attrs

Release 22.1.0.dev0

Hynek Schlawack

Jun 24, 2022

CONTENTS

Release v22.1.0.dev0 (*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). Trusted by NASA for Mars missions since 2020!

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

CHAPTER

ONE

GETTING STARTED

`attrs` is a Python-only package hosted on PyPI. The recommended installation method is pip-installing into a virtualenv:

```
$ python -m pip install attrs
```

The next three steps should bring you up and running in no time:

- *Overview* will show you a simple example of `attrs` in action and introduce you to its philosophy. Afterwards, you can start writing your own classes and understand what drives `attrs`'s design.
- *attrs by Example* will give you a comprehensive tour of `attrs`'s features. After reading, you will know about our advanced features and how to use them.
- If you're confused by all the `attr.s`, `attr.ib`, `attrs`, `attrib`, `define`, `frozen`, and `field`, head over to *On The Core API Names* for a very short explanation, and optionally a quick history lesson.
- Finally *Why not…* gives you a rundown of potential alternatives and why we think `attrs` is superior. Yes, we've heard about `namedtuple`s and Data Classes!
- If at any point you get confused by some terminology, please check out our *Glossary*.

If you need any help while getting started, feel free to use the `python-attrs` tag on Stack Overflow and someone will surely help you out!

CHAPTER

TWO

DAY-TO-DAY USAGE

- *Type Annotations* help you to write *correct* and *self-documenting* code. `attrs` has first class support for them, yet keeps them optional if you're not convinced!
- Instance initialization is one of `attrs` key feature areas. Our goal is to relieve you from writing as much code as possible. *Initialization* gives you an overview what `attrs` has to offer and explains some related philosophies we believe in.
- Comparing and ordering objects is a common task. *Comparison* shows you how `attrs` helps you with that and how you can customize it.
- If you want to put objects into sets or use them as keys in dictionaries, they have to be hashable. The simplest way to do that is to use frozen classes, but the topic is more complex than it seems and *Hashing* will give you a primer on what to look out for.
- Once you're comfortable with the concepts, our *API Reference* contains all information you need to use `attrs` to its fullest.
- `attrs` is built for extension from the ground up. *Extending* will show you the affordances it offers and how to make it a building block of your own projects.

CHAPTER

THREE

GETTING HELP

Please use the `python-attrs` tag on Stack Overflow to get help.

Answering questions of your fellow developers is also a great way to help the project!

CHAPTER

FOUR

FULL TABLE OF CONTENTS

4.1 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:

```
>>> from attrs import asdict, define, make_class, Factory

>>> @define
... class SomeClass:
...     a_number: int = 42
...     list_of_numbers: list[int] = Factory(list)
...
...     def hard_math(self, another_number):
...         return self.a_number + sum(self.list_of_numbers) * another_number


>>> sc = SomeClass(1, [1, 2, 3])
>>> sc
SomeClass(a_number=1, list_of_numbers=[1, 2, 3])

>>> sc.hard_math(3)
19
>>> sc == SomeClass(1, [1, 2, 3])
True
>>> sc != SomeClass(2, [3, 2, 1])
True

>>> asdict(sc)
{'a_number': 1, 'list_of_numbers': [1, 2, 3]}

>>> SomeClass()
SomeClass(a_number=42, list_of_numbers=[])

>>> C = make_class("C", ["a", "b"])
>>> C("foo", "bar")
C(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__`,
- equality-checking methods,
- an initializer,
- and much more,

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

Hate type annotations!? No problem! Types are entirely **optional** with `attrs`. Simply assign `attrs.field()` to the attributes instead of annotating them with types.

This example uses `attrs`'s modern APIs that have been introduced in version 20.1.0, and the `attrs` package import name that has been added in version 21.3.0. The classic APIs (`@attr.s`, `attr.ib`, plus their serious business aliases) and the `attr` package import name will remain **indefinitely**.

Please check out On The Core API Names for a more in-depth explanation.

4.1.1 Data Classes

On the tin, `attrs` might remind you of `dataclasses` (and indeed, `dataclasses` are a descendant of `attrs`). In practice it does a lot more and is more flexible. For instance it allows you to define special handling of NumPy arrays for equality checks, or allows more ways to plug into the initialization process.

For more details, please refer to our comparison page.

4.1.2 Philosophy

It's about regular classes.
: `attrs` is 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 `namedtuple`s 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 metaclasses. 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 except for 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.

4.1.3 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:

1. Take your declaration,
2. write *dunder methods* based on that information,
3. 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.

4.2 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. It predates type annotations and hence Data Classes, but it masterfully illustrates the appeal of class-building packages.

4.2.1 ...Data Classes?

PEP 557 added Data Classes to Python 3.7 that resemble attrs in many ways.

They are the result of the Python community's wish to have an easier way to write classes in the standard library that doesn't carry the problems of namedtuples. To that end, attrs and its developers were involved in the PEP process and while we may disagree with some minor decisions that have been made, it's a fine library and if it stops you from abusing namedtuples, they are a huge win.

Nevertheless, there are still reasons to prefer attrs over Data Classes. Whether they're relevant to *you* depends on your circumstances:

- Data Classes are *intentionally* less powerful than attrs. There is a long list of features that were sacrificed for the sake of simplicity and while the most obvious ones are validators, converters, *equality customization*, or *extensibility* in general, it permeates throughout all APIs.

  On the other hand, Data Classes currently do not offer any significant feature that attrs doesn't already have.
- attrs supports all mainstream Python versions including PyPy.
- attrs doesn't force type annotations on you if you don't like them.
- But since it **also** supports typing, it's the best way to embrace type hints *gradually*, too.
- While Data Classes are implementing features from attrs every now and then, their presence is dependent on the Python version, not the package version. For example, support for __slots__ has only been added in Python 3.10, but it doesn't do cell rewriting and therefore doesn't support bare calls to super(). This may or may not be fixed in later Python releases, but handling all these differences is especially painful for PyPI packages that support multiple Python versions. And of course, this includes possible implementation bugs.
- attrs can and will move faster. We are not bound to any release schedules and we have a clear deprecation policy.

  One of the reasons to not vendor attrs in the standard library was to not impede attrs's future development.

One way to think about attrs vs Data Classes is that attrs is a fully-fledged toolkit to write powerful classes while Data Classes are an easy way to get a class with some attributes. Basically what attrs was in 2015.

4.2.2 ...pydantic?

pydantic is first and foremost a *data validation library*. As such, it is a capable complement to class building libraries like `attrs` (or Data Classes!) for parsing and validating untrusted data.

However, as convenient as it might be, using it for your business or data layer is problematic in several ways: Is it really necessary to re-validate all your objects while reading them from a trusted database? In the parlance of Form, Command, and Model Validation, *pydantic* is the right tool for *Commands*.

Separation of concerns feels tedious at times, but it's one of those things that you get to appreciate once you've shot your own foot often enough.

4.2.3 ...namedtuples?

`collections.namedtuple`s are tuples with names, not classes.[1] Since writing classes is tiresome in Python, every now and then someone discovers all the typing they could save and gets really excited. However, that convenience comes at a price.

The most obvious difference between `namedtuple`s and `attrs`-based classes is that the latter are type-sensitive:

```
>>> import attr
>>> C1 = attr.make_class("C1", ["a"])
>>> C2 = attr.make_class("C2", ["a"])
>>> i1 = C1(1)
>>> i2 = C2(1)
>>> i1.a == i2.a
True
>>> i1 == i2
False
```

...while a `namedtuple` is *intentionally* behaving like a tuple which means the type of a tuple is *ignored*:

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

Other often surprising behaviors include:

- Since they are a subclass of tuples, `namedtuple`s have a length and are both iterable and indexable. That's not what you'd expect from a class and is likely to shadow subtle typo bugs.
- Iterability also implies that it's easy to accidentally unpack a `namedtuple` which leads to hard-to-find bugs.[3]
- `namedtuple`s have their methods *on your instances* whether you like it or not.[2]

[1] The word is that `namedtuple`s were added to the Python standard library as a way to make tuples in return values more readable. And indeed that is something you see throughout the standard library.

Looking at what the makers of `namedtuple`s use it for themselves is a good guideline for deciding on your own use cases.

[3] *`attr.astuple`* can be used to get that behavior in `attrs` on *explicit demand*.

[2] `attrs` only adds a single attribute: `__attrs_attrs__` for introspection. All helpers are functions in the `attr` package. Since they take the instance as first argument, you can easily attach them to your classes under a name of your own choice.

- `namedtuple`s are *always* immutable. Not only does that mean that you can't decide for yourself whether your instances should be immutable or not, it also means that if you want to influence your class' initialization (validation? default values?), you have to implement `__new__()` which is a particularly hacky and error-prone requirement for a very common problem.[4]
- To attach methods to a `namedtuple` you have to subclass it. And if you follow the standard library documentation's recommendation of:

```
class Point(namedtuple('Point', ['x', 'y'])):
    # ...
```

  you end up with a class that has *two* `Point`s in its `__mro__`: `[<class 'point.Point'>, <class 'point.Point'>, <type 'tuple'>, <type 'object'>]`.

  That's not only confusing, it also has very practical consequences: for example if you create documentation that includes class hierarchies like Sphinx's autodoc with `show-inheritance`. Again: common problem, hacky solution with confusing fallout.

All these things make `namedtuple`s a particularly poor choice for public APIs because all your objects are irrevocably tainted. With `attrs` your users won't notice a difference because it creates regular, well-behaved classes.

Summary

If you want a *tuple with names*, by all means: go for a `namedtuple`.[5] But if you want a class with methods, you're doing yourself a disservice by relying on a pile of hacks that requires you to employ even more hacks as your requirements expand.

Other than that, `attrs` also adds nifty features like validators, converters, and (mutable!) default values.

4.2.4 ...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")
```

[4] `attrs` offers *optional* immutability through the `frozen` keyword.

[5] Although `attrs` would serve you just as well! Since both employ the same method of writing and compiling Python code for you, the performance penalty is negligible at worst and in some cases `attrs` is even faster if you use `slots=True` (which is generally a good idea anyway).

?

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 run into mysterious tuples you had no idea what the hell they meant while debugging, you're much smarter than yours truly.

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.

4.2.5 ...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.

`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. So if you never iterate over the keys of a dict, you should use a proper class.

4.2.6 ...hand-written classes?

While we're fans of all things artisanal, writing the same nine methods again and again doesn't qualify. 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:
...     a = attr.ib()
...     b = attr.ib()
>>> SmartClass(1, 2)
SmartClass(a=1, b=2)
```

is roughly

```
>>> class ArtisanalClass:
...     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):
```

(continues on next page)

(continued from previous page)

```
...         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.__class__, 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__`?

It also should be noted that `attrs` is not an all-or-nothing solution. You can freely choose which features you want and disable those that you want more control over:

```
>>> @attr.s(repr=False)
... class SmartClass:
...     a = attr.ib()
...     b = attr.ib()
...
```

(continues on next page)

(continued from previous page)

```
...     def __repr__(self):
...         return "<SmartClass(a=%d)>" % (self.a,)
>>> SmartClass(1, 2)
<SmartClass(a=1)>
```

Summary

If you don't care and like typing, we're not gonna stop you.

However it takes a lot of bias and determined rationalization to claim that `attrs` raises the mental burden on a project given how difficult it is to find the important bits in a hand-written class and how annoying it is to ensure you've copy-pasted your code correctly over all your classes.

In any case, if you ever get sick of the repetitiveness and drowning important code in a sea of boilerplate, `attrs` will be waiting for you.

4.3 `attrs` by Example

4.3.1 Basics

The simplest possible usage is:

```
>>> from attrs import define, field
>>> @define
... class Empty:
...     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:

```
>>> @define
... class Coordinates:
...     x: int
...     y: int
```

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)
```

(continues on next page)

(continued from previous page)

```
>>> c1 == c2
False
```

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

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

```
>>> @define
... class C:
...     _x: int
>>> C(x=1)
C(_x=1)
```

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

```
>>> @define
... class C:
...     _x: int = field(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 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:
...     def __init__(self, x):
...         self.x = x
>>> SomethingFromSomeoneElse = define(
...     these={
...         "x": field()
...     }, init=False)(SomethingFromSomeoneElse)
>>> SomethingFromSomeoneElse(1)
SomethingFromSomeoneElse(x=1)
```

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

```
>>> @define(slots=False)
... class A:
...     a: int
...     def get_a(self):
...         return self.a
>>> @define(slots=False)
... class B:
...     b: int
>>> @define(slots=False)
... class C(B, A):
...     c: int
>>> i = C(1, 2, 3)
>>> i
```

(continues on next page)

(continued from previous page)

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

Slotted classes, which are the default for the new APIs, don't play well with multiple inheritance so we don't use them in the example.

The order of the attributes is defined by the MRO.

Keyword-only Attributes

You can also add keyword-only attributes:

```
>>> @define
... class A:
...     a: int = field(kw_only=True)
>>> A()
Traceback (most recent call last):
  ...
TypeError: A() missing 1 required keyword-only argument: 'a'
>>> A(a=1)
A(a=1)
```

`kw_only` may also be specified at via `define`, and will apply to all attributes:

```
>>> @define(kw_only=True)
... class A:
...     a: int
...     b: int
>>> A(1, 2)
Traceback (most recent call last):
  ...
TypeError: __init__() takes 1 positional argument but 3 were given
>>> A(a=1, b=2)
A(a=1, b=2)
```

If you create an attribute with `init=False`, the `kw_only` argument is ignored.

Keyword-only attributes allow subclasses to add attributes without default values, even if the base class defines attributes with default values:

```
>>> @define
... class A:
...     a: int = 0
>>> @define
... class B(A):
...     b: int = field(kw_only=True)
>>> B(b=1)
B(a=0, b=1)
>>> B()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  ...
TypeError: B() missing 1 required keyword-only argument: 'b'
```

If you don't set `kw_only=True`, then there is no valid attribute ordering, and you'll get an error:

```
>>> @define
... class A:
...     a: int = 0
>>> @define
... class B(A):
...     b: int
Traceback (most recent call last):
  ...
ValueError: No mandatory attributes allowed after an attribute with a default value or␣
↪factory.  Attribute in question: Attribute(name='b', default=NOTHING, validator=None,␣
↪repr=True, cmp=True, hash=None, init=True, converter=None, metadata=mappingproxy({}),␣
↪type=int, kw_only=False)
```

4.3.2 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):

```
>>> from attrs import asdict

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

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

```
>>> @define
... class User:
...     email: str
...     password: str

>>> @define
... class UserList:
...     users: list[User]

>>> 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:

```
>>> from attrs import asdict, filters, fields

>>> @define
```

(continues on next page)

(continued from previous page)

```
... class User:
...     login: str
...     password: str
...     id: int

>>> asdict(
...     User("jane", "s33kred", 42),
...     filter=filters.exclude(fields(User).password, int))
{'login': 'jane'}

>>> @define
... class C:
...     x: str
...     y: str
...     z: int

>>> asdict(C("foo", "2", 3),
...        filter=filters.include(int, fields(C).x))
{'x': 'foo', 'z': 3}
```

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

```
>>> import sqlite3
>>> from attrs import astuple

>>> @define
... class Foo:
...    a: int
...    b: int

>>> 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 (?, ?)", astuple(foo))
...    foo2 = Foo(*c.execute("SELECT x, y FROM foo").fetchone())
<sqlite3.Cursor object at ...>
<sqlite3.Cursor object at ...>
>>> foo == foo2
True
```

For more advanced transformations and conversions, we recommend you look at a companion library (such as cattrs).

4.3.3 Defaults

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

```
>>> import collections

>>> @define
... class Connection:
...     socket: int
...     @classmethod
...     def connect(cls, db_string):
...         # ... connect somehow to db_string ...
...         return cls(socket=42)

>>> @define
... class ConnectionPool:
...     db_string: str
...     pool: collections.deque = Factory(collections.deque)
...     debug: bool = 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.

Default factories can also be set using the `factory` argument to `field`, and using a decorator. The method receives the partially initialized instance which enables you to base a default value on other attributes:

```
>>> @define
... class C:
...     x: int = 1
...     y: int = field()
...     @y.default
...     def _any_name_except_a_name_of_an_attribute(self):
```

(continues on next page)

(continued from previous page)

```
...         return self.x + 1
...     z: list = field(factory=list)
>>> C()
C(x=1, y=2, z=[])
```

Please keep in mind that the decorator approach *only* works if the attribute in question has a `field` assigned to it. As a result, annotating an attribute with a type is *not* enough if you use `@default`.

4.3.4 Validators

Although your initializers should do as little as possible (ideally: just initialize your instance according to the arguments!), it can come in handy to do some kind of validation on the arguments.

`attrs` offers two ways to define validators for each attribute and it's up to you to choose which one suits your style and project better.

You can use a decorator:

```
>>> @define
... class C:
...     x: int = field()
...     @x.validator
...     def check(self, attribute, value):
...         if value > 42:
...             raise ValueError("x must be smaller or equal to 42")
>>> C(42)
C(x=42)
>>> C(43)
Traceback (most recent call last):
   ...
ValueError: x must be smaller or equal to 42
```

…or a callable…

```
>>> from attrs import validators

>>> def x_smaller_than_y(instance, attribute, value):
...     if value >= instance.y:
...         raise ValueError("'x' has to be smaller than 'y'!")
>>> @define
... class C:
...     x: int = field(validator=[validators.instance_of(int),
...                               x_smaller_than_y])
...     y: int
>>> 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'!
```

…or both at once:

```
>>> @define
... class C:
...     x: int = field(validator=validators.instance_of(int))
...     @x.validator
...     def fits_byte(self, attribute, value):
...         if not 0 <= value < 256:
...             raise ValueError("value out of bounds")
>>> C(128)
C(x=128)
>>> C("128")
Traceback (most recent call last):
  ...
TypeError: ("'x' must be <class 'int'> (got '128' that is a <class 'str'>).",␣
↪Attribute(name='x', default=NOTHING, validator=[<instance_of validator for type <class
↪'int'>>, <function fits_byte at 0x10fd7a0d0>], repr=True, cmp=True, hash=True,␣
↪init=True, metadata=mappingproxy({}), type=int, converter=None, kw_only=False), <class
↪'int'>, '128')
>>> C(256)
Traceback (most recent call last):
  ...
ValueError: value out of bounds
```

Please note that the decorator approach only works if – and only if! – the attribute in question has a `field` assigned. Therefore if you use `@validator`, it is *not* enough to annotate said attribute with a type.

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

```
>>> @define
... class C:
...     x: int = field(validator=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=None, kw_only=False), <type 'int'>, '42')
```

Please note that if you use *`attr.s`* (and not *`attrs.define`*) to define your class, validators only run on initialization by default. This behavior can be changed using the `on_setattr` argument.

Check out *Validators* for more details.

4.3.5 Conversion

Attributes can have a `converter` 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.

