
attrs Documentation

Release 16.3.0

Hynek Schlawack

Sep 25, 2017

Contents

1	Testimonials	3
2	User's Guide	5
2.1	Overview	5
2.2	Why not...	7
2.3	Examples	10
2.4	API	20
2.5	Extending	28
2.6	How Does It Work?	29
3	Project Information	31
3.1	License and Credits	31
3.2	Backward Compatibility	31
3.3	How To Contribute	32
3.4	Contributor Covenant Code of Conduct	33
3.5	Changelog	35
4	Indices and tables	39

Release v16.3.0 (*What's new?*).

`attrs` is the Python package that will bring back the **joy** of **writing classes** by relieving you from the drudgery of implementing object protocols (aka **dunder** methods).

Its main goal is to help you to write **concise** and **correct** software without slowing down your code.

If you want to know how this looks like, jump right into *Overview*. If you really want to see `attrs` in action, *Examples* will give you a comprehensive rundown of its features. If you're skeptical and want to know how it works first, check out "*How Does It Work?*".

CHAPTER 1

Testimonials

I'm looking forward to is being able to program in Python-with-attrs everywhere. It exerts a subtle, but positive, design influence in all the codebases I've see it used in.

—Glyph Lefkowitz, inventor of Twisted and Software Developer at Rackspace in [The One Python Library Everyone Needs](#)

I'm increasingly digging your attr.ocity. Good job!

—Łukasz Langa, prolific CPython core developer and Production Engineer at Facebook

Overview

In order to fulfill its ambitious goal of bringing back the joy to writing classes, it gives you a class decorator and a way to declaratively define the attributes on that class:

```
>>> import attr
>>> @attr.s
... class C(object):
...     x = attr.ib(default=42)
...     y = attr.ib(default=attr.Factory(list))
...
...     def hard_math(self, z):
...         return self.x * self.y * z
>>> i = C(x=1, y=2)
>>> i
C(x=1, y=2)
>>> i.hard_math(3)
6
>>> 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')
```

After *declaring* your attributes `attrs` gives you:

- a concise and explicit overview of the class's attributes,

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

without writing dull boilerplate code again and again and *without* runtime performance penalties.

This gives you the power to use actual classes with actual types in your code instead of confusing `tuples` or confusingly behaving `namedtuples`. Which in turn encourages you to write *small classes* that do *one thing well*. Never again violate the *single responsibility principle* just because implementing `__init__` et al is a painful drag.

Philosophy

It's about regular classes. `attrs` for creating well-behaved classes with a type, attributes, methods, and everything that comes with a class. It can be used for data-only containers like `namedtuples` or `types.SimpleNamespace` but they're just a sub-genre of what `attrs` is good for.

The class belongs to the users. You define a class and `attrs` adds static methods to that class based on the attributes you declare. The end. It doesn't add meta classes. It doesn't add classes you've never heard of to your inheritance tree. An `attrs` class in runtime is indistinguishable from a regular class: because it *is* a regular class with a few boilerplate-y methods attached.

Be light on API impact. As convenient as it seems at first, `attrs` will *not* tack on any methods to your classes save the dunder ones. Hence all the useful *tools* that come with `attrs` live in functions that operate on top of instances. Since they take an `attrs` instance as their first argument, you can attach them to your classes with one line of code.

Performance matters. `attrs` runtime impact is very close to zero because all the work is done when the class is defined. Once you're instantiating it, `attrs` is out of the picture completely.

No surprises. `attrs` creates classes that arguably work the way a Python beginner would reasonably expect them to work. It doesn't try to guess what you mean because explicit is better than implicit. It doesn't try to be clever because software shouldn't be clever.

Check out *How Does It Work?* if you'd like to know how it achieves all of the above.

What `attrs` Is Not

`attrs` does *not* invent some kind of magic system that pulls classes out of its hat using meta classes, runtime introspection, and shaky interdependencies.

