x(self):
...         return self._x ** 2
>>> rox = ReadOnlyXSquared(x=5)
>>> rox
ReadOnlyXSquared(_x=5)
>>> rox.x
25
>>> rox.x = 6
Traceback (most recent call last):
   ...
AttributeError: can't set attribute
```

Sub-classing is bad for you, but `attrs` will still do what you'd hope for:

```
>>> @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()
>>> i = C(1, 2, 3)
>>> i
C(a=1, b=2, c=3)
>>> i == C(1, 2, 3)
True
>>> i.get_a()
1
```

The order of the attributes is defined by the MRO.

In Python 3, classes defined within other classes are detected and reflected in the `__repr__`. In Python 2 though, it's impossible. Therefore `@attr.s` comes with the `repr_ns` option to set it manually:

```
>>> @attr.s
... class C(object):
...     @attr.s(repr_ns="C")
...     class D(object):
...         pass
>>> C.D()
C.D()
```

`repr_ns` works on both Python 2 and 3. On Python 3 is overrides the implicit detection.

## Converting to Dictionaries

When you have a class with data, it often is very convenient to transform that class into a `dict` (for example if you want to serialize it to JSON):

```
>>> attr.asdict(Coordinates(x=1, y=2))
{'y': 2, 'x': 1}
```

Some fields cannot or should not be transformed. For that, *attr.asdict()* offers a callback that decides whether an attribute should be included:

```
>>> @attr.s
... class UserList(object):
...     users = attr.ib()
>>> @attr.s
... class User(object):
...     email = attr.ib()
...     password = attr.ib()
>>> attr.asdict(UserList([User("jane@doe.invalid", "s33kred"),
...                       User("joe@doe.invalid", "p4ssw0rd")]),
...              filter=lambda attr, value: attr.name != "password")
{'users': [{'email': 'jane@doe.invalid'}, {'email': 'joe@doe.invalid'}]}
```

For the common case where you want to *include* or *exclude* certain types or attributes, `attrs` ships with a few helpers:

```
>>> @attr.s
... class User(object):
...     login = attr.ib()
...     password = attr.ib()
...     id = attr.ib()
>>> attr.asdict(User("jane", "s33kred", 42), filter=attr.filters.exclude(User.
↪password, int))
{'login': 'jane'}
>>> @attr.s
... class C(object):
...     x = attr.ib()
...     y = attr.ib()
...     z = attr.ib()
>>> attr.asdict(C("foo", "2", 3), filter=attr.filters.include(int, C.x))
{'x': 'foo', 'z': 3}
```

## Defaults

Sometimes you want to have default values for your initializer. And sometimes you even want mutable objects as default values (ever used accidentally `def f(arg=[])`?). `attrs` has you covered in both cases:

```
>>> import collections
>>> @attr.s
... class Connection(object):
...     socket = attr.ib()
...     @classmethod
...     def connect(cl, db_string):
...         # connect somehow to db_string
...         return cl(socket=42)
>>> @attr.s
... class ConnectionPool(object):
...     db_string = attr.ib()
...     pool = attr.ib(default=attr.Factory(collections.deque))
...     debug = attr.ib(default=False)
...     def get_connection(self):
```

```
...         try:
...             return self.pool.pop()
...         except IndexError:
...             if self.debug:
...                 print "New connection!"
...             return Connection.connect(self.db_string)
...     def free_connection(self, conn):
...         if self.debug:
...             print "Connection returned!"
...         self.pool.appendleft(conn)
...
>>> cp = ConnectionPool("postgres://localhost")
>>> cp
ConnectionPool(db_string='postgres://localhost', pool=deque([]), debug=False)
>>> conn = cp.get_connection()
>>> conn
Connection(socket=42)
>>> cp.free_connection(conn)
>>> cp
ConnectionPool(db_string='postgres://localhost', pool=deque([Connection(socket=42)]),␣
↪debug=False)
```

More information on why class methods for constructing objects are awesome can be found in this insightful blog post.

## Validators

Although your initializers should be as dumb as possible, it can come handy to do some kind of validation on the arguments. That's when *attr.ib()*'s `validator` argument comes into play. A validator is simply a callable that takes three arguments:

1. The *instance* that's being validated.
2. The *attribute* that it's validating
3. and finally the *value* that is passed for it.

If the value does not pass the validator's standards, it just raises an appropriate exception. Since the validator runs *after* the instance is initialized, you can refer to other attributes while validating :