```
>>> @define
... class C:
...     x: int = field(converter=int)
```

(continues on next page)

(continued from previous page)

```
>>> o = C("1")
>>> o.x
1
```

Please note that converters only run on initialization.

Check out *Converters* for more details.

4.3.6 Metadata

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

```
>>> from attrs import fields

>>> @define
... class C:
...     x = field(metadata={'my_metadata': 1})
>>> fields(C).x.metadata
mappingproxy({'my_metadata': 1})
>>> 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*.

4.3.7 Types

`attrs` also allows you to associate a type with an attribute using either the *type* argument to *`attr.ib`* or – as of Python 3.6 – using **PEP 526**-annotations:

```
>>> from attrs import fields

>>> @define
... class C:
...     x: int
>>> fields(C).x.type
<class 'int'>

>>> import attr
>>> @attr.s
... class C:
...     x = attr.ib(type=int)
>>> fields(C).x.type
<class 'int'>
```

If you don't mind annotating *all* attributes, you can even drop the *`attrs.field`* and assign default values instead:

```
>>> import typing
>>> from attrs import fields
```

(continues on next page)

(continued from previous page)

```
>>> @define
... class AutoC:
...     cls_var: typing.ClassVar[int] = 5  # this one is ignored
...     l: list[int] = Factory(list)
...     x: int = 1
...     foo: str = "every attrib needs a type if auto_attribs=True"
...     bar: typing.Any = None
>>> fields(AutoC).l.type
list[int]
>>> fields(AutoC).x.type
<class 'int'>
>>> fields(AutoC).foo.type
<class 'str'>
>>> fields(AutoC).bar.type
typing.Any
>>> AutoC()
AutoC(l=[], x=1, foo='every attrib needs a type if auto_attribs=True', bar=None)
>>> AutoC.cls_var
5
```

The generated `__init__` method will have an attribute called `__annotations__` that contains this type information.

If your annotations contain strings (e.g. forward references), you can resolve these after all references have been defined by using *`attrs.resolve_types()`*. This will replace the *type* attribute in the respective fields.

```
>>> from attrs import fields, resolve_types

>>> @define
... class A:
...     a: 'list[A]'
...     b: 'B'
...
>>> @define
... class B:
...     a: A
...
>>> fields(A).a.type
'list[A]'
>>> fields(A).b.type
'B'
>>> resolve_types(A, globals(), locals())
<class 'A'>
>>> fields(A).a.type
list[A]
>>> fields(A).b.type
<class 'B'>
```

Note: If you find yourself using string type annotations to handle forward references, wrap the entire type annotation in quotes instead of only the type you need a forward reference to (so `'list[A]'` instead of `list['A']`). This is a limitation of the Python typing system.

Warning: `attrs` itself doesn't have any features that work on top of type metadata *yet*. However it's useful for writing your own validators or serialization frameworks.

4.3.8 Slots

Slotted classes have several advantages on CPython. Defining `__slots__` by hand is tedious, in `attrs` it's just a matter of using *`attrs.define`* or passing `slots=True` to *`attr.s`*:

```
>>> import attr

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

4.3.9 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:

```
>>> @frozen
... class C:
...     x: int
>>> 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.evolve`*:

```
>>> from attrs import evolve, frozen

>>> @frozen
... class C:
...     x: int
...     y: int
>>> i1 = C(1, 2)
>>> i1
C(x=1, y=2)
>>> i2 = evolve(i1, y=3)
>>> i2
C(x=1, y=3)
>>> i1 == i2
False
```

4.3.10 Other Goodies

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

```
>>> from attrs import fields, make_class
>>> @define
... class C1:
...     x = field()
...     y = field()
>>> C2 = make_class("C2", ["x", "y"])
>>> fields(C1) == fields(C2)
True
```

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

```
>>> from attrs import make_class

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

If you need to dynamically make a class with *attrs.make_class* and it needs to be a subclass of something else than `object`, use the `bases` argument:

```
>>> from attrs import make_class

>>> class D:
...     def __eq__(self, other):
...         return True  # arbitrary example
>>> C = make_class("C", {}, bases=(D,), cmp=False)
>>> isinstance(C(), D)
True
```

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 `@define`. 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.

```
>>> @define
... class C:
...     x: int
...     y: int
...     z: int = field(init=False)
...
...     def __attrs_post_init__(self):
...         self.z = self.x + self.y
```

(continues on next page)

(continued from previous page)

```
>>> obj = C(x=1, y=2)
>>> obj
C(x=1, y=2, z=3)
```

You can exclude single attributes from certain methods:

```
>>> @define
... class C:
...     user: str
...     password: str = field(repr=False)
>>> C("me", "s3kr3t")
C(user='me')
```

Alternatively, to influence how the generated `__repr__()` method formats a specific attribute, specify a custom callable to be used instead of the `repr()` built-in function:

```
>>> @define
... class C:
...     user: str
...     password: str = field(repr=lambda value: '***')
>>> C("me", "s3kr3t")
C(user='me', password=***)
```

4.4 Type Annotations

`attrs` comes with first class support for type annotations for both Python 3.6 (**PEP 526**) and legacy syntax.

However they will forever remain *optional*, therefore the example from the README could also be written as:

```
>>> from attrs import define, field

>>> @define
... class SomeClass:
...     a_number = field(default=42)
...     list_of_numbers = field(factory=list)

>>> sc = SomeClass(1, [1, 2, 3])
>>> sc
SomeClass(a_number=1, list_of_numbers=[1, 2, 3])
```

You can choose freely between the approaches, but please remember that if you choose to use type annotations, you **must** annotate **all** attributes!

Even when going all-in on type annotations, you will need *attr.field* for some advanced features though.

One of those features are the decorator-based features like defaults. It's important to remember that `attrs` doesn't do any magic behind your back. All the decorators are implemented using an object that is returned by the call to *attrs.field*.

Attributes that only carry a class annotation do not have that object so trying to call a method on it will inevitably fail.

Please note that types – however added – are *only metadata* that can be queried from the class and they aren't used for anything out of the box!

Because Python does not allow references to a class object before the class is defined, types may be defined as string literals, so-called *forward references* (**PEP 526**). You can enable this automatically for a whole module by using `from __future__ import annotations` (**PEP 563**) as of Python 3.7. In this case `attrs` simply puts these string literals into the `type` attributes. If you need to resolve these to real types, you can call *`attrs.resolve_types`* which will update the attribute in place.

In practice though, types show their biggest usefulness in combination with tools like mypy, pytype, or pyright that have dedicated support for `attrs` classes.

The addition of static types is certainly one of the most exciting features in the Python ecosystem and helps you write *correct* and *verified self-documenting* code.

If you don't know where to start, Carl Meyer gave a great talk on Type-checked Python in the Real World at PyCon US 2018 that will help you to get started in no time.

4.4.1 mypy

While having a nice syntax for type metadata is great, it's even greater that mypy as of 0.570 ships with a dedicated `attrs` plugin which allows you to statically check your code.

Imagine you add another line that tries to instantiate the defined class using `SomeClass("23")`. Mypy will catch that error for you:

```
$ mypy t.py
t.py:12: error: Argument 1 to "SomeClass" has incompatible type "str"; expected "int"
```

This happens *without* running your code!

And it also works with *both* Python 2-style annotation styles. To mypy, this code is equivalent to the one above:

```
@attr.s
class SomeClass:
    a_number = attr.ib(default=42)  # type: int
    list_of_numbers = attr.ib(factory=list, type=list[int])
```

4.4.2 pyright

`attrs` provides support for pyright though the dataclass_transform specification. This provides static type inference for a subset of `attrs` equivalent to standard-library `dataclasses`, and requires explicit type annotations using the *`attrs.define`* or `@attr.s(auto_attribs=True)` API.

Given the following definition, `pyright` will generate static type signatures for `SomeClass` attribute access, `__init__`, `__eq__`, and comparison methods:

```
@attr.define
class SomeClass:
    a_number: int = 42
    list_of_numbers: list[int] = attr.field(factory=list)
```

Warning: The `pyright` inferred types are a subset of those supported by `mypy`, including:

- The generated `__init__` signature only includes the attribute type annotations. It currently does not include attribute `converter` types.
- The `attr.frozen` decorator is not typed with frozen attributes, which are properly typed via `attr.define(frozen=True)`.

A full list of limitations and incompatibilities can be found in pyright's repository.

Your constructive feedback is welcome in both attrs#795 and pyright#1782. Generally speaking, the decision on improving `attrs` support in pyright is entirely Microsoft's prerogative though.

4.5 Initialization

In Python, instance initialization happens in the `__init__` method. Generally speaking, you should keep as little logic as possible in it, and you should think about what the class needs and not how it is going to be instantiated.

Passing complex objects into `__init__` and then using them to derive data for the class unnecessarily couples your new class with the old class which makes it harder to test and also will cause problems later.

So assuming you use an ORM and want to extract 2D points from a row object, do not write code like this:

```
class Point:
    def __init__(self, database_row):
        self.x = database_row.x
        self.y = database_row.y

pt = Point(row)
```

Instead, write a `classmethod` that will extract it for you:

```
@define
class Point:
    x: float
    y: float

    @classmethod
    def from_row(cls, row):
        return cls(row.x, row.y)

pt = Point.from_row(row)
```

Now you can instantiate `Point`s without creating fake row objects in your tests and you can have as many smart creation helpers as you want, in case more data sources appear.

For similar reasons, we strongly discourage from patterns like:

```
pt = Point(**row.attributes)
```

which couples your classes to the database data model. Try to design your classes in a way that is clean and convenient to use – not based on your database format. The database format can change anytime and you're stuck with a bad class design that is hard to change. Embrace functions and classmethods as a filter between reality and what's best for you to work with.

If you look for object serialization, there's a bunch of projects listed on our `attrs` extensions Wiki page. Some of them even support nested schemas.

4.5.1 Private Attributes

One thing people tend to find confusing is the treatment of private attributes that start with an underscore. `attrs` follows the doctrine that there is no such thing as a private argument and strips the underscores from the name when writing the `__init__` method signature:

```
>>> import inspect, attr, attrs
>>> from attr import define
>>> @define
... class C:
...     _x: int
>>> inspect.signature(C.__init__)
<Signature (self, x: int) -> None>
```

There really isn't a right or wrong, it's a matter of taste. But it's important to be aware of it because it can lead to surprising syntax errors:

```
>>> @define
... class C:
...     _1: int
Traceback (most recent call last):
   ...
SyntaxError: invalid syntax
```

In this case a valid attribute name `_1` got transformed into an invalid argument name `1`.

4.5.2 Defaults

Sometimes you don't want to pass all attribute values to a class. And sometimes, certain attributes aren't even intended to be passed but you want to allow for customization anyways for easier testing.

This is when default values come into play:

```
>>> from attr import define, field, Factory

>>> @define
... class C:
...     a: int = 42
...     b: list = field(factory=list)
...     c: list = Factory(list)  # syntactic sugar for above
...     d: dict = field()
...     @d.default
...     def _any_name_except_a_name_of_an_attribute(self):
...         return {}
>>> C()
C(a=42, b=[], c=[], d={})
```

It's important that the decorated method – or any other method or property! – doesn't have the same name as the attribute, otherwise it would overwrite the attribute definition.

Please note that as with function and method signatures, `default=[]` will *not* do what you may think it might do:

```
>>> @define
... class C:
```

(continues on next page)

(continued from previous page)

```
...     x = []
>>> i = C()
>>> k = C()
>>> i.x.append(42)
>>> k.x
[42]
```

This is why `attrs` comes with factory options.

Warning: Please note that the decorator based defaults have one gotcha: they are executed when the attribute is set, that means depending on the order of attributes, the `self` object may not be fully initialized when they're called.

Therefore you should use `self` as little as possible.

Even the smartest of us can get confused by what happens if you pass partially initialized objects around.

4.5.3 Validators

Another thing that definitely *does* belong in `__init__` is checking the resulting instance for invariants. This is why `attrs` has the concept of validators.

Decorator

The most straightforward way is using the attribute's `validator` method as a decorator.

The method has to accept three arguments:

1. the *instance* that's being validated (aka `self`),
2. the *attribute* that it's validating, and finally
3. the *value* that is passed for it.

These values are passed as *positional arguments*, therefore their names don't matter.

If the value does not pass the validator's standards, it just raises an appropriate exception.

```
>>> @define
... class C:
...     x: int = field()
...     @x.validator
...     def _check_x(self, attribute, value):
...         if value > 42:
...             raise ValueError("x must be smaller or equal to 42")
>>> C(42)
C(x=42)
>>> C(43)
Traceback (most recent call last):
  ...
ValueError: x must be smaller or equal to 42
```

Again, it's important that the decorated method doesn't have the same name as the attribute and that the *attrs.field()* helper is used.

Callables

If you want to re-use your validators, you should have a look at the `validator` argument to *attrs.field*.

It takes either a callable or a list of callables (usually functions) and treats them as validators that receive the same arguments as with the decorator approach. Also as with the decorator approach, they are passed as *positional arguments* so you can name them however you want.

Since the validators run *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'!")
>>> @define
... class C:
...     x = field(validator=[attrs.validators.instance_of(int),
...                          x_smaller_than_y])
...     y = field()
>>> 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'!
```

This example also shows of some syntactic sugar for using the *attrs.validators.and_* validator: if you pass a list, all validators have to pass.

`attrs` won't intercept your changes to those attributes but you can always call *attrs.validate* on any instance to verify that it's still valid: When using *attrs.define* or *attrs.frozen*, `attrs` will run the validators even when setting the attribute.

```
>>> i = C(4, 5)
>>> i.x = 5
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:

```
>>> @define
... class C:
...     x = field(validator=attrs.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=None), <type 'int'>, '42')
```

Of course you can mix and match the two approaches at your convenience. If you define validators both ways for an attribute, they are both ran:

```
>>> @define
... class C:
...     x = field(validator=attrs.validators.instance_of(int))
...     @x.validator
...     def fits_byte(self, attribute, value):
...         if not 0 <= value < 256:
...             raise ValueError("value out of bounds")
>>> C(128)
C(x=128)
>>> C("128")
Traceback (most recent call last):
  ...
TypeError: ("'x' must be <class 'int'> (got '128' that is a <class 'str'>).",␣
↪Attribute(name='x', default=NOTHING, validator=[<instance_of validator for type <class
↪'int'>>, <function fits_byte at 0x10fd7a0d0>], repr=True, cmp=True, hash=True,␣
↪init=True, metadata=mappingproxy({}), type=None, converter=one), <class 'int'>, '128')
>>> C(256)
Traceback (most recent call last):
  ...
ValueError: value out of bounds
```

And finally you can disable validators globally:

```
>>> attrs.validators.set_disabled(True)
>>> C("128")
C(x='128')
>>> attrs.validators.set_disabled(False)
>>> C("128")
Traceback (most recent call last):
  ...
TypeError: ("'x' must be <class 'int'> (got '128' that is a <class 'str'>).",␣
↪Attribute(name='x', default=NOTHING, validator=[<instance_of validator for type <class
↪'int'>>, <function fits_byte at 0x10fd7a0d0>], repr=True, cmp=True, hash=True,␣
↪init=True, metadata=mappingproxy({}), type=None, converter=None), <class 'int'>, '128')
```

You can achieve the same by using the context manager:

```
>>> with attrs.validators.disabled():
...     C("128")
C(x='128')
>>> C("128")
Traceback (most recent call last):
  ...
TypeError: ("'x' must be <class 'int'> (got '128' that is a <class 'str'>).",␣
↪Attribute(name='x', default=NOTHING, validator=[<instance_of validator for type <class
↪'int'>>, <function fits_byte at 0x10fd7a0d0>], repr=True, cmp=True, hash=True,␣
↪init=True, metadata=mappingproxy({}), type=None, converter=None), <class 'int'>, '128')
```

4.5.4 Converters

Finally, sometimes you may want to normalize the values coming in. For that `attrs` comes with converters.

Attributes can have a `converter` 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.