All `attrs` does is take your declaration, write dunder methods based on that information, and attach them to your class. It does *nothing* dynamic at runtime, hence zero runtime overhead. It's still *your* class. Do with it as you please.

On the `attr.s` and `attr.ib` Names

The `attr.s` decorator and the `attr.ib` function aren't any obscure abbreviations. They are a *concise* and highly *readable* way to write `attrs` and `attrib` with an *explicit namespace*.

At first, some people have a negative gut reaction to that; resembling the reactions to Python's significant whitespace. And as with that, once one gets used to it, the readability and explicitness of that API prevails and delights.

For those who can't swallow that API at all, `attrs` comes with serious business aliases: `attr.attrs` and `attr.attrib`.

Therefore, the following class definition is identical to the previous one:

```
>>> from attr import attrs, attrib, Factory
>>> @attrs
... class C(object):
...     x = attrib(default=42)
...     y = attrib(default=Factory(list))
>>> C()
C(x=42, y=[])
```

Use whichever variant fits your taste better.

Why not...

If you'd like third party's account why `attrs` is great, have a look at Glyph's [The One Python Library Everyone Needs!](#)

...tuples?

Readability

What makes more sense while debugging:

```
Point(x=1, y=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 than 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 require less typing as compared with 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 a `namedtuple` is *explicitly* intended to behave like a tuple:

```
>>> 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 and default values.

...dicts?

Dictionaries are not for fixed fields.

If you have a dict, it maps something to something else. You should be able to add and remove values.

Objects, on the other hand, are supposed to have specific fields of specific types, because their methods have strong expectations of what those fields and types are.

`attrs` lets you be specific about those expectations; a dictionary does not. It gives you a named entity (the class) in your code, which lets you explain in other places whether you take a parameter of that class or return a value of that class.

In other words: if your dict has a fixed and known set of keys, it is an object, not a hash.

...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.

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 loong.

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` is 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()
```

By default, all features are added, so you immediately have 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 for both positional and keyword arguments.

If playful naming turns you off, `attrs` comes with serious business aliases:

```
>>> from attr import attrs, attrib
>>> @attrs
... class SeriousCoordinates(object):
...     x = attrib()
...     y = attrib()
>>> 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)
```

If you want to initialize your private attributes yourself, you can do that too:

```
>>> @attr.s
... class C(object):
...     _x = attr.ib(init=False, default=42)
>>> C()
C(_x=42)
```

```
>>> C(23)
Traceback (most recent call last):
...
TypeError: __init__() takes exactly 1 argument (2 given)
```

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
```

Subclassing 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 it overrides the implicit detection.

Converting to Collections Types

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(attr.fields(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, attr.fields(C).x))
{'z': 3, 'x': 'foo'}
```

Other times, all you want is a tuple and `attrs` won't let you down:

```
>>> import sqlite3
>>> import attr
>>> @attr.s
... class Foo:
...     a = attr.ib()
...     b = attr.ib()
>>> foo = Foo(2, 3)
>>> with sqlite3.connect(":memory:") as conn:
...     c = conn.cursor()
...     c.execute("CREATE TABLE foo (x INTEGER PRIMARY KEY ASC, y)")
...     c.execute("INSERT INTO foo VALUES (?, ?)", attr.astuple(foo))
...     foo2 = Foo(*c.execute("SELECT x, y FROM foo").fetchone())
<sqlite3.Cursor object at ...>
<sqlite3.Cursor object at ...>
>>> foo == foo2
True
```

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(cls, db_string):
...         # ... connect somehow to db_string ...
...         return cls(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 in 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, and finally
3. 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 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 at least 0.
```

Metadata

All `attrs` attributes may include arbitrary metadata in the form of a read-only dictionary.

```
>>> @attr.s
... class C(object):
...     x = attr.ib(metadata={'my_metadata': 1})
>>> attr.fields(C).x.metadata
mappingproxy({'my_metadata': 1})
>>> attr.fields(C).x.metadata['my_metadata']
1
```

Metadata is not used by `attrs`, and is meant to enable rich functionality in third-party libraries. The metadata dictionary follows the normal dictionary rules: keys need to be hashable, and both keys and values are recommended to be immutable.