```
>>> def x_smaller_than_y(instance, attribute, value):
...     if value >= instance.y:
...         raise ValueError("'x' has to be smaller than 'y'!")
>>> @attr.s
... class C(object):
...     x = attr.ib(validator=x_smaller_than_y)
...     y = attr.ib()
>>> C(x=3, y=4)
C(x=3, y=4)
>>> C(x=4, y=3)
Traceback (most recent call last):
   ...
ValueError: 'x' has to be smaller than 'y'!
```

`attrs` won't intercept your changes to those attributes but you can always call *attr.validate()* on any instance to verify, that it's still valid:

```
>>> i = C(4, 5)
>>> i.x = 5  # works, no magic here
>>> attr.validate(i)
Traceback (most recent call last):
   ...
ValueError: 'x' has to be smaller than 'y'!
```

attrs ships with a bunch of validators, make sure to *check them out* before writing your own:

```
>>> @attr.s
... class C(object):
...     x = attr.ib(validator=attr.validators.instance_of(int))
>>> C(42)
C(x=42)
>>> C("42")
Traceback (most recent call last):
   ...
TypeError: ("'x' must be <type 'int'> (got '42' that is a <type 'str'>).",␣
↪Attribute(name='x', default=NOTHING, factory=NOTHING, validator=<instance_of␣
↪validator for type <type 'int'>>), <type 'int'>, '42')
```

If you like zope.interface, attrs also comes with a *attr.validators.provides()* validator:

```
>>> import zope.interface
>>> class IFoo(zope.interface.Interface):
...     def f():
...         """A function called f."""
>>> @attr.s
... class C(object):
...     x = attr.ib(validator=attr.validators.provides(IFoo))
>>> C(x=object())
Traceback (most recent call last):
   ...
TypeError: ("'x' must provide <InterfaceClass __builtin__.IFoo> which <object object␣
↪at 0x10bafaaf0> doesn't.", Attribute(name='x', default=NOTHING, factory=NOTHING,␣
↪validator=<provides validator for interface <InterfaceClass __builtin__.IFoo>>),
↪<InterfaceClass __builtin__.IFoo>, <object object at 0x10bafaaf0>)
>>> @zope.interface.implementer(IFoo)
... @attr.s
... class Foo(object):
...     def f(self):
...         print("hello, world")
>>> C(Foo())
C(x=Foo())
```

You can also disable them globally:

```
>>> attr.set_run_validators(False)
>>> C(42)
C(x=42)
>>> attr.set_run_validators(True)
>>> C(42)
Traceback (most recent call last):
   ...
TypeError: ("'x' must provide <InterfaceClass __builtin__.IFoo> which 42 doesn't.",␣
↪Attribute(name='x', default=NOTHING, validator=<provides validator for interface
↪<InterfaceClass __builtin__.IFoo>>, repr=True, cmp=True, hash=True, init=True),
↪<InterfaceClass __builtin__.IFoo>, 42)
```

## Conversion

Attributes can have a `convert` function specified, which will be called with the attribute's passed-in value to get a new value to use. This can be useful for doing type-conversions on values that you don't want to force your callers to do.

```
>>> @attr.s
... class C(object):
...     x = attr.ib(convert=int)
>>> o = C("1")
>>> o.x
1
```

Converters are run *before* validators, so you can use validators to check the final form of the value.

```
>>> def validate_x(instance, attribute, value):
...     if value < 0:
...         raise ValueError("x must be be at least 0.")
>>> @attr.s
... class C(object):
...     x = attr.ib(convert=int, validator=validate_x)
>>> o = C("0")
>>> o.x
0
>>> C("-1")
Traceback (most recent call last):
    ...
ValueError: x must be be at least 0.
```

## Other Goodies

Do you like Rich Hickey? I'm glad to report that Clojure's core feature is part of `attrs`: assoc! I guess that means Clojure can be shut down now, sorry Rich!