```
>>> @define
... class C:
...     x = field(converter=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.")
>>> @define
... class C:
...     x = field(converter=int, validator=validate_x)
>>> o = C("0")
>>> o.x
0
>>> C("-1")
Traceback (most recent call last):
    ...
ValueError: x must be at least 0.
```

Arguably, you can abuse converters as one-argument validators:

```
>>> C("x")
Traceback (most recent call last):
    ...
ValueError: invalid literal for int() with base 10: 'x'
```

If a converter's first argument has a type annotation, that type will appear in the signature for `__init__`. A converter will override an explicit type annotation or `type` argument.

```
>>> def str2int(x: str) -> int:
...     return int(x)
>>> @define
... class C:
...     x = field(converter=str2int)
>>> C.__init__.__annotations__
{'return': None, 'x': <class 'str'>}
```

4.5.5 Hooking Yourself Into Initialization

Generally speaking, the moment you think that you need finer control over how your class is instantiated than what `attrs` offers, it's usually best to use a classmethod factory or to apply the builder pattern.

However, sometimes you need to do that one quick thing before or after your class is initialized. And for that `attrs` offers three means:

- `__attrs_pre_init__` is automatically detected and run *before* `attrs` starts initializing. This is useful if you need to inject a call to `super().__init__()`.
- `__attrs_post_init__` is automatically detected and run *after* `attrs` is done initializing your instance. This is useful if you want to derive some attribute from others or perform some kind of validation over the whole instance.
- `__attrs_init__` is written and attached to your class *instead* of `__init__`, if `attrs` is told to not write one (i.e. `init=False` or a combination of `auto_detect=True` and a custom `__init__`). This is useful if you want full control over the initialization process, but don't want to set the attributes by hand.

Pre Init

The sole reason for the existence of `__attrs_pre_init__` is to give users the chance to call `super().__init__()`, because some subclassing-based APIs require that.

```
>>> @define
... class C:
...     x: int
...     def __attrs_pre_init__(self):
...         super().__init__()
>>> C(42)
C(x=42)
```

If you need more control, use the custom init approach described next.

Custom Init

If you tell `attrs` to not write an `__init__`, it will write an `__attrs_init__` instead, with the same code that it would have used for `__init__`. You have full control over the initialization, but also have to type out the types of your arguments etc. Here's an example of a manual default value:

```
>>> from typing import Optional

>>> @define
... class C:
...     x: int
...
...     def __init__(self, x: int = 42):
...         self.__attrs_init__(x)
>>> C()
C(x=42)
```

Post Init

```
>>> @define
... class C:
...     x: int
...     y: int = field(init=False)
...     def __attrs_post_init__(self):
...         self.y = self.x + 1
>>> C(1)
C(x=1, y=2)
```

Please note that you can't directly set attributes on frozen classes:

```
>>> @frozen
... class FrozenBroken:
...     x: int
...     y: int = field(init=False)
...     def __attrs_post_init__(self):
...         self.y = self.x + 1
>>> FrozenBroken(1)
Traceback (most recent call last):
   ...
attrs.exceptions.FrozenInstanceError: can't set attribute
```

If you need to set attributes on a frozen class, you'll have to resort to the *same trick* as `attrs` and use `object.__setattr__()`:

```
>>> @define
... class Frozen:
...     x: int
...     y: int = field(init=False)
...     def __attrs_post_init__(self):
...         object.__setattr__(self, "y", self.x + 1)
>>> Frozen(1)
Frozen(x=1, y=2)
```

Note that you *must not* access the hash code of the object in `__attrs_post_init__` if `cache_hash=True`.

4.5.6 Order of Execution

If present, the hooks are executed in the following order:

1. `__attrs_pre_init__` (if present on *current* class)
2. For each attribute, in the order it was declared:
   a. default factory
   b. converter
3. *all* validators
4. `__attrs_post_init__` (if present on *current* class)

Notably this means, that you can access all attributes from within your validators, but your converters have to deal with invalid values and have to return a valid value.

4.5.7 Derived Attributes

One of the most common `attrs` questions on *Stack Overflow* is how to have attributes that depend on other attributes. For example if you have an API token and want to instantiate a web client that uses it for authentication. Based on the previous sections, there are two approaches.

The simpler one is using `__attrs_post_init__`:

```
@define
class APIClient:
    token: str
    client: WebClient = field(init=False)

    def __attrs_post_init__(self):
        self.client = WebClient(self.token)
```

The second one is using a decorator-based default:

```
@define
class APIClient:
    token: str
    client: WebClient = field()  # needed! attr.ib works too

    @client.default
    def _client_factory(self):
        return WebClient(self.token)
```

That said, and as pointed out in the beginning of the chapter, a better approach would be to have a factory class method:

```
@define
class APIClient:
    client: WebClient

    @classmethod
    def from_token(cls, token: str) -> "APIClient":
        return cls(client=WebClient(token))
```

This makes the class more testable.

4.6 Comparison

By default, two instances of `attrs` classes are equal if all their fields are equal. For that, `attrs` writes `__eq__` and `__ne__` methods for you.

Additionally, if you pass `order=True` (which is the default if you use the *attr.s* decorator), `attrs` will also create a full set of ordering methods that are based on the defined fields: `__le__`, `__lt__`, `__ge__`, and `__gt__`.

4.6.1 Customization

As with other features, you can exclude fields from being involved in comparison operations:

```
>>> from attr import define, field

>>> @define
... class C:
...     x: int
...     y: int = field(eq=False)

>>> C(1, 2) == C(1, 3)
True
```

Additionally you can also pass a *callable* instead of a bool to both *eq* and *order*. It is then used as a key function like you may know from `sorted`:

```
>>> from attr import define, field

>>> @define
... class S:
...     x: str = field(eq=str.lower)

>>> S("foo") == S("FOO")
True

>>> @define(order=True)
... class C:
...     x: str = field(order=int)

>>> C("10") > C("2")
True
```

This is especially useful when you have fields with objects that have atypical comparison properties. Common examples of such objects are NumPy arrays.

To save you unnecessary boilerplate, `attrs` comes with the *`attrs.cmp_using`* helper to create such functions. For NumPy arrays it would look like this:

```
import numpy

@define(order=False)
class C:
   an_array = field(eq=attr.cmp_using(eq=numpy.array_equal))
```

Warning: Please note that *eq* and *order* are set *independently*, because *order* is `False` by default in *`attrs.define`* (but not in *`attr.s`*). You can set both at once by using the *cmp* argument that we've undeprecated just for this use-case.

4.7 Hashing

4.7.1 Hash Method Generation

Warning: The overarching theme is to never set the `@attr.s(hash=X)` parameter yourself. Leave it at `None` which means that `attrs` will do the right thing for you, depending on the other parameters:

- If you want to make objects hashable by value: use `@attr.s(frozen=True)`.
- If you want hashing and equality by object identity: use `@attr.s(eq=False)`

Setting `hash` yourself can have unexpected consequences so we recommend to tinker with it only if you know exactly what you're doing.

Under certain circumstances, it's necessary for objects to be *hashable*. For example if you want to put them into a `set` or if you want to use them as keys in a `dict`.

The *hash* of an object is an integer that represents the contents of an object. It can be obtained by calling `hash` on an object and is implemented by writing a `__hash__` method for your class.

`attrs` will happily write a `__hash__` method for you[1], however it will *not* do so by default. Because according to the definition from the official Python docs, the returned hash has to fulfill certain constraints:

1. Two objects that are equal, **must** have the same hash. This means that if `x == y`, it *must* follow that `hash(x) == hash(y)`.

   By default, Python classes are compared *and* hashed by their `id`. That means that every instance of a class has a different hash, no matter what attributes it carries.

   It follows that the moment you (or `attrs`) change the way equality is handled by implementing `__eq__` which is based on attribute values, this constraint is broken. For that reason Python 3 will make a class that has customized equality unhashable. Python 2 on the other hand will happily let you shoot your foot off. Unfortunately, `attrs` still mimics (otherwise unsupported) Python 2's behavior for backward compatibility reasons if you set `hash=False`.

   The *correct way* to achieve hashing by id is to set `@attr.s(eq=False)`. Setting `@attr.s(hash=False)` (which implies `eq=True`) is almost certainly a *bug*.

   Warning: Be careful when subclassing! Setting `eq=False` on a class whose base class has a non-default `__hash__` method will *not* make `attrs` remove that `__hash__` for you.

   It is part of `attrs`'s philosophy to only *add* to classes so you have the freedom to customize your classes as you wish. So if you want to *get rid* of methods, you'll have to do it by hand.

   The easiest way to reset `__hash__` on a class is adding `__hash__ = object.__hash__` in the class body.

2. If two objects are not equal, their hash **should** be different.

   While this isn't a requirement from a standpoint of correctness, sets and dicts become less effective if there are a lot of identical hashes. The worst case is when all objects have the same hash which turns a set into a list.

3. The hash of an object **must not** change.

   If you create a class with `@attr.s(frozen=True)` this is fulfilled by definition, therefore `attrs` will write a `__hash__` function for you automatically. You can also force it to write one with `hash=True` but then it's *your*

[1] The hash is computed by hashing a tuple that consists of a unique id for the class plus all attribute values.

responsibility to make sure that the object is not mutated.

This point is the reason why mutable structures like lists, dictionaries, or sets aren't hashable while immutable ones like tuples or frozensets are: point 1 and 2 require that the hash changes with the contents but point 3 forbids it.

For a more thorough explanation of this topic, please refer to this blog post: Python Hashes and Equality.

4.7.2 Hashing and Mutability

Changing any field involved in hash code computation after the first call to `__hash__` (typically this would be after its insertion into a hash-based collection) can result in silent bugs. Therefore, it is strongly recommended that hashable classes be `frozen`. Beware, however, that this is not a complete guarantee of safety: if a field points to an object and that object is mutated, the hash code may change, but `frozen` will not protect you.

4.7.3 Hash Code Caching

Some objects have hash codes which are expensive to compute. If such objects are to be stored in hash-based collections, it can be useful to compute the hash codes only once and then store the result on the object to make future hash code requests fast. To enable caching of hash codes, pass `cache_hash=True` to `@attrs`. This may only be done if `attrs` is already generating a hash function for the object.

4.8 API Reference

`attrs` works by decorating a class using *`attrs.define`* or *`attr.s`* and then optionally defining attributes on the class using *`attrs.field`*, *`attr.ib`*, or a type annotation.

If you're confused by the many names, please check out *On The Core API Names* for clarification.

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

As of version 21.3.0, `attrs` consists of **two** top-level package names:

- The classic `attr` that powered the venerable *`attr.s`* and *`attr.ib`*
- The modern `attrs` that only contains most modern APIs and relies on *`attrs.define`* and *`attrs.field`* to define your classes. Additionally it offers some `attr` APIs with nicer defaults (e.g. *`attrs.asdict`*). Using this namespace requires Python 3.6 or later.

The `attrs` namespace is built *on top of* `attr` which will *never* go away.

4.8.1 Core

Note: Please note that the `attrs` namespace has been added in version 21.3.0. Most of the objects are simply re-imported from `attr`. Therefore if a class, method, or function claims that it has been added in an older version, it is only available in the `attr` namespace.

`attrs.NOTHING = NOTHING`

Sentinel class to indicate the lack of a value when `None` is ambiguous.

`_Nothing` is a singleton. There is only ever one of it.

Changed in version 21.1.0: `bool(NOTHING)` is now False.

attrs.**define**(*maybe_cls=None*, ***, *these=None*, *repr=None*, *hash=None*, *init=None*, *slots=True*, *frozen=False*, *weakref_slot=True*, *str=False*, *auto_attribs=None*, *kw_only=False*, *cache_hash=False*, *auto_exc=True*, *eq=None*, *order=False*, *auto_detect=True*, *getstate_setstate=None*, *on_setattr=None*, *field_transformer=None*, *match_args=True*)

Define an `attrs` class.

Differences to the classic *attr.s* that it uses underneath:

- Automatically detect whether or not *auto_attribs* should be `True` (c.f. *auto_attribs* parameter).
- If *frozen* is `False`, run converters and validators when setting an attribute by default.
- *slots=True* (see *slotted classes* for potentially surprising behaviors)
- *auto_exc=True*
- *auto_detect=True*
- *order=False*
- Some options that were only relevant on Python 2 or were kept around for backwards-compatibility have been removed.

Please note that these are all defaults and you can change them as you wish.

Parameters

auto_attribs (*Optional[bool]*) – If set to `True` or `False`, it behaves exactly like *attr.s*. If left `None`, *attr.s* will try to guess:

1. If any attributes are annotated and no unannotated *attrs.fields*s are found, it assumes *auto_attribs=True*.
2. Otherwise it assumes *auto_attribs=False* and tries to collect *attrs.fields*s.

For now, please refer to *attr.s* for the rest of the parameters.

New in version 20.1.0.

Changed in version 21.3.0: Converters are also run `on_setattr`.

attrs.**mutable**(*same_as_define*)

Alias for *attrs.define*.

New in version 20.1.0.

attrs.**frozen**(*same_as_define*)

Behaves the same as *attrs.define* but sets *frozen=True* and *on_setattr=None*.

New in version 20.1.0.

attrs.**field**(***, *default=NOTHING*, *validator=None*, *repr=True*, *hash=None*, *init=True*, *metadata=None*, *converter=None*, *factory=None*, *kw_only=False*, *eq=None*, *order=None*, *on_setattr=None*)

Identical to *attr.ib*, except keyword-only and with some arguments removed.

New in version 20.1.0.

attr.**define**()

Old import path for *attrs.define*.

attr.**mutable**()

Old import path for *attrs.mutable*.

attr.**frozen**()

Old import path for *attrs.frozen*.

attr.**field**()

Old import path for *attrs.field*.

class attrs.**Attribute**(*name*, *default*, *validator*, *repr*, *cmp*, *hash*, *init*, *inherited*, *metadata=None*, *type=None*, *converter=None*, *kw_only=False*, *eq=None*, *eq_key=None*, *order=None*, *order_key=None*, *on_setattr=None*)

Read-only representation of an attribute.

The class has *all* arguments of *attr.ib* (except for `factory` which is only syntactic sugar for `default=Factory(...)` plus the following:

- `name` (`str`): The name of the attribute.
- `inherited` (`bool`): Whether or not that attribute has been inherited from a base class.
- `eq_key` and `order_key` (`typing.Callable` or `None`): The callables that are used for comparing and ordering objects by this attribute, respectively. These are set by passing a callable to *attr.ib*'s `eq`, `order`, or `cmp` arguments. See also *comparison customization*.

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.
- The *field transformer* hook receives a list of them.

New in version 20.1.0: *inherited*

New in version 20.1.0: *on_setattr*

Changed in version 20.2.0: *inherited* is not taken into account for equality checks and hashing anymore.

New in version 21.1.0: *eq_key* and *order_key*

For the full version history of the fields, see *attr.ib*.

For example:

```
>>> import attr
>>> @attr.s
... class C:
...     x = attr.ib()
>>> attr.fields(C).x
Attribute(name='x', default=NOTHING, validator=None, repr=True, eq=True, eq_
↪key=None, order=True, order_key=None, hash=None, init=True, metadata=mappingproxy(
↪{}), type=None, converter=None, kw_only=False, inherited=False, on_setattr=None)
```

evolve(***changes*)

Copy *self* and apply *changes*.

This works similarly to *attr.evolve* but that function does not work with `Attribute`.

It is mainly meant to be used for *Automatic Field Transformation and Modification*.

New in version 20.3.0.

attrs.**make_class**(*name*, *attrs*, *bases=(<class 'object'>,)*, ***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.

  If *attrs* is a list or an ordered dict (dict on Python 3.6+, collections.OrderedDict otherwise), the order is deduced from the order of the names or attributes inside *attrs*. Otherwise the order of the definition of the attributes is used.

- **bases** (*tuple*) – Classes that the new class will subclass.
- **attributes_arguments** – Passed unmodified to *attr.s*.

Returns
A new class with *attrs*.

Return type
type

New in version 17.1.0: *bases*

Changed in version 18.1.0: If *attrs* is ordered, the order is retained.

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 attrs.**Factory**(*factory*, *takes_self=False*)

Stores a factory callable.

If passed as the default value to *attrs.field*, the factory is used to generate a new value.

Parameters

- **factory** (*callable*) – A callable that takes either none or exactly one mandatory positional argument depending on *takes_self*.
- **takes_self** (*bool*) – Pass the partially initialized instance that is being initialized as a positional argument.

New in version 17.1.0: *takes_self*

For example:

```
>>> @attr.s
... class C:
...     x = attr.ib(default=attr.Factory(list))
...     y = attr.ib(default=attr.Factory(
...         lambda self: set(self.x),
...         takes_self=True)
...     )
>>> C()
C(x=[], y=set())
>>> C([1, 2, 3])
C(x=[1, 2, 3], y={1, 2, 3})
```

Classic

attr.**NOTHING**

Same as *attrs.NOTHING*.

attr.**s**(*these=None*, *repr_ns=None*, *repr=None*, *cmp=None*, *hash=None*, *init=None*, *slots=False*, *frozen=False*, *weakref_slot=True*, *str=False*, *auto_attribs=False*, *kw_only=False*, *cache_hash=False*, *auto_exc=False*, *eq=None*, *order=None*, *auto_detect=False*, *collect_by_mro=False*, *getstate_setstate=None*, *on_setattr=None*, *field_transformer=None*, *match_args=True*)

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.

  If *these* is not `None`, `attrs` will *not* search the class body for attributes and will *not* remove any attributes from it.

  If *these* is an ordered dict (`dict` on Python 3.6+, `collections.OrderedDict` otherwise), the order is deduced from the order of the attributes inside *these*. Otherwise the order of the definition of the attributes is used.

- **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.

- **auto_detect** (*bool*) – Instead of setting the *init*, *repr*, *eq*, *order*, and *hash* arguments explicitly, assume they are set to `True` **unless any** of the involved methods for one of the arguments is implemented in the *current* class (i.e. it is *not* inherited from some base class).

  So for example by implementing `__eq__` on a class yourself, `attrs` will deduce `eq=False` and will create *neither* `__eq__` *nor* `__ne__` (but Python classes come with a sensible `__ne__` by default, so it *should* be enough to only implement `__eq__` in most cases).

  > **Warning:** If you prevent `attrs` from creating the ordering methods for you (`order=False`, e.g. by implementing `__le__`), it becomes *your* responsibility to make sure its ordering is sound. The best way is to use the `functools.total_ordering` decorator.

  Passing `True` or `False` to *init*, *repr*, *eq*, *order*, *cmp*, or *hash* overrides whatever *auto_detect* would determine.

  auto_detect requires Python 3. Setting it `True` on Python 2 raises an *attrs.exceptions.PythonTooOldError*.

- **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 `Exception`s.

- **eq** (*Optional[bool]*) – If `True` or `None` (default), add `__eq__` and `__ne__` methods that check two instances for equality.

  They compare the instances as if they were tuples of their `attrs` attributes if and only if the types of both classes are *identical*!

- **order** (*Optional[bool]*) – If `True`, add `__lt__`, `__le__`, `__gt__`, and `__ge__` methods that behave like *eq* above and allow instances to be ordered. If `None` (default) mirror value of *eq*.
- **cmp** (*Optional[bool]*) – Setting *cmp* is equivalent to setting *eq* and *order* to the same value. Must not be mixed with *eq* or *order*.
- **hash** (*Optional[bool]*) – If `None` (default), the `__hash__` method is generated according how *eq* and *frozen* are set.

  1. If *both* are True, `attrs` will generate a `__hash__` for you.
  2. If *eq* is True and *frozen* is False, `__hash__` will be set to None, marking it unhashable (which it is).
  3. If *eq* is False, `__hash__` will be left untouched meaning the `__hash__` method of the base class will be used (if base class is `object`, this means it will fall back to id-based hashing.).

  Although not recommended, you can decide for yourself and force `attrs` to create one (e.g. if the class is immutable even though you didn't freeze it programmatically) by passing `True` or not. Both of these cases are rather special and should be used carefully.

  See our documentation on *Hashing*, Python's documentation on `object.__hash__`, and the GitHub issue that led to the default behavior for more details.
- **init** (*bool*) – Create a `__init__` method that initializes the `attrs` attributes. Leading underscores are stripped for the argument name. If a `__attrs_pre_init__` method exists on the class, it will be called before the class is initialized. If a `__attrs_post_init__` method exists on the class, it will be called after the class is fully initialized.

  If `init` is `False`, an `__attrs_init__` method will be injected instead. This allows you to define a custom `__init__` method that can do pre-init work such as `super().__init__()`, and then call `__attrs_init__()` and `__attrs_post_init__()`.
- **slots** (*bool*) – Create a *slotted class* that's more memory-efficient. Slotted classes are generally superior to the default dict classes, but have some gotchas you should know about, so we encourage you to read the *glossary entry*.
- **frozen** (*bool*) – Make instances immutable after initialization. If someone attempts to modify a frozen instance, *attr.exceptions.FrozenInstanceError* is raised.

  Note:

  1. This is achieved by installing a custom `__setattr__` method on your class, so you can't implement your own.
  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`.
  4. If a class is frozen, you cannot modify `self` in `__attrs_post_init__` or a self-written `__init__`. You can circumvent that limitation by using `object.__setattr__(self, "attribute_name", value)`.
  5. Subclasses of a frozen class are frozen too.

- **weakref_slot** (*bool*) – Make instances weak-referenceable. This has no effect unless `slots` is also enabled.

- **auto_attribs** (*bool*) – If `True`, collect **PEP 526**-annotated attributes (Python 3.6 and later only) from the class body.

  In this case, you **must** annotate every field. If `attrs` encounters a field that is set to an *attr.ib* but lacks a type annotation, an *attr.exceptions.UnannotatedAttributeError* is raised. Use `field_name: typing.Any = attr.ib(...)` if you don't want to set a type.

  If you assign a value to those attributes (e.g. `x: int = 42`), that value becomes the default value like if it were passed using `attr.ib(default=42)`. Passing an instance of *attrs.Factory* also works as expected in most cases (see warning below).

  Attributes annotated as `typing.ClassVar`, and attributes that are neither annotated nor set to an *attr.ib* are **ignored**.

  > **Warning:** For features that use the attribute name to create decorators (e.g. *validators*), you still *must* assign *attr.ib* to them. Otherwise Python will either not find the name or try to use the default value to call e.g. `validator` on it.
  >
  > These errors can be quite confusing and probably the most common bug report on our bug tracker.

- **kw_only** (*bool*) – Make all attributes keyword-only (Python 3+) in the generated `__init__` (if `init` is `False`, this parameter is ignored).
- **cache_hash** (*bool*) – Ensure that the object's hash code is computed only once and stored on the object. If this is set to `True`, hashing must be either explicitly or implicitly enabled for this class. If the hash code is cached, avoid any reassignments of fields involved in hash code computation or mutations of the objects those fields point to after object creation. If such changes occur, the behavior of the object's hash code is undefined.
- **auto_exc** (*bool*) – If the class subclasses `BaseException` (which implicitly includes any subclass of any exception), the following happens to behave like a well-behaved Python exceptions class:
  - the values for *eq*, *order*, and *hash* are ignored and the instances compare and hash by the instance's ids (N.B. `attrs` will *not* remove existing implementations of `__hash__` or the equality methods. It just won't add own ones.),
  - all attributes that are either passed into `__init__` or have a default value are additionally available as a tuple in the `args` attribute,
  - the value of *str* is ignored leaving `__str__` to base classes.
- **collect_by_mro** (*bool*) – Setting this to `True` fixes the way `attrs` collects attributes from base classes. The default behavior is incorrect in certain cases of multiple inheritance. It should be on by default but is kept off for backward-compatibility.

  See issue #428 for more details.
- **getstate_setstate** (*Optional[bool]*) –

  > **Note:** This is usually only interesting for slotted classes and you should probably just set *auto_detect* to `True`.

  If `True`, `__getstate__` and `__setstate__` are generated and attached to the class. This is necessary for slotted classes to be pickleable. If left `None`, it's `True` by default for slotted classes and `False` for dict classes.

If *auto_detect* is `True`, and *getstate_setstate* is left `None`, and **either** `__getstate__` or `__setstate__` is detected directly on the class (i.e. not inherited), it is set to `False` (this is usually what you want).

- **on_setattr** (`callable`, or a list of callables, or `None`, or *attrs.setters.NO_OP*) – A callable that is run whenever the user attempts to set an attribute (either by assignment like `i.x = 42` or by using `setattr` like `setattr(i, "x", 42)`). It receives the same arguments as validators: the instance, the attribute that is being modified, and the new value.

  If no exception is raised, the attribute is set to the return value of the callable.

  If a list of callables is passed, they're automatically wrapped in an *attrs.setters.pipe*.

- **field_transformer** (*Optional[callable]*) – A function that is called with the original class object and all fields right before `attrs` finalizes the class. You can use this, e.g., to automatically add converters or validators to fields based on their types. See *Automatic Field Transformation and Modification* for more details.

- **match_args** (*bool*) – If `True` (default), set `__match_args__` on the class to support **PEP 634** (Structural Pattern Matching). It is a tuple of all non-keyword-only `__init__` parameter names on Python 3.10 and later. Ignored on older Python versions.

New in version 16.0.0: *slots*

New in version 16.1.0: *frozen*

New in version 16.3.0: *str*

New in version 16.3.0: Support for `__attrs_post_init__`.

Changed in version 17.1.0: *hash* supports `None` as value which is also the default now.

New in version 17.3.0: *auto_attribs*

Changed in version 18.1.0: If *these* is passed, no attributes are deleted from the class body.

Changed in version 18.1.0: If *these* is ordered, the order is retained.

New in version 18.2.0: *weakref_slot*

Deprecated since version 18.2.0: `__lt__`, `__le__`, `__gt__`, and `__ge__` now raise a `DeprecationWarning` if the classes compared are subclasses of each other. `__eq` and `__ne__` never tried to compared subclasses to each other.

Changed in version 19.2.0: `__lt__`, `__le__`, `__gt__`, and `__ge__` now do not consider subclasses comparable anymore.

New in version 18.2.0: *kw_only*

New in version 18.2.0: *cache_hash*

New in version 19.1.0: *auto_exc*

Deprecated since version 19.2.0: *cmp* Removal on or after 2021-06-01.

New in version 19.2.0: *eq* and *order*

New in version 20.1.0: *auto_detect*

New in version 20.1.0: *collect_by_mro*

New in version 20.1.0: *getstate_setstate*

New in version 20.1.0: *on_setattr*

New in version 20.3.0: *field_transformer*

Changed in version 21.1.0: `init=False` injects `__attrs_init__`

Changed in version 21.1.0: Support for `__attrs_pre_init__`

Changed in version 21.1.0: *cmp* undeprecated

New in version 21.3.0: *match_args*

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

For example:

```
>>> import attr
>>> @attr.s
... class C:
...     _private = attr.ib()
>>> C(private=42)
C(_private=42)
>>> class D:
...     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.s(auto_exc=True)
... class Error(Exception):
...     x = attr.ib()
...     y = attr.ib(default=42, init=False)
>>> Error("foo")
Error(x='foo', y=42)
>>> raise Error("foo")
Traceback (most recent call last):
   ...
Error: ('foo', 42)
>>> raise ValueError("foo", 42)   # for comparison
Traceback (most recent call last):
   ...
ValueError: ('foo', 42)
```

`attr.ib`(*default=NOTHING*, *validator=None*, *repr=True*, *cmp=None*, *hash=None*, *init=True*, *metadata=None*, *type=None*, *converter=None*, *factory=None*, *kw_only=False*, *eq=None*, *order=None*, *on_setattr=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*) – 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 *attrs.Factory*, its callable will be used to construct a new value (useful for mutable data types like lists or dicts).

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

The default can also be set using decorator notation as shown below.

- **factory** (*callable*) – Syntactic sugar for default=attr.Factory(factory).
- **validator** (callable or a list of callables.) – 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.

  If a list is passed, its items are treated as validators and must all pass.

  Validators can be globally disabled and re-enabled using *get_run_validators*.

  The validator can also be set using decorator notation as shown below.
- **repr** (a bool or a callable to use a custom function.) – Include this attribute in the generated __repr__ method. If True, include the attribute; if False, omit it. By default, the built-in repr() function is used. To override how the attribute value is formatted, pass a callable that takes a single value and returns a string. Note that the resulting string is used as-is, i.e. it will be used directly *instead* of calling repr() (the default).
- **eq** (a bool or a callable.) – If True (default), include this attribute in the generated __eq__ and __ne__ methods that check two instances for equality. To override how the attribute value is compared, pass a callable that takes a single value and returns the value to be compared.
- **order** (a bool or a callable.) – If True (default), include this attributes in the generated __lt__, __le__, __gt__ and __ge__ methods. To override how the attribute value is ordered, pass a callable that takes a single value and returns the value to be ordered.
- **cmp** (a bool or a callable.) – Setting *cmp* is equivalent to setting *eq* and *order* to the same value. Must not be mixed with *eq* or *order*.
- **hash** (*Optional[bool]*) – Include this attribute in the generated __hash__ method. If None (default), mirror *eq*'s value. This is the correct behavior according the Python spec. Setting this value to anything else than None is *discouraged*.
- **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.
- **converter** (*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. See *Metadata*.
- **type** – The type of the attribute. In Python 3.6 or greater, the preferred method to specify the type is using a variable annotation (see **PEP 526**). This argument is provided for backward compatibility. Regardless of the approach used, the type will be stored on Attribute.type.

  Please note that attrs doesn't do anything with this metadata by itself. You can use it as part of your own code or for *static type checking*.
- **kw_only** – Make this attribute keyword-only (Python 3+) in the generated __init__ (if init is False, this parameter is ignored).

- **on_setattr** (callable, or a list of callables, or None, or *attrs.setters.NO_OP*) – Allows to overwrite the *on_setattr* setting from *attr.s*. If left None, the *on_setattr* value from *attr.s* is used. Set to *attrs.setters.NO_OP* to run **no** setattr hooks for this attribute – regardless of the setting in *attr.s*.

New in version 15.2.0: *convert*

New in version 16.3.0: *metadata*

Changed in version 17.1.0: *validator* can be a `list` now.

Changed in version 17.1.0: *hash* is `None` and therefore mirrors *eq* by default.

New in version 17.3.0: *type*

Deprecated since version 17.4.0: *convert*

New in version 17.4.0: *converter* as a replacement for the deprecated *convert* to achieve consistency with other noun-based arguments.

New in version 18.1.0: `factory=f` is syntactic sugar for `default=attr.Factory(f)`.

New in version 18.2.0: *kw_only*

Changed in version 19.2.0: *convert* keyword argument removed.

Changed in version 19.2.0: *repr* also accepts a custom callable.

Deprecated since version 19.2.0: *cmp* Removal on or after 2021-06-01.

New in version 19.2.0: *eq* and *order*

New in version 20.1.0: *on_setattr*

Changed in version 20.3.0: *kw_only* backported to Python 2

Changed in version 21.1.0: *eq*, *order*, and *cmp* also accept a custom callable

Changed in version 21.1.0: *cmp* undeprecated

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

The object returned by *attr.ib* also allows for setting the default and the validator using decorators:

```
>>> @attr.s
... class C:
...     x = attr.ib()
...     y = attr.ib()
...     @x.validator
...     def _any_name_except_a_name_of_an_attribute(self, attribute, value):
...         if value < 0:
...             raise ValueError("x must be positive")
...     @y.default
...     def _any_name_except_a_name_of_an_attribute(self):
...         return self.x + 1
>>> C(1)
C(x=1, y=2)
>>> C(-1)
Traceback (most recent call last):
    ...
ValueError: x must be positive
```

4.8.2 Exceptions

All exceptions are available from both `attr.exceptions` and `attrs.exceptions` and are the same thing. That means that it doesn't matter from from which namespace they've been raised and/or caught:

```
>>> import attrs, attr
>>> try:
...     raise attrs.exceptions.FrozenError()
... except attr.exceptions.FrozenError:
...     print("this works!")
this works!
```

exception attrs.exceptions.**PythonTooOldError**

It was attempted to use an `attrs` feature that requires a newer Python version.

New in version 18.2.0.

exception attrs.exceptions.**FrozenError**

A frozen/immutable instance or attribute have been attempted to be modified.

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

New in version 20.1.0.

exception attrs.exceptions.**FrozenInstanceError**

A frozen instance has been attempted to be modified.

New in version 16.1.0.

exception attrs.exceptions.**FrozenAttributeError**

A frozen attribute has been attempted to be modified.

New in version 20.1.0.

exception attrs.exceptions.**AttrsAttributeNotFoundError**

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

New in version 16.2.0.

exception attrs.exceptions.**NotAnAttrsClassError**

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

New in version 16.2.0.

exception attrs.exceptions.**DefaultAlreadySetError**

A default has been set using `attr.ib()` and is attempted to be reset using the decorator.

New in version 17.1.0.

exception attrs.exceptions.**UnannotatedAttributeError**

A class with `auto_attribs=True` has an `attr.ib()` without a type annotation.

New in version 17.3.0.

exception attrs.exceptions.**NotCallableError**(*msg*, *value*)

A `attr.ib()` requiring a callable has been set with a value that is not callable.

New in version 19.2.0.

For example:

```
@attr.s(auto_attribs=True)
class C:
    x: int
    y = attr.ib()  # <- ERROR!
```

4.8.3 Helpers

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

`attrs.cmp_using`(*eq=None*, *lt=None*, *le=None*, *gt=None*, *ge=None*, *require_same_type=True*, *class_name='Comparable'*)

Create a class that can be passed into *attr.ib*'s `eq`, `order`, and `cmp` arguments to customize field comparison.

The resulting class will have a full set of ordering methods if at least one of `{lt, le, gt, ge}` and `eq` are provided.

Parameters

- **eq** (*Optional[callable]*) – `callable` used to evaluate equality of two objects.
- **lt** (*Optional[callable]*) – `callable` used to evaluate whether one object is less than another object.
- **le** (*Optional[callable]*) – `callable` used to evaluate whether one object is less than or equal to another object.
- **gt** (*Optional[callable]*) – `callable` used to evaluate whether one object is greater than another object.
- **ge** (*Optional[callable]*) – `callable` used to evaluate whether one object is greater than or equal to another object.
- **require_same_type** (*bool*) – When `True`, equality and ordering methods will return `NotImplemented` if objects are not of the same type.
- **class_name** (*Optional[str]*) – Name of class. Defaults to 'Comparable'.

See *Comparison* for more details.

New in version 21.1.0.

`attr.cmp_using`()

Same as *attrs.cmp_using*.

`attrs.fields`(*cls*)

Return 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 *attrs.Attribute*

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

For example:

```
>>> @attr.s
... class C:
...     x = attr.ib()
...     y = attr.ib()
>>> attrs.fields(C)
(Attribute(name='x', default=NOTHING, validator=None, repr=True, eq=True, eq_
↪key=None, order=True, order_key=None, hash=None, init=True, metadata=mappingproxy(
↪{}), type=None, converter=None, kw_only=False, inherited=False, on_setattr=None),␣
↪Attribute(name='y', default=NOTHING, validator=None, repr=True, eq=True, eq_
↪key=None, order=True, order_key=None, hash=None, init=True, metadata=mappingproxy(
↪{}), type=None, converter=None, kw_only=False, inherited=False, on_setattr=None))
>>> attrs.fields(C)[1]
Attribute(name='y', default=NOTHING, validator=None, repr=True, eq=True, eq_
↪key=None, order=True, order_key=None, hash=None, init=True, metadata=mappingproxy(
↪{}), type=None, converter=None, kw_only=False, inherited=False, on_setattr=None)
>>> attrs.fields(C).y is attrs.fields(C)[1]
True
```

`attr.fields()`

Same as *attrs.fields*.

`attrs.fields_dict(cls)`

Return an ordered dictionary of `attrs` attributes for a class, whose keys are the attribute names.

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

an ordered dict where keys are attribute names and values are *attrs.Attribute*s. This will be a `dict` if it's naturally ordered like on Python 3.6+ or an `OrderedDict` otherwise.

New in version 18.1.0.

For example:

```
>>> @attr.s
... class C:
...     x = attr.ib()
...     y = attr.ib()
>>> attrs.fields_dict(C)
{'x': Attribute(name='x', default=NOTHING, validator=None, repr=True, eq=True, eq_
↪key=None, order=True, order_key=None, hash=None, init=True, metadata=mappingproxy(
↪{}), type=None, converter=None, kw_only=False, inherited=False, on_setattr=None),
↪'y': Attribute(name='y', default=NOTHING, validator=None, repr=True, eq=True, eq_
↪key=None, order=True, order_key=None, hash=None, init=True, metadata=mappingproxy(
↪{}), type=None, converter=None, kw_only=False, inherited=False, on_setattr=None)}
>>> attr.fields_dict(C)['y']
Attribute(name='y', default=NOTHING, validator=None, repr=True, eq=True, eq_
↪key=None, order=True, order_key=None, hash=None, init=True, metadata=mappingproxy(
↪{}), type=None, converter=None, kw_only=False, inherited=False, on_setattr=None)
```

(continued from previous page)

```
>>> attrs.fields_dict(C)['y'] is attrs.fields(C).y
True
```

attr.**fields_dict**()

Same as *attrs.fields_dict*.

attrs.**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:
...     pass
>>> attr.has(C)
True
>>> attr.has(object)
False
```

attr.**has**()

Same as *attrs.has*.

attrs.**resolve_types**(*cls*, *globalns=None*, *localns=None*, *attribs=None*)

Resolve any strings and forward annotations in type annotations.

This is only required if you need concrete types in *Attribute*'s *type* field. In other words, you don't need to resolve your types if you only use them for static type checking.

With no arguments, names will be looked up in the module in which the class was created. If this is not what you want, e.g. if the name only exists inside a method, you may pass *globalns* or *localns* to specify other dictionaries in which to look up these names. See the docs of `typing.get_type_hints` for more details.

Parameters

- **cls** (*type*) – Class to resolve.
- **globalns** (*Optional[dict]*) – Dictionary containing global variables.
- **localns** (*Optional[dict]*) – Dictionary containing local variables.
- **attribs** (*Optional[list]*) – List of attribs for the given class. This is necessary when calling from inside a `field_transformer` since *cls* is not an `attrs` class yet.

Raises

- **TypeError** – If *cls* is not a class.
- ***attr.exceptions.NotAnAttrsClassError*** – If *cls* is not an `attrs` class and you didn't pass any attribs.
- **NameError** – If types cannot be resolved because of missing variables.

Returns

cls so you can use this function also as a class decorator. Please note that you have to apply it **after** *attrs.define*. That means the decorator has to come in the line **before** *attrs.define*.

New in version 20.1.0.

New in version 21.1.0: *attribs*

For example:

```
>>> import typing
>>> @attrs.define
... class A:
...     a: typing.List['A']
...     b: 'B'
...
>>> @attrs.define
... class B:
...     a: A
...
>>> attrs.fields(A).a.type
typing.List[ForwardRef('A')]
>>> attrs.fields(A).b.type
'B'
>>> attrs.resolve_types(A, globals(), locals())
<class 'A'>
>>> attrs.fields(A).a.type
typing.List[A]
>>> attrs.fields(A).b.type
<class 'B'>
```

attr.**resolve_types**()

Same as *attrs.resolve_types*.

attrs.**asdict**(*inst*, *, *recurse=True*, *filter=None*, *value_serializer=None*)

Same as *attr.asdict*, except that collections types are always retained and dict is always used as *dict_factory*.

New in version 21.3.0.

For example:

```
>>> @attrs.define
... class C:
...     x: int
...     y: int
>>> attrs.asdict(C(1, C(2, 3)))
{'x': 1, 'y': {'x': 2, 'y': 3}}
```

attr.**asdict**(*inst*, *recurse=True*, *filter=None*, *dict_factory=<class 'dict'>*, *retain_collection_types=False*, *value_serializer=None*)

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 *attrs.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`.
- **value_serializer** (*Optional[callable]*) – A hook that is called for every attribute or dict key/value. It receives the current instance, field and value and must return the (updated) value. The hook is run *after* the optional *filter* has been applied.

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*

New in version 20.3.0: *value_serializer*

New in version 21.3.0: If a dict has a collection for a key, it is serialized as a tuple.

attrs.**astuple**(*inst*, *, *recurse=True*, *filter=None*)

Same as *attr.astuple*, except that collections types are always retained and `tuple` is always used as the *tuple_factory*.

New in version 21.3.0.

For example:

```
>>> @attrs.define
... class C:
...     x = attr.field()
...     y = attr.field()
>>> attrs.astuple(C(1,2))
(1, 2)
```

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 *attrs.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.

`attrs` includes some handy helpers for filtering the attributes in *attrs.asdict* and *attrs.astuple*:

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

Include *what*.

Parameters
what (`list` of `type` or *attrs.Attribute*s) – What to include.

Return type
`callable`

`attrs.filters.`**exclude**(**what*)

Exclude *what*.

Parameters
what (`list` of classes or *attrs.Attribute*s.) – What to exclude.

Return type
`callable`

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

Same as *attrs.filters.include*.

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

Same as *attrs.filters.exclude*.

See *attrs.asdict()* for examples.

All objects from `attrs.filters` are also available from `attr.filters`.

`attrs.`**evolve**(*inst*, ***changes*)

Create a new instance, based on *inst* with *changes* applied.

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

- `TypeError` – If *attr_name* couldn't be found in the class `__init__`.
- *attr.exceptions.NotAnAttrsClassError* – If *cls* is not an `attrs` class.

New in version 17.1.0.

For example:

```
>>> @attrs.define
... class C:
...     x: int
...     y: int
>>> i1 = C(1, 2)
>>> i1
C(x=1, y=2)
>>> i2 = attrs.evolve(i1, y=3)
>>> i2
C(x=1, y=3)
>>> i1 == i2
False
```

`evolve` creates a new instance using `__init__`. This fact has several implications:

- private attributes should be specified without the leading underscore, just like in `__init__`.
- attributes with `init=False` can't be set with `evolve`.
- the usual `__init__` validators will validate the new values.

`attr.evolve()`

Same as *attrs.evolve*.

`attrs.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:

```
>>> @attrs.define(on_setattr=attrs.setters.NO_OP)
... class C:
...     x = attrs.field(validator=attrs.validators.instance_of(int))
>>> i = C(1)
>>> i.x = "1"
>>> attrs.validate(i)
Traceback (most recent call last):
   ...
TypeError: ("'x' must be <class 'int'> (got '1' that is a <class 'str'>).", ...)
```

`attr.validate()`

Same as *attrs.validate*.

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.

Deprecated since version 21.3.0: It will not be removed, but it also will not be moved to new `attrs` namespace. Use *attrs.validators.set_disabled()* instead.

`attr.get_run_validators()`

Return whether or not validators are run.

Deprecated since version 21.3.0: It will not be removed, but it also will not be moved to new `attrs` namespace. Use *attrs.validators.get_disabled()* instead.

4.8.4 Validators

`attrs` comes with some common validators in the `attrs.validators` module. All objects from `attrs.validators` are also available from `attr.validators`.

`attrs.validators.lt`(*val*)

A validator that raises `ValueError` if the initializer is called with a number larger or equal to *val*.

Parameters
val – Exclusive upper bound for values

New in version 21.3.0.

For example:

```
>>> @attrs.define
... class C:
...     x = attrs.field(validator=attrs.validators.lt(42))
>>> C(41)
C(x=41)
>>> C(42)
Traceback (most recent call last):
   ...
ValueError: ("'x' must be < 42: 42")
```

`attrs.validators.le`(*val*)

A validator that raises `ValueError` if the initializer is called with a number greater than *val*.

Parameters
val – Inclusive upper bound for values

New in version 21.3.0.

For example:

```
>>> @attrs.define
... class C:
...     x = attrs.field(validator=attr.validators.le(42))
>>> C(42)
C(x=42)
>>> C(43)
Traceback (most recent call last):
   ...
ValueError: ("'x' must be <= 42: 43")
```

`attrs.validators.ge`(*val*)

A validator that raises `ValueError` if the initializer is called with a number smaller than *val*.

Parameters
val – Inclusive lower bound for values

New in version 21.3.0.

For example:

```
>>> @attrs.define
... class C:
...     x = attrs.field(validator=attrs.validators.ge(42))
>>> C(42)
C(x=42)
>>> C(41)
Traceback (most recent call last):
   ...
ValueError: ("'x' must be => 42: 41")
```

attrs.validators.**gt**(*val*)

A validator that raises `ValueError` if the initializer is called with a number smaller or equal to *val*.

Parameters
val – Exclusive lower bound for values

New in version 21.3.0.

For example:

```
>>> @attrs.define
... class C:
...     x = attr.field(validator=attrs.validators.gt(42))
>>> C(43)
C(x=43)
>>> C(42)
Traceback (most recent call last):
   ...
ValueError: ("'x' must be > 42: 42")
```

attrs.validators.**max_len**(*length*)

A validator that raises `ValueError` if the initializer is called with a string or iterable that is longer than *length*.

Parameters
length (*int*) – Maximum length of the string or iterable

New in version 21.3.0.

For example:

```
>>> @attrs.define
... class C:
...     x = attrs.field(validator=attrs.validators.max_len(4))
>>> C("spam")
C(x='spam')
>>> C("bacon")
Traceback (most recent call last):
   ...
ValueError: ("Length of 'x' must be <= 4: 5")
```

attrs.validators.**min_len**(*length*)

A validator that raises `ValueError` if the initializer is called with a string or iterable that is shorter than *length*.

Parameters
length (*int*) – Minimum length of the string or iterable

New in version 22.1.0.

For example:

```
>>> @attrs.define
... class C:
...     x = attrs.field(validator=attrs.validators.min_len(1))
>>> C("bacon")
C(x='bacon')
>>> C("")
Traceback (most recent call last):
   ...
ValueError: ("Length of 'x' must be => 1: 0")
```

attrs.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.

Raises

TypeError – With a human readable error message, the attribute (of type *attrs.Attribute*), the expected type, and the value it got.

For example:

```
>>> @attrs.define
... class C:
...     x = attrs.field(validator=attrs.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=None, kw_only=False), <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=None, init=True, type=None, kw_
↪only=False), <type 'int'>, None)
```

attrs.validators.**in_**(*options*)

A validator that raises a `ValueError` if the initializer is called with a value that does not belong in the options provided. The check is performed using `value in options`.

Parameters

options (list, tuple, `enum.Enum`, ...) – Allowed options.

Raises

ValueError – With a human readable error message, the attribute (of type *attrs.Attribute*), the expected options, and the value it got.

New in version 17.1.0.

Changed in version 22.1.0: The ValueError was incomplete until now and only contained the human readable error message. Now it contains all the information that has been promised since 17.1.0.

For example:

```
>>> import enum
>>> class State(enum.Enum):
...     ON = "on"
...     OFF = "off"
>>> @attrs.define
... class C:
...     state = attrs.field(validator=attrs.validators.in_(State))
...     val = attrs.field(validator=attrs.validators.in_([1, 2, 3]))
>>> C(State.ON, 1)
C(state=<State.ON: 'on'>, val=1)
>>> C("on", 1)
Traceback (most recent call last):
   ...
ValueError: 'state' must be in <enum 'State'> (got 'on'), Attribute(name='state',␣
↪default=NOTHING, validator=<in_ validator with options <enum 'State'>>, repr=True,
↪ eq=True, eq_key=None, order=True, order_key=None, hash=None, init=True,␣
↪metadata=mappingproxy({}), type=None, converter=None, kw_only=False,␣
↪inherited=False, on_setattr=None), <enum 'State'>, 'on')
>>> C(State.ON, 4)
Traceback (most recent call last):
...
ValueError: 'val' must be in [1, 2, 3] (got 4), Attribute(name='val',␣
↪default=NOTHING, validator=<in_ validator with options [1, 2, 3]>, repr=True,␣
↪eq=True, eq_key=None, order=True, order_key=None, hash=None, init=True,␣
↪metadata=mappingproxy({}), type=None, converter=None, kw_only=False,␣
↪inherited=False, on_setattr=None), [1, 2, 3], 4)
```

attrs.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.

Raises
`TypeError` – With a human readable error message, the attribute (of type *attrs.Attribute*), the expected interface, and the value it got.

attrs.validators.**and_**(**validators*)

A validator that composes multiple validators into one.

When called on a value, it runs all wrapped validators.

Parameters
validators (*callables*) – Arbitrary number of validators.

New in version 17.1.0.

For convenience, it's also possible to pass a list to *attrs.field*'s validator argument.

Thus the following two statements are equivalent:

```
x = attrs.field(validator=attrs.validators.and_(v1, v2, v3))
x = attrs.field(validator=[v1, v2, v3])
```

attrs.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 (callable or list of callables.) – A validator (or a list of validators) that is used for non-None values.

New in version 15.1.0.

Changed in version 17.1.0: *validator* can be a list of validators.

For example:

```
>>> @attrs.define
... class C:
...     x = attrs.field(validator=attrs.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=None, kw_only=False), <type 'int'>, '42')
>>> C(None)
C(x=None)
```

attrs.validators.**is_callable**()

A validator that raises a *attr.exceptions.NotCallableError* if the initializer is called with a value for this particular attribute that is not callable.

New in version 19.1.0.

Raises

attr.exceptions.NotCallableError – With a human readable error message containing the attribute (*attrs.Attribute*) name, and the value it got.

For example:

```
>>> @attrs.define
... class C:
...     x = attrs.field(validator=attrs.validators.is_callable())
>>> C(isinstance)
C(x=<built-in function isinstance>)
>>> C("not a callable")
Traceback (most recent call last):
    ...
attr.exceptions.NotCallableError: 'x' must be callable (got 'not a callable' that␣
↪is a <class 'str'>).
```

attrs.validators.**matches_re**(*regex*, *flags=0*, *func=None*)

A validator that raises ValueError if the initializer is called with a string that doesn't match *regex*.

Parameters

- **regex** – a regex string or precompiled pattern to match against
- **flags** (*int*) – flags that will be passed to the underlying re function (default 0)
- **func** (*callable*) – which underlying `re` function to call. Valid options are `re.fullmatch`, `re.search`, and `re.match`; the default `None` means `re.fullmatch`. For performance reasons, the pattern is always precompiled using `re.compile`.

New in version 19.2.0.

Changed in version 21.3.0: *regex* can be a pre-compiled pattern.

For example:

```
>>> @attrs.define
... class User:
...     email = attrs.field(validator=attrs.validators.matches_re(
...         "(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$)"))
>>> User(email="user@example.com")
User(email='user@example.com')
>>> User(email="user@example.com@test.com")
Traceback (most recent call last):
    ...
ValueError: ("'email' must match regex '(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\\\.[a-zA-
↪Z0-9-.]+$)' ('user@example.com@test.com' doesn't)", Attribute(name='email',␣
↪default=NOTHING, validator=<matches_re validator for pattern re.compile('(^[a-zA-
↪Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$)')>, repr=True, cmp=True, hash=None,␣
↪init=True, metadata=mappingproxy({}), type=None, converter=None, kw_only=False),␣
↪re.compile('(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$)'), 'user@example.
↪com@test.com')
```

attrs.validators.**deep_iterable**(*member_validator*, *iterable_validator=None*)

A validator that performs deep validation of an iterable.

Parameters

- **member_validator** – Validator(s) to apply to iterable members
- **iterable_validator** – Validator to apply to iterable itself (optional)

New in version 19.1.0.