If you're the author of a third-party library with `attrs` integration, please see [Extending Metadata](#).

Slots

By default, instances of classes have a dictionary for attribute storage. This wastes space for objects having very few data attributes. The space consumption can become significant when creating large numbers of instances.

Normal Python classes can avoid using a separate dictionary for each instance of a class by [defining](#) `__slots__`. For `attrs` classes it's enough to set `slots=True`:

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

Note: `attrs` slot classes can inherit from other classes just like non-slot classes, but some of the benefits of slot classes are lost if you do that. If you must inherit from other classes, try to inherit only from other slot classes.

Slot classes are a little different than ordinary, dictionary-backed classes:

- Assigning to a non-existent attribute of an instance will result in an `AttributeError` being raised. Depending on your needs, this might be a good thing since it will let you catch typos early. This is not the case if your class inherits from any non-slot classes.

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

```
...     y = attr.ib()
...
>>> c = Coordinates(x=1, y=2)
>>> c.z = 3
Traceback (most recent call last):
...
AttributeError: 'Coordinates' object has no attribute 'z'
```

- Slot classes cannot share attribute names with their instances, while non-slot classes can. The following behaves differently if slot classes are used:

```
>>> @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, metadata=mappingproxy({}))
>>> @attr.s(slots=True)
... class C(object):
...     x = attr.ib()
>>> C.x
<member 'x' of 'C' objects>
```

- Since non-slot classes cannot be turned into slot classes after they have been created, `attr.s(..., slots=True)` will *replace* the class it is applied to with a copy. In almost all cases this isn't a problem, but we mention it for the sake of completeness.
- Using `pickle` with slot classes requires pickle protocol 2 or greater. Python 2 uses protocol 0 by default so the protocol needs to be specified. Python 3 uses protocol 3 by default. You can support protocol 0 and 1 by implementing `__getstate__` and `__setstate__` methods yourself. Those methods are created for frozen slot classes because they won't pickle otherwise. Think twice before using `pickle` though.

All in all, setting `slots=True` is usually a very good idea.

Immutability

Sometimes you have instances that shouldn't be changed after instantiation. Immutability is especially popular in functional programming and is generally a very good thing. If you'd like to enforce it, `attrs` will try to help:

```
>>> @attr.s(frozen=True)
... class C(object):
...     x = attr.ib()
>>> i = C(1)
>>> i.x = 2
Traceback (most recent call last):
...
attr.exceptions.FrozenInstanceError: can't set attribute
>>> i.x
1
```

Please note that true immutability is impossible in Python but it will *get* you 99% there. By themselves, immutable classes are useful for long-lived objects that should never change; like configurations for example.

In order to use them in regular program flow, you'll need a way to easily create new instances with changed attributes. In Clojure that function is called `assoc` and `attrs` shamelessly imitates it: `attr.assoc()`:

```
>>> @attr.s(frozen=True)
... 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
```

Other Goodies

Sometimes you may want to create a class programmatically. `attrs` won't let you down and gives you `attr.make_class()`:

```
>>> @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
[]
```

Sometimes, you want to have your class's `__init__` method do more than just the initialization, validation, etc. that gets done for you automatically when using `@attr.s`. To do this, just define a `__attrs_post_init__` method in your class. It will get called at the end of the generated `__init__` method.

```
>>> @attr.s
... class C(object):
...     x = attr.ib()
...     y = attr.ib()
...     z = attr.ib(init=False)
...
...     def __attrs_post_init__(self):
...         self.z = self.x + self.y
>>> obj = C(x=1, y=2)
>>> obj
C(x=1, y=2, z=3)
```

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, slots=False, frozen=False, str=False)`
A class decorator that adds dunder-methods according to the specified attributes using `attr.ib()` or the *these* argument.