```
>>> @attr.s
... class C(object):
...     x = attr.ib()
...     y = attr.ib()
>>> i1 = C(1, 2)
>>> i1
C(x=1, y=2)
>>> i2 = attr.assoc(i1, y=3)
>>> i2
C(x=1, y=3)
>>> i1 == i2
False
```

Sometimes you may want to create a class programmatically. `attrs` won't let you down:

```
>>> @attr.s
... class C1(object):
...     x = attr.ib()
...     y = attr.ib()
>>> C2 = attr.make_class("C2", ["x", "y"])
>>> attr.fields(C1) == attr.fields(C2)
True
```

You can still have power over the attributes if you pass a dictionary of name: `attr.ib` mappings and can pass arguments to `@attr.s`:

```
>>> C = attr.make_class("C", {"x": attr.ib(default=42),
...                           "y": attr.ib(default=attr.Factory(list))},
...                       repr=False)
>>> i = C()
>>> i  # no repr added!
<attr._make.C object at ...>
>>> i.x
42
>>> i.y
[]
```

Finally, you can exclude single attributes from certain methods:

```
>>> @attr.s
... class C(object):
...     user = attr.ib()
...     password = attr.ib(repr=False)
>>> C("me", "s3kr3t")
C(user='me')
```

# API

`attrs` works by decorating a class using *attr.s()* and then optionally defining attributes on the class using *attr.ib()*.

---

**Note:** When this documentation speaks about "`attrs` attributes" it means those attributes that are defined using *attr.ib()* in the class body.

---

What follows is the API explanation, if you'd like a more hands-on introduction, have a look at *Examples*.

## Core

`attr.s`(*these=None*, *repr_ns=None*, *repr=True*, *cmp=True*, *hash=True*, *init=True*)

A class decorator that adds dunder-methods according to the specified attributes using *attr.ib()* or the *these* argument.

**Parameters**

- **these** (class:*dict* of `str` to *attr.ib()*) – A dictionary of name to *attr.ib()* mappings. This is useful to avoid the definition of your attributes within the class body because you can't (e.g. if you want to add `__repr__` methods to Django models) or don't want to (e.g. if you want to use `properties`).

  If *these* is not *None*, the class body is *ignored*.

- **repr_ns** – When using nested classes, there's no way in Python 2 to automatically detect that. Therefore it's possible to set the namespace explicitly for a more meaningful `repr` output.

- **repr** (*bool*) – Create a `__repr__` method with a human readable represantation of `attrs` attributes..

- **cmp** (*bool*) – Create `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, and `__ge__` methods that compare the class as if it were a tuple of its `attrs` attributes. But the attributes are *only* compared, if the type of both classes is *identical*!
- **hash** (*bool*) – Create a `__hash__` method that returns the `hash()` of a tuple of all `attrs` attribute values.
- **init** (*bool*) – Create a `__init__` method that initializes the `attrs` attributes. Leading underscores are stripped for the argument name.

**Note:** `attrs` also comes with a less playful alias `attr.attributes`.

For example:

```
>>> import attr
>>> @attr.s
... class C(object):
...     _private = attr.ib()
>>> C(private=42)
C(_private=42)
>>> class D(object):
...     def __init__(self, x):
...         self.x = x
>>> D(1)
<D object at ...>
>>> D = attr.s(these={"x": attr.ib()}, init=False)(D)
>>> D(1)
D(x=1)
```

`attr.ib`(*default=NOTHING*, *validator=None*, *repr=True*, *cmp=True*, *hash=True*, *init=True*, *convert=None*)

Create a new attribute on a class.

**Warning:** Does *not* do anything unless the class is also decorated with *attr.s()*!

**Parameters**

- **default** (*Any value.*) – Value that is used if an `attrs`-generated `__init__` is used and no value is passed while instantiating. If the value an instance of *Factory*, it callable will be use to construct a new value (useful for mutable datatypes like lists or dicts).
- **validator** (*callable*) – `callable()` that is called by `attrs`-generated `__init__` methods after the instance has been initialized. They receive the initialized instance, the *Attribute*, and the passed value.

  The return value is *not* inspected so the validator has to throw an exception itself.

  They can be globally disabled and re-enabled using *get_run_validators()*.
- **repr** (*bool*) – Include this attribute in the generated `__repr__` method.
- **cmp** (*bool*) – Include this attribute in the generated comparison methods (`__eq__` et al).
- **hash** (*bool*) – Include this attribute in the generated `__hash__` method.
- **init** (*bool*) – Include this attribute in the generated `__init__` method.
- **convert** (*callable*) – `callable()` that is called by `attrs`-generated `__init__` methods to convert attribute's value to the desired format. It is given the passed-in value,

and the returned value will be used as the new value of the attribute. The value is converted before being passed to the validator, if any.

---

**Note:** `attrs` also comes with a less playful alias `attr.attr`.

---

**class** `attr.`**`Attribute`**(***kw*)

*Read-only* representation of an attribute.

**Attribute name** The name of the attribute.

Plus *all* arguments of *`attr.ib()`*.

Instances of this class are frequently used for introspection purposes like:

- Class attributes on `attrs`-decorated classes *after* `@attr.s` has been applied.
- *`fields()`* returns a tuple of them.
- Validators get them passed as the first argument.

**Warning:** You should never instantiate this class yourself!