Raises

TypeError – if any sub-validators fail

For example:

```
>>> @attrs.define
... class C:
...     x = attrs.field(validator=attrs.validators.deep_iterable(
...     member_validator=attrs.validators.instance_of(int),
...     iterable_validator=attrs.validators.instance_of(list)
...     ))
>>> C(x=[1, 2, 3])
C(x=[1, 2, 3])
>>> C(x=set([1, 2, 3]))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
TypeError: ("'x' must be <class 'list'> (got {1, 2, 3} that is a <class 'set'>).",␣
↪Attribute(name='x', default=NOTHING, validator=<deep_iterable validator for
↪<instance_of validator for type <class 'list'>> iterables of <instance_of␣
↪validator for type <class 'int'>>>, repr=True, cmp=True, hash=None, init=True,␣
↪metadata=mappingproxy({}), type=None, converter=None, kw_only=False), <class 'list
↪'>, {1, 2, 3})
>>> C(x=[1, 2, "3"])
Traceback (most recent call last):
    ...
TypeError: ("'x' must be <class 'int'> (got '3' that is a <class 'str'>).",␣
↪Attribute(name='x', default=NOTHING, validator=<deep_iterable validator for
↪<instance_of validator for type <class 'list'>> iterables of <instance_of␣
↪validator for type <class 'int'>>>, repr=True, cmp=True, hash=None, init=True,␣
↪metadata=mappingproxy({}), type=None, converter=None, kw_only=False), <class 'int
↪'>, '3')
```

attrs.validators.**deep_mapping**(*key_validator*, *value_validator*, *mapping_validator=None*)

A validator that performs deep validation of a dictionary.

Parameters

- **key_validator** – Validator to apply to dictionary keys
- **value_validator** – Validator to apply to dictionary values
- **mapping_validator** – Validator to apply to top-level mapping attribute (optional)

New in version 19.1.0.

Raises

TypeError – if any sub-validators fail

For example:

```
>>> @attrs.define
... class C:
...     x = attrs.field(validator=attrs.validators.deep_mapping(
...         key_validator=attrs.validators.instance_of(str),
...         value_validator=attrs.validators.instance_of(int),
...         mapping_validator=attrs.validators.instance_of(dict)
...     ))
>>> C(x={"a": 1, "b": 2})
C(x={'a': 1, 'b': 2})
>>> C(x=None)
Traceback (most recent call last):
    ...
TypeError: ("'x' must be <class 'dict'> (got None that is a <class 'NoneType'>).",␣
↪Attribute(name='x', default=NOTHING, validator=<deep_mapping validator for␣
↪objects mapping <instance_of validator for type <class 'str'>> to <instance_of␣
↪validator for type <class 'int'>>>, repr=True, cmp=True, hash=None, init=True,␣
↪metadata=mappingproxy({}), type=None, converter=None, kw_only=False), <class 'dict
↪'>, None)
>>> C(x={"a": 1.0, "b": 2})
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
TypeError: ("'x' must be <class 'int'> (got 1.0 that is a <class 'float'>).",␣
↪Attribute(name='x', default=NOTHING, validator=<deep_mapping validator for␣
↪objects mapping <instance_of validator for type <class 'str'>> to <instance_of␣
↪validator for type <class 'int'>>>, repr=True, cmp=True, hash=None, init=True,␣
↪metadata=mappingproxy({}), type=None, converter=None, kw_only=False), <class 'int
↪'>, 1.0)
>>> C(x={"a": 1, 7: 2})
Traceback (most recent call last):
    ...
TypeError: ("'x' must be <class 'str'> (got 7 that is a <class 'int'>).",␣
↪Attribute(name='x', default=NOTHING, validator=<deep_mapping validator for␣
↪objects mapping <instance_of validator for type <class 'str'>> to <instance_of␣
↪validator for type <class 'int'>>>, repr=True, cmp=True, hash=None, init=True,␣
↪metadata=mappingproxy({}), type=None, converter=None, kw_only=False), <class 'str
↪'>, 7)
```

Validators can be both globally and locally disabled:

attrs.validators.**set_disabled**(*disabled*)

Globally disable or enable running validators.

By default, they are run.

Parameters
disabled (*bool*) – If `True`, disable running all validators.

> **Warning:** This function is not thread-safe!

New in version 21.3.0.

attrs.validators.**get_disabled**()

Return a bool indicating whether validators are currently disabled or not.

Returns
`True` if validators are currently disabled.

Return type
bool

New in version 21.3.0.

attrs.validators.**disabled**()

Context manager that disables running validators within its context.

> **Warning:** This context manager is not thread-safe!

New in version 21.3.0.

4.8.5 Converters

All objects from `attrs.converters` are also available from `attr.converters`.

`attrs.converters.pipe(*converters)`

A converter that composes multiple converters into one.

When called on a value, it runs all wrapped converters, returning the *last* value.

Type annotations will be inferred from the wrapped converters', if they have any.

Parameters
converters (*callables*) – Arbitrary number of converters.

New in version 20.1.0.

For convenience, it's also possible to pass a list to *attr.ib*'s converter argument.

Thus the following two statements are equivalent:

```
x = attr.ib(converter=attr.converter.pipe(c1, c2, c3))
x = attr.ib(converter=[c1, c2, c3])
```

`attrs.converters.optional(converter)`

A converter that allows an attribute to be optional. An optional attribute is one which can be set to `None`.

Type annotations will be inferred from the wrapped converter's, if it has any.

Parameters
converter (*callable*) – the converter that is used for non-`None` values.

New in version 17.1.0.

For example:

```
>>> @attr.s
... class C:
...     x = attr.ib(converter=attr.converters.optional(int))
>>> C(None)
C(x=None)
>>> C(42)
C(x=42)
```

`attrs.converters.default_if_none(default=NOTHING, factory=None)`

A converter that allows to replace `None` values by *default* or the result of *factory*.

Parameters

- **default** – Value to be used if `None` is passed. Passing an instance of *attrs.Factory* is supported, however the `takes_self` option is *not*.
- **factory** (*callable*) – A callable that takes no parameters whose result is used if `None` is passed.

Raises

- `TypeError` – If **neither** *default* or *factory* is passed.
- `TypeError` – If **both** *default* and *factory* are passed.
- `ValueError` – If an instance of *attrs.Factory* is passed with `takes_self=True`.

New in version 18.2.0.

For example:

```
>>> @attr.s
... class C:
...     x = attr.ib(
...         converter=attr.converters.default_if_none("")
...     )
>>> C(None)
C(x='')
```

attrs.converters.**to_bool**(*val*)

Convert "boolean" strings (e.g., from env. vars.) to real booleans.

Values mapping to `True`:

- `True`
- `"true"` / `"t"`
- `"yes"` / `"y"`
- `"on"`
- `"1"`
- `1`

Values mapping to `False`:

- `False`
- `"false"` / `"f"`
- `"no"` / `"n"`
- `"off"`
- `"0"`
- `0`

Raises

`ValueError` – for any other value.

New in version 21.3.0.

For example:

```
>>> @attr.s
... class C:
...     x = attr.ib(
...         converter=attr.converters.to_bool
...     )
>>> C("yes")
C(x=True)
>>> C(0)
C(x=False)
>>> C("foo")
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
ValueError: Cannot convert value to bool: foo
```

4.8.6 Setters

These are helpers that you can use together with *attrs.define*'s and *attrs.fields*'s `on_setattr` arguments. All setters in `attrs.setters` are also available from `attr.setters`.

`attrs.setters.frozen(_, __, ___)`

Prevent an attribute to be modified.

New in version 20.1.0.

`attrs.setters.validate(instance, attrib, new_value)`

Run *attrib*'s validator on *new_value* if it has one.

New in version 20.1.0.

`attrs.setters.convert(instance, attrib, new_value)`

Run *attrib*'s converter – if it has one – on *new_value* and return the result.

New in version 20.1.0.

`attrs.setters.pipe(*setters)`

Run all *setters* and return the return value of the last one.

New in version 20.1.0.

`attrs.setters.NO_OP`

Sentinel for disabling class-wide *on_setattr* hooks for certain attributes.

Does not work in *attrs.setters.pipe* or within lists.

New in version 20.1.0.

For example, only `x` is frozen here:

```
>>> @attrs.define(on_setattr=attr.setters.frozen)
... class C:
...     x = attr.field()
...     y = attr.field(on_setattr=attr.setters.NO_OP)
>>> c = C(1, 2)
>>> c.y = 3
>>> c.y
3
>>> c.x = 4
Traceback (most recent call last):
    ...
attrs.exceptions.FrozenAttributeError: ()
```

N.B. Please use *attrs.define*'s *frozen* argument (or *attrs.frozen*) to freeze whole classes; it is more efficient.

4.8.7 Deprecated APIs

To help you write backward compatible code that doesn't throw warnings on modern releases, the `attr` module has an `__version_info__` attribute as of version 19.2.0. It behaves similarly to `sys.version_info` and is an instance of *VersionInfo*:

class `attr.`**VersionInfo**(*year: int*, *minor: int*, *micro: int*, *releaselevel: str*)

A version object that can be compared to tuple of length 1–4:

```
>>> attr.VersionInfo(19, 1, 0, "final")  <= (19, 2)
True
>>> attr.VersionInfo(19, 1, 0, "final") < (19, 1, 1)
True
>>> vi = attr.VersionInfo(19, 2, 0, "final")
>>> vi < (19, 1, 1)
False
>>> vi < (19,)
False
>>> vi == (19, 2,)
True
>>> vi == (19, 2, 1)
False
```

New in version 19.2.

With its help you can write code like this:

```
>>> if getattr(attr, "__version_info__", (0,)) >= (19, 2):
...     cmp_off = {"eq": False}
... else:
...     cmp_off = {"cmp": False}
>>> cmp_off == {"eq":  False}
True
>>> @attr.s(**cmp_off)
... class C:
...     pass
```

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.

`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.

Deprecated since version 17.1.0: Use *attrs.evolve* instead if you can. This function will not be removed du to the slightly different approach compared to *attrs.evolve*.

4.9 Extending

Each `attrs`-decorated class has a `__attrs_attrs__` class attribute. It's a tuple of *attrs.Attribute* carrying metadata about each attribute.

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

```
>>> from attr import define
>>> def print_attrs(cls):
...     print(cls.__attrs_attrs__)
...     return cls
>>> @print_attrs
... @define
... class C:
...     a: int
(Attribute(name='a', default=NOTHING, validator=None, repr=True, eq=True, eq_key=None,␣
↪order=True, order_key=None, hash=None, init=True, metadata=mappingproxy({}), type=
↪<class 'int'>, converter=None, kw_only=False, inherited=False, on_setattr=None),)
```

Warning: The *attrs.define*/*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))
```

4.9.1 Wrapping the Decorator

A more elegant way can be to wrap `attrs` altogether and build a class DSL on top of it.

An example for that is the package environ-config that uses `attrs` under the hood to define environment-based configurations declaratively without exposing `attrs` APIs at all.

Another common use case is to overwrite `attrs`'s defaults.

Mypy

Unfortunately, decorator wrapping currently confuses mypy's `attrs` plugin. At the moment, the best workaround is to hold your nose, write a fake mypy plugin, and mutate a bunch of global variables:

```
from mypy.plugin import Plugin
from mypy.plugins.attrs import (
   attr_attrib_makers,
   attr_class_makers,
   attr_dataclass_makers,
)

# These work just like `attr.dataclass`.
attr_dataclass_makers.add("my_module.method_looks_like_attr_dataclass")

# This works just like `attr.s`.
attr_class_makers.add("my_module.method_looks_like_attr_s")

# These are our `attr.ib` makers.
attr_attrib_makers.add("my_module.method_looks_like_attrib")

class MyPlugin(Plugin):
    # Our plugin does nothing but it has to exist so this file gets loaded.
    pass


def plugin(version):
    return MyPlugin
```

Then tell mypy about your plugin using your project's `mypy.ini`:

```
[mypy]
plugins=<path to file>
```

Warning: Please note that it is currently *impossible* to let mypy know that you've changed defaults like *eq* or *order*. You can only use this trick to tell mypy that a class is actually an `attrs` class.

Pyright

Generic decorator wrapping is supported in pyright via their dataclass_transform specification.

For a custom wrapping of the form:

```
def custom_define(f):
    return attr.define(f)
```

This is implemented via a `__dataclass_transform__` type decorator in the custom extension's `.pyi` of the form:

```
def __dataclass_transform__(
    *,
    eq_default: bool = True,
    order_default: bool = False,
```

(continues on next page)

(continued from previous page)

```
    kw_only_default: bool = False,
    field_descriptors: Tuple[Union[type, Callable[..., Any]], ...] = ((),),
) -> Callable[[_T], _T]: ...

@__dataclass_transform__(field_descriptors=(attr.attrib, attr.field))
def custom_define(f): ...
```

Warning: `dataclass_transform` is supported **provisionally** as of `pyright` 1.1.135.

Both the `pyright` dataclass_transform specification and `attrs` implementation may change in future versions.

4.9.2 Types

`attrs` offers two ways of attaching type information to attributes:

- **PEP 526** annotations on Python 3.6 and later,
- and the *type* argument to *`attr.ib`*.

This information is available to you:

```
>>> from attr import attrib, define, field, fields
>>> @define
... class C:
...     x: int = field()
...     y = attrib(type=str)
>>> fields(C).x.type
<class 'int'>
>>> fields(C).y.type
<class 'str'>
```

Currently, `attrs` doesn't do anything with this information but it's very useful if you'd like to write your own validators or serializers!

4.9.3 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

  @define
  class C:
    x = field(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 `field` wrappers for your specific metadata. This is a more graceful approach if your users don't require metadata from other libraries.

```
>>> from attr import fields, NOTHING
>>> MY_TYPE_METADATA = '__my_type_metadata'
>>>
>>> def typed(
...     cls, default=NOTHING, validator=None, repr=True,
...     eq=True, order=None, hash=None, init=True, metadata=None,
...     converter=None
... ):
...     metadata = metadata or {}
...     metadata[MY_TYPE_METADATA] = cls
...     return field(
...         default=default, validator=validator, repr=repr,
...         eq=eq, order=order, hash=hash, init=init,
...         metadata=metadata, converter=converter
...     )
>>>
>>> @define
... class C:
...     x: int = typed(int, default=1, init=False)
>>> fields(C).x.metadata[MY_TYPE_METADATA]
<class 'int'>
```

4.9.4 Automatic Field Transformation and Modification

`attrs` allows you to automatically modify or transform the class' fields while the class is being created. You do this by passing a *field_transformer* hook to *attr.define* (and its friends). Its main purpose is to automatically add converters to attributes based on their type to aid the development of API clients and other typed data loaders.

This hook must have the following signature:

your_hook(*cls: type*, *fields: list[*attrs.Attribute*]*) → list[*attrs.Attribute*]

- *cls* is your class right *before* it is being converted into an attrs class. This means it does not yet have the `__attrs_attrs__` attribute.
- *fields* is a list of all *attrs.Attribute* instances that will later be set to `__attrs_attrs__`. You can modify these attributes any way you want: You can add converters, change types, and even remove attributes completely or create new ones!

For example, let's assume that you really don't like floats:

```
>>> def drop_floats(cls, fields):
...     return [f for f in fields if f.type not in {float, 'float'}]
...
>>> @frozen(field_transformer=drop_floats)
... class Data:
...     a: int
...     b: float
...     c: str
...
>>> Data(42, "spam")
Data(a=42, c='spam')
```

A more realistic example would be to automatically convert data that you, e.g., load from JSON:

```
>>> from datetime import datetime
>>>
>>> def auto_convert(cls, fields):
...     results = []
...     for field in fields:
...         if field.converter is not None:
...             results.append(field)
...             continue
...         if field.type in {datetime, 'datetime'}:
...             converter = (lambda d: datetime.fromisoformat(d) if isinstance(d, str)␣
↪else d)
...         else:
...             converter = None
...         results.append(field.evolve(converter=converter))
...     return results
...
>>> @frozen(field_transformer=auto_convert)
... class Data:
...     a: int
...     b: str
...     c: datetime
...
>>> from_json = {"a": 3, "b": "spam", "c": "2020-05-04T13:37:00"}
>>> Data(**from_json)  # ****
Data(a=3, b='spam', c=datetime.datetime(2020, 5, 4, 13, 37))
```

4.9.5 Customize Value Serialization in `asdict()`

`attrs` allows you to serialize instances of `attrs` classes to dicts using the *`attrs.asdict`* function. However, the result can not always be serialized since most data types will remain as they are:

```
>>> import json
>>> import datetime
>>> from attrs import asdict
>>>
>>> @frozen
... class Data:
...    dt: datetime.datetime
...
>>> data = asdict(Data(datetime.datetime(2020, 5, 4, 13, 37)))
>>> data
{'dt': datetime.datetime(2020, 5, 4, 13, 37)}
>>> json.dumps(data)
Traceback (most recent call last):
  ...
TypeError: Object of type datetime is not JSON serializable
```

To help you with this, *`attr.asdict`* allows you to pass a *value_serializer* hook. It has the signature

your_hook(*inst:* *type*, *field:* attrs.Attribute, *value:* *Any*) → Any

```
>>> from attr import asdict
>>> def serialize(inst, field, value):
...     if isinstance(value, datetime.datetime):
...         return value.isoformat()
...     return value
...
>>> data = asdict(
...     Data(datetime.datetime(2020, 5, 4, 13, 37)),
...     value_serializer=serialize,
... )
>>> data
{'dt': '2020-05-04T13:37:00'}
>>> json.dumps(data)
'{"dt": "2020-05-04T13:37:00"}'
```

4.10 How Does It Work?

4.10.1 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 `@attrs.define` (or `@attr.s`) decorator to a class, `attrs` searches the class object for instances of `attr.ib`s. Internally they're a representation of the data passed into `attr.ib` along with a counter to preserve the order of the attributes. Alternatively, it's possible to define them using *Type Annotations*.

In order to ensure that subclassing works as you'd expect it to work, `attrs` also walks the class hierarchy and collects the attributes of all base 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 fewer function calls.

Once `attrs` knows what attributes it has to work on, it writes the requested *dunder methods* and – depending on whether you wish to have a *dict* or *slotted* class – creates a new class for you (`slots=True`) or attaches them to the original class (`slots=False`). While creating new classes is more elegant, we've run into several edge cases surrounding metaclasses that make it impossible to go this route unconditionally.

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 inspect
>>> from attr import define
>>> @define
... class C:
...     x: int
>>> 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 `attrs` uses internally.

Much of the information is accessible via *attrs.fields* and other functions which can be used for introspection or for writing your own tools and decorators on top of `attrs` (like *attrs.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.

4.10.2 Immutability

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

The same is true if you choose to freeze individual attributes using the *attrs.setters.frozen* *on_setattr* hook – except that the exception becomes *attrs.exceptions.FrozenAttributeError*.

Both errors subclass *attrs.exceptions.FrozenError*.

Depending on whether a class is a dict class or a slotted class, `attrs` uses a different technique to circumvent that limitation in the `__init__` method.

Once constructed, frozen instances don't differ in any way from regular ones except that you cannot change its attributes.

Dict Classes

Dict classes – i.e. regular classes – simply assign the value directly into the class' eponymous `__dict__` (and there's nothing we can do to stop the user to do the same).

The performance impact is negligible.

Slotted Classes

Slotted classes are more complicated. Here it 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)"
.....................................
Mean +- std dev: 228 ns +- 18 ns

$ pyperf timeit --rigorous \
      -s "import attr; C = attr.make_class('C', ['x', 'y', 'z'], slots=True, frozen=True)
↪" \
      "C(1, 2, 3)"
.....................................
Mean +- std dev: 450 ns +- 26 ns
```

So on a laptop computer the difference is about 230 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.