Parameters

- **these** (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** (str) – 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 representation of attrs attributes..
- **str** (bool) – Create a `__str__` method that is identical to `__repr__`. This is usually not necessary except for `Exceptions`.
- **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. If a `__attrs_post_init__` method exists on the class, it will be called after the class is fully initialized.

- **slots** (*bool*) – Create a *slots*-style class that's more memory-efficient. See *Slots* for further ramifications.
- **frozen** (*bool*) – Make instances immutable after initialization. If someone attempts to modify a frozen instance, `attr.exceptions.FrozenInstanceError` is raised.

Please note:

1. This is achieved by installing a custom `__setattr__` method on your class so you can't implement an own one.
2. True immutability is impossible in Python.
3. This *does* have a minor a runtime performance *impact* when initializing new instances. In other words: `__init__` is slightly slower with `frozen=True`.

New in version 16.0.0: *slots*

New in version 16.1.0: *frozen*

New in version 16.3.0: *str*, and support for `__attrs_post_init__`.

Note: `attrs` also comes with a serious business alias `attr.attrs`.

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*, *metadata={}*)

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.*) – A value that is used if an `attrs`-generated `__init__` is used and no value is passed while instantiating or the attribute is excluded using `init=False`.

If the value is an instance of *Factory*, its callable will be used to construct a new value (useful for mutable datatypes like lists or dicts).

If a default is not set (or set manually to `attr.NOTHING`), a value *must* be supplied when instantiating; otherwise a `TypeError` will be raised.

- **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. It is possible to set this to `False` and set a default value. In that case this attributed is unconditionally initialized with the specified default value or factory.
- **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.
- **metadata** – An arbitrary mapping, to be used by third-party components.

Note: `attrs` also comes with a serious business alias `attr.attrib`.

class `attr.Attribute` (*name, default, validator, repr, cmp, hash, init, convert=None, metadata=None*)
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:

- `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, metadata=mappingproxy({}))
```

`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=[])
```

exception `attr.exceptions.FrozenInstanceError`

A frozen/immutable instance has been attempted to be modified.

It mirrors the behavior of `namedtuples` by using the same error message and subclassing `AttributeError`.

New in version 16.1.0.

exception `attr.exceptions.AttrsAttributeNotFoundError`

An `attrs` function couldn't find an attribute that the user asked for.

New in version 16.2.0.

exception `attr.exceptions.NotAnAttrsClassError`

A non-`attrs` class has been passed into an `attrs` function.

New in version 16.2.0.

Helpers

`attrs` comes with a bunch of helper methods that make working with it easier:

`attr.fields(cls)`

Returns the tuple of `attrs` attributes for a class.

The tuple also allows accessing the fields by their names (see below for examples).

Parameters `cls (type)` – Class to introspect.

Raises

- `TypeError` – If `cls` is not a class.

- `attr.exceptions.NotAnAttrsClassError` – If `cls` is not an `attrs` class.

Return type tuple (with name accessors) of `attr.Attribute`

Changed in version 16.2.0: Returned tuple allows accessing the fields by name.