```
>>> import attr
>>> @attr.s
... class C(object):
...     x = attr.ib()
>>> C.x
Attribute(name='x', default=NOTHING, validator=None, repr=True, cmp=True,␣
↪hash=True, init=True, convert=None)
```

`attr.`**`make_class`**(*name*, *attrs*, ***attributes_arguments*)

A quick way to create a new class called *name* with *attrs*.

**Parameters**

- **`name`** (*`str`*) – The name for the new class.
- **`attrs`** (`list` or `dict`) – A list of names or a dictionary of mappings of names to attributes.
- **`attributes_arguments`** – Passed unmodified to *`attr.s()`*.

**Returns** A new class with *attrs*.

**Return type** type

This is handy if you want to programmatically create classes.

For example:

```
>>> C1 = attr.make_class("C1", ["x", "y"])
>>> C1(1, 2)
C1(x=1, y=2)
>>> C2 = attr.make_class("C2", {"x": attr.ib(default=42),
...                             "y": attr.ib(default=attr.Factory(list))})
>>> C2()
C2(x=42, y=[])
```

**class** attr.**Factory**(*factory*)
Stores a factory callable.

If passed as the default value to *attr.ib()*, the factory is used to generate a new value.

For example:

```
>>> @attr.s
... class C(object):
...     x = attr.ib(default=attr.Factory(list))
>>> C()
C(x=[])
```

## Helpers

attrs comes with a bunch of helper methods that make the work with it easier:

attr.**fields**(*cl*)
Returns the tuple of attrs attributes for a class.

**Parameters** **cl** (*class*) – Class to introspect.

**Raises**

- **TypeError** – If *cl* is not a class.
- **ValueError** – If *cl* is not an attrs class.

**Return type** tuple of *attr.Attribute*

For example:

```
>>> @attr.s
... class C(object):
...     x = attr.ib()
...     y = attr.ib()
>>> attr.fields(C)
(Attribute(name='x', default=NOTHING, validator=None, repr=True, cmp=True,␣
↪hash=True, init=True, convert=None), Attribute(name='y', default=NOTHING,␣
↪validator=None, repr=True, cmp=True, hash=True, init=True, convert=None))
```

attr.**has**(*cl*)
Check whether *cl* is a class with attrs attributes.

**Parameters** **cl** (*type*) – Class to introspect.

**Raises** **TypeError** – If *cl* is not a class.

**Return type** bool

For example:

```
>>> @attr.s
... class C(object):
...     pass
>>> attr.has(C)
True
>>> attr.has(object)
False
```

attr.**asdict**(*inst*, *recurse=True*, *filter=None*)
Return the attrs attribute values of *i* as a dict. Optionally recurse into other attrs-decorated classes.

**Parameters**

- **inst** – Instance of a attrs-decorated class.
- **recurse** (*bool*) – Recurse into classes that are also attrs-decorated.
- **filter** – A callable whose return code deteremines whether an attribute or element is included (True) or dropped (False). Is called with the *attr.Attribute* as the first argument and the value as the second argument.

**Return type** dict

For example:

```
>>> @attr.s
... class C(object):
...     x = attr.ib()
...     y = attr.ib()
>>> attr.asdict(C(1, C(2, 3)))
{'y': {'y': 3, 'x': 2}, 'x': 1}
```

attrs comes with some handy helpers for filtering:

attr.filters.**include**(**what*)
Whitelist *what*.

**Parameters** **what** (list of type or *attr.Attribute* s.) – What to whitelist.

**Return type** callable

attr.filters.**exclude**(**what*)
Blacklist *what*.

**Parameters** **what** (list of classes or *attr.Attribute* s.) – What to blacklist.

**Return type** callable

attr.**assoc**(*inst*, ***changes*)
Copy *inst* and apply *changes*.

**Parameters**

- **inst** – Instance of a class with attrs attributes.
- **changes** – Keyword changes in the new copy.

**Returns** A copy of inst with *changes* incorporated.

For example:

```
>>> @attr.s
... class C(object):
...     x = attr.ib()
...     y = attr.ib()
>>> i1 = C(1, 2)
>>> i1
C(x=1, y=2)
>>> i2 = attr.assoc(i1, y=3)
>>> i2
C(x=1, y=3)
>>> i1 == i2
False
```

attr.**validate**(*inst*)
Validate all attributes on *inst* that have a validator.

Leaves all exceptions through.

**Parameters** **inst** – Instance of a class with attrs attributes.

For example:

```
>>> @attr.s
... class C(object):
...     x = attr.ib(validator=attr.validators.instance_of(int))
>>> i = C(1)
>>> i.x = "1"
>>> attr.validate(i)
Traceback (most recent call last):
   ...
TypeError: ("'x' must be <type 'int'> (got '1' that is a <type 'str'>).",␣
↪Attribute(name='x', default=NOTHING, validator=<instance_of validator for type
↪<type 'int'>>, repr=True, cmp=True, hash=True, init=True), <type 'int'>, '1')
```

Validators can be globally disabled if you want to run them only in development and tests but not in production because you fear their performance impact:

attr.**set_run_validators**(*run*)
Set whether or not validators are run. By default, they are run.

attr.**get_run_validators**()
Return whether or not validators are run.

## Validators

attrs comes with some common validators within the attrs.validators module:

attr.validators.**instance_of**(*type*)
A validator that raises a TypeError if the initializer is called with a wrong type for this particular attribute (checks are perfomed using isinstance()).

**Parameters** **type** (*type*) – The type to check for.

The TypeError is raised with a human readable error message, the attribute (of type *attr.Attribute*), the expected type and the value it got.

For example:

```
>>> @attr.s
... class C(object):
...     x = attr.ib(validator=attr.validators.instance_of(int))
>>> C(42)
C(x=42)
>>> C("42")
Traceback (most recent call last):
   ...
TypeError: ("'x' must be <type 'int'> (got '42' that is a <type 'str'>).",␣
↪Attribute(name='x', default=NOTHING, validator=<instance_of validator for type
↪<type 'int'>>), <type 'int'>, '42')
>>> C(None)
Traceback (most recent call last):
```

```
   ...
TypeError: ("'x' must be <type 'int'> (got None that is a <type 'NoneType'>).",␣
↪Attribute(name='x', default=NOTHING, validator=<instance_of validator for type
↪<type 'int'>>, repr=True, cmp=True, hash=True, init=True), <type 'int'>, None)
```

attr.validators.**provides**(*interface*)

A validator that raises a `TypeError` if the initializer is called with an object that does not provide the requested *interface* (checks are performed using `interface.providedBy(value)` (see zope.interface).

**Parameters** **interface** (*zope.interface.Interface*) – The interface to check for.

The `TypeError` is raised with a human readable error message, the attribute (of type *attr.Attribute*), the expected interface, and the value it got.

attr.validators.**optional**(*validator*)

A validator that makes an attribute optional. An optional attribute is one which can be set to `None` in addition to satisfying the requirements of the sub-validator.

**Parameters** **validator** – A validator that is used for non-`None` values.

For example:

```
>>> @attr.s
... class C(object):
...     x = attr.ib(validator=attr.validators.optional(attr.validators.instance_
↪of(int)))
>>> C(42)
C(x=42)
>>> C("42")
Traceback (most recent call last):
   ...
TypeError: ("'x' must be <type 'int'> (got '42' that is a <type 'str'>).",␣
↪Attribute(name='x', default=NOTHING, validator=<instance_of validator for type
↪<type 'int'>>), <type 'int'>, '42')
>>> C(None)
C(x=None)
```

## Extending

Each `attrs`-decorated class has a `__attrs_attrs__` class attribute. It is a tuple of *attr.Attribute* carrying meta-data about each attribute.

So it is fairly simple to build your own decorators on top of `attrs`:

```
>>> import attr
>>> def print_attrs(cl):
...     print cl.__attrs_attrs__
>>> @print_attrs
... @attr.s
... class C(object):
...     a = attr.ib()
(Attribute(name='a', default=NOTHING, validator=None, repr=True, cmp=True, hash=True,␣
↪init=True, convert=None),)
```

**Warning:** The *attr.s()* decorator **must** be applied first because it puts `__attrs_attrs__` in place! That means that is has to come *after* your decorator because:

```
@a
@b
def f():
   pass
```

is just syntactic sugar for:

```
def original_f():
   pass

f = a(b(original_f))
```

CHAPTER 2

# Project Information

## Backward Compatibility

`attrs` has a very strong backward compatibility policy that is inspired by the one of the Twisted framework.

Put simply, you shouldn't ever be afraid to upgrade `attrs` if you're using its public APIs. If there will ever be need to break compatibility, it will be announced in the *Changelog*, raise deprecation warning for a year before it's finally really broken.

> **Warning:** The structure of the `attr.Attribute` class is exempted from this rule. It *will* change in the future since it should be considered read-only, that shouldn't matter.
>
> However if you intend to build extensions on top of `attrs` you have to anticipate that.

## License and Hall of Fame

`attrs` is licensed under the MIT license. The full license text can be also found in the source code repository.

### Authors

`attrs` is written and maintained by Hynek Schlawack.

The development is kindly supported by Variomedia AG.

A full list of contributors can be found in GitHub's overview.

It's the spiritual successor of characteristic and aspires to fix some of it clunkiness and unfortunate decisions. Both were inspired by Twisted's FancyEqMixin but both are implemented using class decorators because sub-classing is bad for you, m'kay?

## How To Contribute

Every open source project lives from the generous help by contributors that sacrifice their time and `attrs` is no different.

Here are a few guidelines to get you started:

- Try to limit each pull request to one change only.
- To run the test suite, all you need is a recent tox. It will ensure the test suite runs with all dependencies against all Python versions just as it will on Travis CI. If you lack some Python version, you can can always limit the environments like `tox -e py27,py35` (in that case you may want to look into pyenv that makes it very easy to install many different Python versions in parallel).
- Make sure your changes pass our CI. You won't get any feedback until it's green unless you ask for it.
- If your change is noteworthy, add an entry to the changelog. Use present tense, semantic newlines, and add link to your pull request.
- No contribution is too small; please submit as many fixes for typos and grammar bloopers as you can!
- Don't break backward compatibility.
- *Always* add tests and docs for your code. This is a hard rule; patches with missing tests or documentation won't be merged.
- Write good test docstrings.
- Obey PEP 8 and PEP 257.
- If you address review feedback, make sure to bump the pull request. Maintainers don't receive notifications if you push new commits.

Please note that this project is released with a Contributor Code of Conduct. By participating in this project you agree to abide by its terms. Please report any harm to Hynek Schlawack in any way you find appropriate.

Thank you for considering to contribute to `attrs`!

## Changelog

Versions are year-based with a strict backwards compatibility policy. The third digit is only for regressions.

### 15.2.0 (2015-12-08)

#### Changes:

- Add a `convert` argument to `attr.ib`, which allows specifying a function to run on arguments. This allows for simple type conversions, e.g. with `attr.ib(convert=int)`. [26]
- Speed up object creation when attribute validators are used. [28]

### 15.1.0 (2015-08-20)

#### Changes:

- Add `attr.validators.optional` that wraps other validators allowing attributes to be `None`. [16]

- Fix multi-level inheritance. [24]
- Fix `__repr__` to work for non-redecorated subclasses. [20]

## 15.0.0 (2015-04-15)

### Changes:

Initial release.

CHAPTER 3

# Indices and tables

- genindex
- search

# Index