Summary

You should avoid instantiating lots of frozen slotted classes (i.e. `@frozen`) in performance-critical code.

Frozen dict classes have barely a performance impact, unfrozen slotted classes are even *faster* than unfrozen dict classes (i.e. regular classes).

4.11 On The Core API Names

You may be surprised seeing `attrs` classes being created using *attrs.define* and with type annotated fields, instead of *attr.s* and *attr.ib()*.

Or, you wonder why the web and talks are full of this weird *attr.s* and *attr.ib* – including people having strong opinions about it and using `attr.attrs` and `attr.attrib` instead.

And what even is `attr.dataclass` that's not documented but commonly used!?

4.11.1 TL;DR

We recommend our modern APIs for new code:

- *attrs.define()* to define a new class,
- *attrs.mutable()* is an alias for *attrs.define()*,
- *attrs.frozen()* is an alias for `define(frozen=True)`
- and *attrs.field()* to define an attribute.

They have been added in `attrs` 20.1.0, they are expressive, and they have modern defaults like slots and type annotation awareness switched on by default. They are only available in Python 3.6 and later. Sometimes they're referred to as *next-generation* or *NG* APIs. As of `attrs` 21.3.0 you can also import them from the `attrs` package namespace.

The traditional APIs *attr.s* / *attr.ib*, their serious business aliases `attr.attrs` / `attr.attrib`, and the never-documented, but popular `attr.dataclass` easter egg will stay **forever**.

`attrs` will **never** force you to use type annotations.

4.11.2 A Short History Lesson

At this point, `attrs` is an old project. It had its first release in April 2015 – back when most Python code was on Python 2.7 and Python 3.4 was the first Python 3 release that showed promise. `attrs` was always Python 3-first, but type annotations came only into Python 3.5 that was released in September 2015 and were largely ignored until years later.

At this time, if you didn't want to implement all the *dunder methods*, the most common way to create a class with some attributes on it was to subclass a `collections.namedtuple`, or one of the many hacks that allowed you to access dictionary keys using attribute lookup.

But `attrs` history goes even a bit further back, to the now-forgotten characteristic that came out in May 2014 and already used a class decorator, but was overall too unergonomic.

In the wake of all of that, glyph and Hynek came together on IRC and brainstormed how to take the good ideas of `characteristic`, but make them easier to use and read. At this point the plan was not to make `attrs` what it is now – a flexible class building kit. All we wanted was an ergonomic little library to succinctly define classes with attributes.

Under the impression of the unwieldy `characteristic` name, we went to the other side and decided to make the package name part of the API, and keep the API functions very short. This led to the infamous *attr.s* and *attr.ib*

which some found confusing and pronounced it as "attr dot s" or used a singular `@s` as the decorator. But it was really just a way to say `attrs` and `attrib`[1].

Some people hated this cutey API from day one, which is why we added aliases for them that we called *serious business*: `@attr.attrs` and `attr.attrib()`. Fans of them usually imported the names and didn't use the package name in the first place. Unfortunately, the `attr` package name started creaking the moment we added `attr.Factory`, since it couldn't be morphed into something meaningful in any way. A problem that grew worse over time, as more APIs and even modules were added.

But overall, `attrs` in this shape was a **huge** success – especially after glyph's blog post The One Python Library Everyone Needs in August 2016 and pytest adopting it.

Being able to just write:

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

was a big step for those who wanted to write small, focused classes.

Dataclasses Enter The Arena

A big change happened in May 2017 when Hynek sat down with Guido van Rossum and Eric V. Smith at PyCon US 2017.

Type annotations for class attributes have just landed in Python 3.6 and Guido felt like it would be a good mechanic to introduce something similar to `attrs` to the Python standard library. The result, of course, was **PEP 557**[2] which eventually became the `dataclasses` module in Python 3.7.

`attrs` at this point was lucky to have several people on board who were also very excited about type annotations and helped implement it; including a Mypy plugin. And so it happened that `attrs` shipped the new method of defining classes more than half a year before Python 3.7 – and thus `dataclasses` – were released.

Due to backward-compatibility concerns, this feature is off by default in the *`attr.s`* decorator and has to be activated using `@attr.s(auto_attribs=True)`, though. As a little easter egg and to save ourselves some typing, we've also added an alias called `attr.dataclasses` that just set `auto_attribs=True`. It was never documented, but people found it and used it and loved it.

Over the next months and years it became clear that type annotations have become the popular way to define classes and their attributes. However, it has also become clear that some people viscerally hate type annotations. We're determined to serve both.

`attrs` TNG

Over its existence, `attrs` never stood still. But since we also greatly care about backward compatibility and not breaking our users' code, many features and niceties have to be manually activated.

That is not only annoying, it also leads to the problem that many of `attrs`'s users don't even know what it can do for them. We've spent years alone explaining that defining attributes using type annotations is in no way unique to `dataclasses`.

Finally we've decided to take the Go route: instead of fiddling with the old APIs – whose names felt anachronistic anyway – we'd define new ones, with better defaults. So in July 2018, we looked for better names and came up with

[1] We considered calling the PyPI package just `attr` too, but the name was already taken by an *ostensibly* inactive package on PyPI.
[2] The highly readable PEP also explains why `attrs` wasn't just added to the standard library. Don't believe the myths and rumors.

attr.define, *attr.field*, and friends. Then in January 2019, we started looking for inconvenient defaults that we now could fix without any repercussions.

These APIs proved to be very popular, so we've finally changed the documentation to them in November of 2021.

All of this took way too long, of course. One reason is the COVID-19 pandemic, but also our fear to fumble this historic chance to fix our APIs.

Finally, in December 2021, we've added the `attrs` package namespace.

We hope you like the result:

```
from attrs import define

@define
class Point:
    x: int
    y: int
```

4.12 Glossary

dunder methods
"Dunder" is a contraction of "double underscore".

It's methods like `__init__` or `__eq__` that are sometimes also called *magic methods* or it's said that they implement an *object protocol*.

In spoken form, you'd call `__init__` just "dunder init".

Its first documented use is a mailing list posting by Mark Jackson from 2002.

dict classes
A regular class whose attributes are stored in the `object.__dict__` attribute of every single instance. This is quite wasteful especially for objects with very few data attributes and the space consumption can become significant when creating large numbers of instances.

This is the type of class you get by default both with and without `attrs` (except with the next APIs *attr.define*, *attr.mutable*, and *attr.frozen*).

slotted classes
A class whose instances have no `object.__dict__` attribute and define their attributes in a `object.__slots__` attribute instead. In `attrs`, they are created by passing `slots=True` to `@attr.s` (and are on by default in *attr.define*/*attr.mutable*/*attr.frozen*).

Their main advantage is that they use less memory on CPython[1] and are slightly faster.

However they also come with several possibly surprising gotchas:

- Slotted classes don't allow for any other attribute to be set except for those defined in one of the class' hierarchies `__slots__`:

```
>>> from attr import define
>>> @define
... class Coordinates:
...     x: int
...     y: int
```

(continues on next page)

[1] On PyPy, there is no memory advantage in using slotted classes.

(continued from previous page)

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

- Slotted classes can inherit from other classes just like non-slotted classes, but some of the benefits of slotted classes are lost if you do that. If you must inherit from other classes, try to inherit only from other slotted classes.
- However, it's not possible to inherit from more than one class that has attributes in `__slots__` (you will get an `TypeError: multiple bases have instance lay-out conflict`).
- It's not possible to monkeypatch methods on slotted classes. This can feel limiting in test code, however the need to monkeypatch your own classes is usually a design smell.

  If you really need to monkeypatch an instance in your tests, but don't want to give up on the advantages of slotted classes in production code, you can always subclass a slotted class as a dict class with no further changes and all the limitations go away:

```
>>> import attr, unittest.mock
>>> @define
... class Slotted:
...     x: int
...
...     def method(self):
...         return self.x
>>> s = Slotted(42)
>>> s.method()
42
>>> with unittest.mock.patch.object(s, "method", return_value=23):
...     pass
Traceback (most recent call last):
   ...
AttributeError: 'Slotted' object attribute 'method' is read-only
>>> @define(slots=False)
... class Dicted(Slotted):
...     pass
>>> d = Dicted(42)
>>> d.method()
42
>>> with unittest.mock.patch.object(d, "method", return_value=23):
...     assert 23 == d.method()
```

- Slotted classes must implement `__getstate__` and `__setstate__` to be serializable with `pickle` protocol 0 and 1. Therefore, `attrs` creates these methods automatically for `slots=True` classes.

Note: If the `@attr.s(slots=True)` decorated class already implements the `__getstate__` and `__setstate__` methods, they will be *overwritten* by `attrs` autogenerated implementation by default.

This can be avoided by setting `@attr.s(getstate_setstate=False)` or by setting `@attr.s(auto_detect=True)`.

Also, think twice before using `pickle`.

- Slotted classes are weak-referenceable by default. This can be disabled in CPython by passing `weakref_slot=False` to `@attr.s`[2].
- Since it's currently impossible to make a class slotted after it's been created, `attrs` has to replace your class with a new one. While it tries to do that as graciously as possible, certain metaclass features like `object.__init_subclass__` do not work with slotted classes.
- The `class.__subclasses__` attribute needs a garbage collection run (which can be manually triggered using `gc.collect`), for the original class to be removed. See issue #407 for more details.

[2] On PyPy, slotted classes are naturally weak-referenceable so `weakref_slot=False` has no effect.

CHAPTER

FIVE

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 3.5+ and PyPy. The last version with Python 2.7 support is 21.4.0.

We collect information on **third-party extensions** in our wiki. Feel free to browse and add your own!

If you'd like to contribute to `attrs` you're most welcome and we've written a little guide to get you started!

5.1 `attrs` for Enterprise

Available as part of the Tidelift Subscription.

The maintainers of `attrs` and thousands of other packages are working with Tidelift to deliver commercial support and maintenance for the open source packages you use to build your applications. Save time, reduce risk, and improve code health, while paying the maintainers of the exact packages you use. Learn more.

5.1.1 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 it clunkiness and unfortunate decisions. Both were inspired by Twisted's FancyEqMixin but both are implemented using class decorators because subclassing is bad for you, m'kay?

5.1.2 Changelog

Versions follow CalVer with a strict backwards-compatibility policy.

The **first number** of the version is the year. The **second number** is incremented with each release, starting at 1 for each year. The **third number** is when we need to start branches for older releases (only for emergencies).

Put simply, you shouldn't ever be afraid to upgrade `attrs` if you're only using its public APIs. Whenever there is a need to break compatibility, it is announced here in the changelog, and raises a `DeprecationWarning` for a year (if possible) before it's finally really broken.

> **Warning:** The structure of the *`attrs.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.

Changes for the upcoming release can be found in the "changelog.d" directory in our repository.

21.4.0 (2021-12-29)

Changes

- Fixed the test suite on PyPy3.8 where `cloudpickle` does not work. #892
- Fixed `coverage report` for projects that use `attrs` and don't set a `--source`. #895, #896

21.3.0 (2021-12-28)

Backward-incompatible Changes

- When using `@define`, converters are now run by default when setting an attribute on an instance – additionally to validators. I.e. the new default is `on_setattr=[attrs.setters.convert, attrs.setters.validate]`.

  This is unfortunately a breaking change, but it was an oversight, impossible to raise a `DeprecationWarning` about, and it's better to fix it now while the APIs are very fresh with few users. #835, #886

- `import attrs` has finally landed! As of this release, you can finally import `attrs` using its proper name.

  Not all names from the `attr` namespace have been transferred; most notably `attr.s` and `attr.ib` are missing. See `attrs.define` and `attrs.field` if you haven't seen our next-generation APIs yet. A more elaborate explanation can be found On The Core API Names

  This feature is at least for one release **provisional**. We don't *plan* on changing anything, but such a big change is unlikely to go perfectly on the first strike.

  The API docs have been mostly updated, but it will be an ongoing effort to change everything to the new APIs. Please note that we have **not** moved – or even removed – anything from `attr`!

  Please do report any bugs or documentation inconsistencies! #887

Changes

- `attr.asdict(retain_collection_types=False)` (default) dumps collection-esque keys as tuples. #646, #888
- `__match_args__` are now generated to support Python 3.10's Structural Pattern Matching. This can be controlled by the `match_args` argument to the class decorators on Python 3.10 and later. On older versions, it is never added and the argument is ignored. #815
- If the class-level *on_setattr* is set to `attrs.setters.validate` (default in `@define` and `@mutable`) but no field defines a validator, pretend that it's not set. #817
- The generated `__repr__` is significantly faster on Pythons with f-strings. #819
- Attributes transformed via `field_transformer` are wrapped with `AttrsClass` again. #824
- Generated source code is now cached more efficiently for identical classes. #828
- Added `attrs.converters.to_bool()`. #830
- `attrs.resolve_types()` now resolves types of subclasses after the parents are resolved. #842 #843
- Added new validators: `lt(val)` (< val), `le(va)` (val), `ge(val)` (val), `gt(val)` (> val), and `maxlen(n)`. #845
- `attrs` classes are now fully compatible with cloudpickle (no need to disable `repr` anymore). #857
- Added new context manager `attrs.validators.disabled()` and functions `attrs.validators.(set|get)_disabled()`. They deprecate `attrs.(set|get)_run_validators()`. All functions are interoperable and modify the same internal state. They are not – and never were – thread-safe, though. #859
- `attrs.validators.matches_re()` now accepts pre-compiled regular expressions in addition to pattern strings. #877

21.2.0 (2021-05-07)

Backward-incompatible Changes

- We had to revert the recursive feature for `attr.evolve()` because it broke some use-cases – sorry! #806
- Python 3.4 is now blocked using packaging metadata because `attrs` can't be imported on it anymore. To ensure that 3.4 users can keep installing `attrs` easily, we will yank 21.1.0 from PyPI. This has **no** consequences if you pin `attrs` to 21.1.0. #807

21.1.0 (2021-05-06)

Deprecations

- The long-awaited, much-talked-about, little-delivered `import attrs` is finally upon us!

  Since the NG APIs have now been proclaimed stable, the **next** release of `attrs` will allow you to actually `import attrs`. We're taking this opportunity to replace some defaults in our APIs that made sense in 2015, but don't in 2021.

  So please, if you have any pet peeves about defaults in `attrs`'s APIs, *now* is the time to air your grievances in #487! We're not gonna get such a chance for a second time, without breaking our backward-compatibility guarantees, or long deprecation cycles. Therefore, speak now or forever hold you peace! #487

- The *cmp* argument to `attr.s()` and *attr.ib()* has been **undeprecated** It will continue to be supported as syntactic sugar to set *eq* and *order* in one go.

  I'm terribly sorry for the hassle around this argument! The reason we're bringing it back is it's usefulness regarding customization of equality/ordering.

  The `cmp` attribute and argument on `attr.Attribute` remains deprecated and will be removed later this year. #773

Changes

- It's now possible to customize the behavior of `eq` and `order` by passing in a callable. #435, #627
- The instant favorite next-generation APIs are not provisional anymore!

  They are also officially supported by Mypy as of their 0.800 release.

  We hope the next release will already contain an (additional) importable package called `attrs`. #668, #786
- If an attribute defines a converter, the type of its parameter is used as type annotation for its corresponding `__init__` parameter.