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, metadata=mappingproxy({})), Attribute(name=
↳ 'y', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True,
↳ convert=None, metadata=mappingproxy({})))
>>> attr.fields(C)[1]
Attribute(name='y', default=NOTHING, validator=None, repr=True, cmp=True,
↳ hash=True, init=True, convert=None, metadata=mappingproxy({}))
>>> attr.fields(C).y is attr.fields(C)[1]
True
```

`attr.has(cls)`

Check whether `cls` is a class with `attrs` attributes.

Parameters `cls` (*type*) – Class to introspect.

Raises `TypeError` – If `cls` 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, dict_factory=<class 'dict'>, re-
tain_collection_types=False)`

Return the `attrs` attribute values of `inst` as a dict.

Optionally recurse into other `attrs`-decorated classes.

Parameters

- **inst** – Instance of an `attrs`-decorated class.
- **recurse** (*bool*) – Recurse into classes that are also `attrs`-decorated.
- **filter** (*callable*) – A callable whose return code determines 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.
- **dict_factory** (*callable*) – A callable to produce dictionaries from. For example, to produce ordered dictionaries instead of normal Python dictionaries, pass in `collections.OrderedDict`.

- **retain_collection_types** (*bool*) – Do not convert to `list` when encountering an attribute whose type is `tuple` or `set`. Only meaningful if `recurse` is `True`.

Return type return type of `dict_factory`

Raises `attr.exceptions.NotAnAttrsClassError` – If `cls` is not an `attrs` class.

New in version 16.0.0: `dict_factory`

New in version 16.1.0: `retain_collection_types`

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}
```

`attr.astuple` (*inst*, *recurse=True*, *filter=None*, *tuple_factory=<class 'tuple'>*, *retain_collection_types=False*)

Return the `attrs` attribute values of *inst* as a tuple.

Optionally recurse into other `attrs`-decorated classes.

Parameters

- **inst** – Instance of an `attrs`-decorated class.
- **recurse** (*bool*) – Recurse into classes that are also `attrs`-decorated.
- **filter** (*callable*) – A callable whose return code determines 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.
- **tuple_factory** (*callable*) – A callable to produce tuples from. For example, to produce lists instead of tuples.
- **retain_collection_types** (*bool*) – Do not convert to `list` or `dict` when encountering an attribute which type is `tuple`, `dict` or `set`. Only meaningful if `recurse` is `True`.

Return type return type of `tuple_factory`

Raises `attr.exceptions.NotAnAttrsClassError` – If `cls` is not an `attrs` class.

New in version 16.2.0.

For example:

```
>>> @attr.s
... class C(object):
...     x = attr.ib()
...     y = attr.ib()
>>> attr.astuple(C(1, 2))
(1, 2)
```

`attrs` includes some handy helpers for filtering:

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

Whitelist *what*.

Parameters *what* (*list* of *type* or `attr.Attributes`) – What to whitelist.

Return type callable

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

Parameters *what* (list of classes or `attr.Attributes`.) – What to blacklist.

Return type callable

See *Converting to Collections Types* for examples.

`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.

Raises

- `attr.exceptions.AttrsAttributeNotFoundError` – If *attr_name* couldn't be found on *cls*.
- `attr.exceptions.NotAnAttrsClassError` – If *cls* is not an `attrs` class.

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 in 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 performed using `isinstance()` therefore it's also valid to pass a tuple of types).

Parameters `type` (*type or tuple of types*) – 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)
```

Deprecated APIs

The serious business aliases used to be called `attr.attributes` and `attr.attr`. There are no plans to remove them but they shouldn't be used in new code.

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(cls):
...     print(cls.__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, metadata=mappingproxy({})),)
```

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


Metadata

If you're the author of a third-party library with `attrs` integration, you may want to take advantage of attribute metadata.

Here are some tips for effective use of metadata:

- Try making your metadata keys and values immutable. This keeps the entire `Attribute` instances immutable too.
- To avoid metadata key collisions, consider exposing your metadata keys from your modules.:

```
from mylib import MY_METADATA_KEY

@attr.s
class C(object):
    x = attr.ib(metadata={MY_METADATA_KEY: 1})
```

Metadata should be composable, so consider supporting this approach even if you decide implementing your metadata in one of the following ways.

- Expose `attr.ib` wrappers for your specific metadata. This is a more graceful approach if your users don't require metadata from other libraries.

```
>>> MY_TYPE_METADATA = '__my_type_metadata'
>>>
>>> def typed(cls, default=attr.NOTHING, validator=None, repr=True, cmp=True,
↳ hash=True, init=True, convert=None, metadata={}):
...     metadata = dict() if not metadata else metadata
...     metadata[MY_TYPE_METADATA] = cls
...     return attr.ib(default, validator, repr, cmp, hash, init, convert,
↳ metadata)
>>>
>>> @attr.s
... class C(object):
...     x = typed(int, default=1, init=False)
>>> attr.fields(C).x.metadata[MY_TYPE_METADATA]
<class 'int'>
```

How Does It Work?

Boilerplate

`attrs` certainly isn't the first library that aims to simplify class definition in Python. But its **declarative** approach combined with **no runtime overhead** lets it stand out.

Once you apply the `@attr.s` decorator to a class, `attrs` searches the class object for instances of `attr.ibs`. Internally they're a representation of the data passed into `attr.ib` along with a counter to preserve the order of the attributes.

In order to ensure that sub-classing works as you'd expect it to work, `attrs` also walks the class hierarchy and collects the attributes of all super-classes. Please note that `attrs` does *not* call `super()` *ever*. It will write dunder methods to work on *all* of those attributes which also has performance benefits due to less function calls.

Once `attrs` knows what attributes it has to work on, it writes the requested dunder methods and attaches them to your class. To be very clear: if you define a class with a single attribute without a default value, the generated `__init__` will look *exactly* how you'd expect:

```
>>> import attr, inspect
>>> @attr.s
... class C:
...     x = attr.ib()
>>> print(inspect.getsource(C.__init__))
def __init__(self, x):
    self.x = x
```

No magic, no meta programming, no expensive introspection at runtime.

Everything until this point happens exactly *once* when the class is defined. As soon as a class is done, it's done. And it's just a regular Python class like any other, except for a single `__attrs_attrs__` attribute that can be used for introspection or for writing your own tools and decorators on top of `attrs` (like `attr.asdict()`).

And once you start instantiating your classes, `attrs` is out of your way completely.

This **static** approach was very much a design goal of `attrs` and what I strongly believe makes it distinct.

Immutability

In order to give you immutability, `attrs` will attach a `__setattr__` method to your class that raises a `attr.exceptions.FrozenInstanceError` whenever anyone tries to set an attribute.

In order to circumvent that ourselves in `__init__`, `attrs` uses (an aggressively cached) `object.__setattr__()` to set your attributes. This is (still) slower than a plain assignment:

```
$ pyperf timeit --rigorous \
-s "import attr; C = attr.make_class('C', ['x', 'y', 'z'], slots=True)" \
  "C(1, 2, 3)"
.....
Median +- std dev: 378 ns +- 12 ns

$ pyperf timeit --rigorous \
-s "import attr; C = attr.make_class('C', ['x', 'y', 'z'], slots=True,
↪frozen=True)" \
  "C(1, 2, 3)"
.....
Median +- std dev: 676 ns +- 16 ns
```

So on my notebook the difference is about 300 nanoseconds (1 second is 1,000,000,000 nanoseconds). It's certainly something you'll feel in a hot loop but shouldn't matter in normal code. Pick what's more important to you.

Once constructed, frozen instances differ in no way from regular ones except that you cannot change its attributes.

CHAPTER 3

Project Information

`attrs` is released under the [MIT](#) license, its documentation lives at [Read the Docs](#), the code on [GitHub](#), and the latest release on [PyPI](#). It's rigorously tested on Python 2.7, 3.4+, and PyPy.

License and Credits

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

Credits

`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 its 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?

Backward Compatibility

`attrs` has a very strong backward compatibility policy that is inspired by the policy of the [Twisted framework](#).

Put simply, you shouldn't ever be afraid to upgrade `attrs` if you're only using its public APIs. If there will ever be a need to break compatibility, it will be announced in the [Changelog](#) and raise a `DeprecationWarning` for a year (if possible) before it's finally really broken.

Warning: The structure of the `attr.Attribute` class is exempt from this rule. It *will* change in the future, but 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.

How To Contribute

First off, thank you for considering contributing to `attrs`! It's people like *you* who make it is such a great tool for everyone.

Here are a few guidelines to get you started (but don't be afraid to open half-finished PRs and ask questions if something is unclear!):

Workflow

- No contribution is too small! Please submit as many fixes for typos and grammar bloopers as you can!
- Try to limit each pull request to *one* change only.
- *Always* add tests and docs for your code. This is a hard rule; patches with missing tests or documentation can't be accepted.
- Make sure your changes pass our [CI](#). You won't get any feedback until it's green unless you ask for it.
- Once you've addressed review feedback, make sure to bump the pull request with a short note. Maintainers don't receive notifications when you push new commits.
- Don't break [backward compatibility](#).

Code

- Obey [PEP 8](#) and [PEP 257](#). We use the `"""`-on-separate-lines style for docstrings:

```
def func(x):  
    """  
    Does something.  
  
    :param str x: A very important parameter.  
  
    :rtype: str  
    """
```

- If you add or change public APIs, tag the docstring using `.. versionadded:: 16.0.0 WHAT` or `.. versionchanged:: 16.2.0 WHAT`.
- Prefer double quotes (`"`) over single quotes (`'`) unless the string contains double quotes itself.

Tests

- Write your asserts as `expected == actual` to line them up nicely:

```
x = f()

assert 42 == x.some_attribute
assert "foo" == x._a_private_attribute
```

- 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 versions, you can always limit the environments like `tox -e py27,py35` (in that case you may want to look into `pyenv`, which makes it very easy to install many different Python versions in parallel).
- Write [good test docstrings](#).
- To ensure new features work well with the rest of the system, they should be also added to our [Hypothesis](#) testing strategy which you find in `tests/util.py`.

Documentation

- Use [semantic newlines](#) in `reStructuredText` files (files ending in `.rst`):

```
This is a sentence.
This is another sentence.
```

- If you add a new feature, demonstrate its awesomeness in the [examples page](#)!
- If your change is noteworthy, add an entry to the [changelog](#). Use present tense, [semantic newlines](#), and add a link to your pull request:

```
- Add awesome new feature.
  The feature really is awesome.
  [#1 <https://github.com/hynek/attrs/pull/1>`_]]
- Fix nasty bug.
  The bug really was nasty.
  [#2 <https://github.com/hynek/attrs/pull/2>`_]]
```

Again, this list is mainly to help you to get started by codifying tribal knowledge and expectations. If something is unclear, feel free to ask for help!

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 contributing to `attrs`!

Contributor Covenant Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to make participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at hs@ox.cx. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the [Contributor Covenant](http://contributor-covenant.org/version/1/4), version 1.4, available at <http://contributor-covenant.org/version/1/4>.

Changelog

Versions follow [CalVer](#) with a strict backwards compatibility policy. The third digit is only for regressions.

16.3.0 (2016-11-24)

Changes:

- Attributes now can have user-defined metadata which greatly improves `attrs`'s extensibility. [#96](#)
- Allow for a `__attrs_post_init__` method that – if defined – will get called at the end of the `attrs`-generated `__init__` method. [#111](#)
- Add `@attr.s(str=True)` that will optionally create a `__str__` method that is identical to `__repr__`. This is mainly useful with `Exceptions` and other classes that rely on a useful `__str__` implementation but overwrite the default one through a poor own one. Default Python class behavior is to use `__repr__` as `__str__` anyways.

If you tried using `attrs` with `Exceptions` and were puzzled by the tracebacks: this option is for you.

- Don't overwrite `__name__` with `__qualname__` for `attr.s(slots=True)` classes. [#99](#)
-

16.2.0 (2016-09-17)

Changes:

- Add `attr.astuple()` that – similarly to `attr.asdict()` – returns the instance as a tuple. [#77](#)
 - Converts now work with frozen classes. [#76](#)
 - Instantiation of `attrs` classes with converters is now significantly faster. [#80](#)
 - Pickling now works with `__slots__` classes. [#81](#)
 - `attr.assoc()` now works with `__slots__` classes. [#84](#)
 - The tuple returned by `attr.fields()` now also allows to access the `Attribute` instances by name. Yes, we've subclassed `tuple` so you don't have to! Therefore `attr.fields(C).x` is equivalent to the deprecated `C.x` and works with `__slots__` classes. [#88](#)
-

16.1.0 (2016-08-30)

Backward-incompatible changes:

- All instances where function arguments were called `cl` have been changed to the more Pythonic `cls`. Since it was always the first argument, it's doubtful anyone ever called those function with in the keyword form. If so, sorry for any breakage but there's no practical deprecation path to solve this ugly wart.

Deprecations:

- Accessing `Attribute` instances on class objects is now deprecated and will stop working in 2017. If you need introspection please use the `__attrs_attrs__` attribute or the `attr.fields()` function that carry them too. In the future, the attributes that are defined on the class body and are usually overwritten in your `__init__` method are simply removed after `@attr.s` has been applied.

This will remove the confusing error message if you write your own `__init__` and forget to initialize some attribute. Instead you will get a straightforward `AttributeError`. In other words: decorated classes will work more like plain Python classes which was always `attrs`'s goal.

- The serious business aliases `attr.attributes` and `attr.attr` have been deprecated in favor of `attr.attrs` and `attr.attrib` which are much more consistent and frankly obvious in hindsight. They will be purged from documentation immediately but there are no plans to actually remove them.

Changes:

- `attr.asdict()`'s `dict_factory` arguments is now propagated on recursion. [#45](#)
- `attr.asdict()`, `attr.has()` and `attr.fields()` are significantly faster. [#48](#) [#51](#)
- Add `attr.attrs` and `attr.attrib` as a more consistent aliases for `attr.s` and `attr.ib`.
- Add `frozen` option to `attr.s` that will make instances best-effort immutable. [#60](#)
- `attr.asdict()` now takes `retain_collection_types` as an argument. If `True`, it does not convert attributes of type `tuple` or `set` to `list`. [#69](#)

16.0.0 (2016-05-23)

Backward-incompatible changes:

- Python 3.3 and 2.6 aren't supported anymore. They may work by chance but any effort to keep them working has ceased.

The last Python 2.6 release was on October 29, 2013 and isn't supported by the CPython core team anymore. Major Python packages like Django and Twisted dropped Python 2.6 a while ago already.

Python 3.3 never had a significant user base and wasn't part of any distribution's LTS release.

Changes:

- `__slots__` have arrived! Classes now can automatically be `slots`-style (and save your precious memory) just by passing `slots=True`. [#35](#)
- Allow the case of initializing attributes that are set to `init=False`. This allows for clean initializer parameter lists while being able to initialize attributes to default values. [#32](#)
- `attr.asdict()` can now produce arbitrary mappings instead of Python `dicts` when provided with a `dict_factory` argument. [#40](#)
- Multiple performance improvements.

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 4

Indices and tables

- `genindex`
- `search`

A

asdict() (in module attr), [24](#)
assoc() (in module attr), [26](#)
astuple() (in module attr), [25](#)
Attribute (class in attr), [22](#)
AttrsAttributeNotFoundError, [23](#)

E

exclude() (in module attr.filters), [26](#)

F

Factory (class in attr), [23](#)
fields() (in module attr), [23](#)
FrozenInstanceError, [23](#)

G

get_run_validators() (in module attr), [27](#)

H

has() (in module attr), [24](#)

I

ib() (in module attr), [21](#)
include() (in module attr.filters), [25](#)
instance_of() (in module attr.validators), [27](#)

M

make_class() (in module attr), [22](#)

N

NotAnAttrsClassError, [23](#)

O

optional() (in module attr.validators), [27](#)

P

provides() (in module attr.validators), [27](#)

S

s() (in module attr), [20](#)
set_run_validators() (in module attr), [27](#)

V

validate() (in module attr), [26](#)