  If an `attr.converters.pipe` is used, the first one's is used. #710
- Fixed the creation of an extra slot for an `attr.ib` when the parent class already has a slot with the same name. #718
- `__attrs__init__()` will now be injected if `init=False`, or if `auto_detect=True` and a user-defined `__init__()` exists.

  This enables users to do "pre-init" work in their `__init__()` (such as `super().__init__()`).

  `__init__()` can then delegate constructor argument processing to `self.__attrs_init__(*args, **kwargs)`. #731
- `bool(attr.NOTHING)` is now `False`. #732
- It's now possible to use `super()` inside of properties of slotted classes. #747
- Allow for a `__attrs_pre_init__()` method that – if defined – will get called at the beginning of the `attrs`-generated `__init__()` method. #750
- Added forgotten `attr.Attribute.evolve()` to type stubs. #752
- `attrs.evolve()` now works recursively with nested `attrs` classes. #759
- Python 3.10 is now officially supported. #763
- `attr.resolve_types()` now takes an optional *attrib* argument to work inside a `field_transformer`. #774
- `ClassVar`s are now also detected if they come from typing-extensions. #782
- To make it easier to customize attribute comparison (#435), we have added the `attr.cmp_with()` helper.

  See the new docs on comparison for more details. #787
- Added **provisional** support for static typing in `pyright` via the dataclass_transforms specification. Both the `pyright` specification and `attrs` implementation may change in future versions of both projects.

  Your constructive feedback is welcome in both attrs#795 and pyright#1782. #796

20.3.0 (2020-11-05)

Backward-incompatible Changes

- `attr.define()`, `attr.frozen()`, `attr.mutable()`, and `attr.field()` remain **provisional**.

  This release does **not** change anything about them and they are already used widely in production though.

  If you wish to use them together with mypy, you can simply drop this plugin into your project.

  Feel free to provide feedback to them in the linked issue #668.

  We will release the `attrs` namespace once we have the feeling that the APIs have properly settled. #668

Changes

- `attr.s()` now has a *field_transformer* hook that is called for all `Attribute`s and returns a (modified or updated) list of `Attribute` instances. `attr.asdict()` has a *value_serializer* hook that can change the way values are converted. Both hooks are meant to help with data (de-)serialization workflows. #653
- `kw_only=True` now works on Python 2. #700
- `raise from` now works on frozen classes on PyPy. #703, #712
- `attr.asdict()` and `attr.astuple()` now treat `frozenset`s like `set`s with regards to the *retain_collection_types* argument. #704
- The type stubs for `attr.s()` and `attr.make_class()` are not missing the *collect_by_mro* argument anymore. #711

20.2.0 (2020-09-05)

Backward-incompatible Changes

- `attr.define()`, `attr.frozen()`, `attr.mutable()`, and `attr.field()` remain **provisional**.

  This release fixes a bunch of bugs and ergonomics but they remain mostly unchanged.

  If you wish to use them together with mypy, you can simply drop this plugin into your project.

  Feel free to provide feedback to them in the linked issue #668.

  We will release the `attrs` namespace once we have the feeling that the APIs have properly settled. #668

Changes

- `attr.define()` et al now correct detect `__eq__` and `__ne__`. #671
- `attr.define()` et al's hybrid behavior now also works correctly when arguments are passed. #675
- It's possible to define custom `__setattr__` methods on slotted classes again. #681
- In 20.1.0 we introduced the `inherited` attribute on the `attr.Attribute` class to differentiate attributes that have been inherited and those that have been defined directly on the class.

  It has shown to be problematic to involve that attribute when comparing instances of `attr.Attribute` though, because when sub-classing, attributes from base classes are suddenly not equal to themselves in a super class.

Therefore the `inherited` attribute will now be ignored when hashing and comparing instances of `attr.Attribute`. #684

- `zope.interface` is now a "soft dependency" when running the test suite; if `zope.interface` is not installed when running the test suite, the interface-related tests will be automatically skipped. #685
- The ergonomics of creating frozen classes using `@define(frozen=True)` and sub-classing frozen classes has been improved: you don't have to set `on_setattr=None` anymore. #687

20.1.0 (2020-08-20)

Backward-incompatible Changes

- Python 3.4 is not supported anymore. It has been unsupported by the Python core team for a while now, its PyPI downloads are negligible, and our CI provider removed it as a supported option.

  It's very unlikely that `attrs` will break under 3.4 anytime soon, which is why we do *not* block its installation on Python 3.4. But we don't test it anymore and will block it once someone reports breakage. #608

Deprecations

- Less of a deprecation and more of a heads up: the next release of `attrs` will introduce an `attrs` namespace. That means that you'll finally be able to run `import attrs` with new functions that aren't cute abbreviations and that will carry better defaults.

  This should not break any of your code, because project-local packages have priority before installed ones. If this is a problem for you for some reason, please report it to our bug tracker and we'll figure something out.

  The old `attr` namespace isn't going anywhere and its defaults are not changing – this is a purely additive measure. Please check out the linked issue for more details.

  These new APIs have been added *provisionally* as part of #666 so you can try them out today and provide feedback. Learn more in the API docs. #408

Changes

- Added `attr.resolve_types()`. It ensures that all forward-references and types in string form are resolved into concrete types.

  You need this only if you need concrete types at runtime. That means that if you only use types for static type checking, you do **not** need this function. #288, #302

- Added `@attr.s(collect_by_mro=False)` argument that if set to `True` fixes the collection of attributes from base classes.

  It's only necessary for certain cases of multiple-inheritance but is kept off for now for backward-compatibility reasons. It will be turned on by default in the future.

  As a side-effect, `attr.Attribute` now *always* has an `inherited` attribute indicating whether an attribute on a class was directly defined or inherited. #428, #635

- On Python 3, all generated methods now have a docstring explaining that they have been created by `attrs`. #506

- It is now possible to prevent `attrs` from auto-generating the `__setstate__` and `__getstate__` methods that are required for pickling of slotted classes.

  Either pass `@attr.s(getstate_setstate=False)` or pass `@attr.s(auto_detect=True)` and implement them yourself: if `attrs` finds either of the two methods directly on the decorated class, it assumes implicitly `getstate_setstate=False` (and implements neither).

  This option works with dict classes but should never be necessary. #512, #513, #642

- Fixed a `ValueError: Cell is empty` bug that could happen in some rare edge cases. #590

- `attrs` can now automatically detect your own implementations and infer `init=False`, `repr=False`, `eq=False`, `order=False`, and `hash=False` if you set `@attr.s(auto_detect=True)`. `attrs` will ignore inherited methods. If the argument implies more than one method (e.g. `eq=True` creates both `__eq__` and `__ne__`), it's enough for *one* of them to exist and `attrs` will create *neither*.

  This feature requires Python 3. #607

- Added `attr.converters.pipe()`. The feature allows combining multiple conversion callbacks into one by piping the value through all of them, and retuning the last result.

  As part of this feature, we had to relax the type information for converter callables. #618

- Fixed serialization behavior of non-slots classes with `cache_hash=True`. The hash cache will be cleared on operations which make "deep copies" of instances of classes with hash caching, though the cache will not be cleared with shallow copies like those made by `copy.copy()`.

  Previously, `copy.deepcopy()` or serialization and deserialization with `pickle` would result in an un-initialized object.

  This change also allows the creation of `cache_hash=True` classes with a custom `__setstate__`, which was previously forbidden (#494). #620

- It is now possible to specify hooks that are called whenever an attribute is set **after** a class has been instantiated.

  You can pass `on_setattr` both to `@attr.s()` to set the default for all attributes on a class, and to `@attr.ib()` to overwrite it for individual attributes.

  `attrs` also comes with a new module `attr.setters` that brings helpers that run validators, converters, or allow to freeze a subset of attributes. #645, #660

- **Provisional** APIs called `attr.define()`, `attr.mutable()`, and `attr.frozen()` have been added.

  They are only available on Python 3.6 and later, and call `attr.s()` with different default values.

  If nothing comes up, they will become the official way for creating classes in 20.2.0 (see above).

  Please note that it may take some time until mypy – and other tools that have dedicated support for `attrs` – recognize these new APIs. Please **do not** open issues on our bug tracker, there is nothing we can do about it. #666

- We have also provisionally added `attr.field()` that supplants `attr.ib()`. It also requires at least Python 3.6 and is keyword-only. Other than that, it only dropped a few arguments, but changed no defaults.

  As with `attr.s()`: `attr.ib()` is not going anywhere. #669

19.3.0 (2019-10-15)

Changes

- Fixed `auto_attribs` usage when default values cannot be compared directly with `==`, such as `numpy` arrays. #585

19.2.0 (2019-10-01)

Backward-incompatible Changes

- Removed deprecated `Attribute` attribute `convert` per scheduled removal on 2019/1. This planned deprecation is tracked in issue #307. #504
- `__lt__`, `__le__`, `__gt__`, and `__ge__` do not consider subclasses comparable anymore.

  This has been deprecated since 18.2.0 and was raising a `DeprecationWarning` for over a year. #570

Deprecations

- The `cmp` argument to `attr.s()` and `attr.ib()` is now deprecated.

  Please use `eq` to add equality methods (`__eq__` and `__ne__`) and `order` to add ordering methods (`__lt__`, `__le__`, `__gt__`, and `__ge__`) instead – just like with dataclasses.

  Both are effectively `True` by default but it's enough to set `eq=False` to disable both at once. Passing `eq=False, order=True` explicitly will raise a `ValueError` though.

  Since this is arguably a deeper backward-compatibility break, it will have an extended deprecation period until 2021-06-01. After that day, the `cmp` argument will be removed.

  `attr.Attribute` also isn't orderable anymore. #574

Changes

- Updated `attr.validators.__all__` to include new validators added in #425. #517
- Slotted classes now use a pure Python mechanism to rewrite the `__class__` cell when rebuilding the class, so `super()` works even on environments where `ctypes` is not installed. #522
- When collecting attributes using `@attr.s(auto_attribs=True)`, attributes with a default of `None` are now deleted too. #523, #556
- Fixed `attr.validators.deep_iterable()` and `attr.validators.deep_mapping()` type stubs. #533
- `attr.validators.is_callable()` validator now raises an exception `attr.exceptions.NotCallableError`, a subclass of `TypeError`, informing the received value. #536
- `@attr.s(auto_exc=True)` now generates classes that are hashable by ID, as the documentation always claimed it would. #543, #563
- Added `attr.validators.matches_re()` that checks string attributes whether they match a regular expression. #552
- Keyword-only attributes (`kw_only=True`) and attributes that are excluded from the `attrs`'s `__init__` (`init=False`) now can appear before mandatory attributes. #559

- The fake filename for generated methods is now more stable. It won't change when you restart the process. #560
- The value passed to `@attr.ib(repr=...)` can now be either a boolean (as before) or a callable. That callable must return a string and is then used for formatting the attribute by the generated `__repr__()` method. #568
- Added `attr.__version_info__` that can be used to reliably check the version of `attrs` and write forward- and backward-compatible code. Please check out the section on deprecated APIs on how to use it. #580

19.1.0 (2019-03-03)

Backward-incompatible Changes

- Fixed a bug where deserialized objects with `cache_hash=True` could have incorrect hash code values. This change breaks classes with `cache_hash=True` when a custom `__setstate__` is present. An exception will be thrown when applying the `attrs` annotation to such a class. This limitation is tracked in issue #494. #482

Changes

- Add `is_callable`, `deep_iterable`, and `deep_mapping` validators.
  - `is_callable`: validates that a value is callable
  - `deep_iterable`: Allows recursion down into an iterable, applying another validator to every member in the iterable as well as applying an optional validator to the iterable itself.
  - `deep_mapping`: Allows recursion down into the items in a mapping object, applying a key validator and a value validator to the key and value in every item. Also applies an optional validator to the mapping object itself.

  You can find them in the `attr.validators` package. #425
- Fixed stub files to prevent errors raised by mypy's `disallow_any_generics = True` option. #443
- Attributes with `init=False` now can follow after `kw_only=True` attributes. #450
- `attrs` now has first class support for defining exception classes.

  If you define a class using `@attr.s(auto_exc=True)` and subclass an exception, the class will behave like a well-behaved exception class including an appropriate `__str__` method, and all attributes additionally available in an `args` attribute. #500
- Clarified documentation for hashing to warn that hashable objects should be deeply immutable (in their usage, even if this is not enforced). #503

18.2.0 (2018-09-01)

Deprecations

- Comparing subclasses using <, >, <=, and >= is now deprecated. The docs always claimed that instances are only compared if the types are identical, so this is a first step to conform to the docs.

  Equality operators (== and `!=`) were always strict in this regard. #394

Changes

- `attrs` now ships its own PEP 484 type hints. Together with mypy's `attrs` plugin, you've got all you need for writing statically typed code in both Python 2 and 3!

  At that occasion, we've also added narrative docs about type annotations in `attrs`. #238
- Added *kw_only* arguments to `attr.ib` and `attr.s`, and a corresponding *kw_only* attribute to `attr.Attribute`. This change makes it possible to have a generated `__init__` with keyword-only arguments on Python 3, relaxing the required ordering of default and non-default valued attributes. #281, #411
- The test suite now runs with `hypothesis.HealthCheck.too_slow` disabled to prevent CI breakage on slower computers. #364, #396
- `attr.validators.in_()` now raises a `ValueError` with a useful message even if the options are a string and the value is not a string. #383
- `attr.asdict()` now properly handles deeply nested lists and dictionaries. #395
- Added `attr.converters.default_if_none()` that allows to replace `None` values in attributes. For example `attr.ib(converter=default_if_none(""))` replaces `None` by empty strings. #400, #414
- Fixed a reference leak where the original class would remain live after being replaced when `slots=True` is set. #407
- Slotted classes can now be made weakly referenceable by passing `@attr.s(weakref_slot=True)`. #420
- Added *cache_hash* option to `@attr.s` which causes the hash code to be computed once and stored on the object. #426
- Attributes can be named `property` and `itemgetter` now. #430
- It is now possible to override a base class' class variable using only class annotations. #431

18.1.0 (2018-05-03)

Changes

- `x=X(); x.cycle = x; repr(x)` will no longer raise a `RecursionError`, and will instead show as `X(x=...)`.

  #95
- `attr.ib(factory=f)` is now syntactic sugar for the common case of `attr.ib(default=attr.Factory(f))`.

  #178, #356

- Added `attr.field_dict()` to return an ordered dictionary of `attrs` attributes for a class, whose keys are the attribute names.

  #290, #349

- The order of attributes that are passed into `attr.make_class()` or the *these* argument of `@attr.s()` is now retained if the dictionary is ordered (i.e. `dict` on Python 3.6 and later, `collections.OrderedDict` otherwise).

  Before, the order was always determined by the order in which the attributes have been defined which may not be desirable when creating classes programmatically.

  #300, #339, #343

- In slotted classes, `__getstate__` and `__setstate__` now ignore the `__weakref__` attribute.

  #311, #326

- Setting the cell type is now completely best effort. This fixes `attrs` on Jython.

  We cannot make any guarantees regarding Jython though, because our test suite cannot run due to dependency incompatibilities.

  #321, #334

- If `attr.s` is passed a *these* argument, it will no longer attempt to remove attributes with the same name from the class body.

  #322, #323

- The hash of `attr.NOTHING` is now vegan and faster on 32bit Python builds.

  #331, #332

- The overhead of instantiating frozen dict classes is virtually eliminated. #336

- Generated `__init__` methods now have an `__annotations__` attribute derived from the types of the fields.

  #363

- We have restructured the documentation a bit to account for `attrs`' growth in scope. Instead of putting everything into the examples page, we have started to extract narrative chapters.

  So far, we've added chapters on initialization and hashing.

  Expect more to come!

  #369, #370

17.4.0 (2017-12-30)

Backward-incompatible Changes

- The traversal of MROs when using multiple inheritance was backward: If you defined a class `C` that subclasses `A` and `B` like `C(A, B)`, `attrs` would have collected the attributes from `B` *before* those of `A`.

  This is now fixed and means that in classes that employ multiple inheritance, the output of `__repr__` and the order of positional arguments in `__init__` changes. Because of the nature of this bug, a proper deprecation cycle was unfortunately impossible.

  Generally speaking, it's advisable to prefer `kwargs`-based initialization anyways – *especially* if you employ multiple inheritance and diamond-shaped hierarchies.

  #298, #299, #304

- The `__repr__` set by `attrs` no longer produces an `AttributeError` when the instance is missing some of the specified attributes (either through deleting or after using `init=False` on some attributes).

  This can break code that relied on `repr(attr_cls_instance)` raising `AttributeError` to check if any `attrs`-specified members were unset.

  If you were using this, you can implement a custom method for checking this:

```
def has_unset_members(self):
    for field in attr.fields(type(self)):
        try:
            getattr(self, field.name)
        except AttributeError:
            return True
    return False
```

  #308

Deprecations

- The `attr.ib(convert=callable)` option is now deprecated in favor of `attr.ib(converter=callable)`.

  This is done to achieve consistency with other noun-based arguments like *validator*.

  convert will keep working until at least January 2019 while raising a `DeprecationWarning`.

  #307

Changes

- Generated `__hash__` methods now hash the class type along with the attribute values. Until now the hashes of two classes with the same values were identical which was a bug.

  The generated method is also *much* faster now.

  #261, #295, #296

- `attr.ib`'s *metadata* argument now defaults to a unique empty `dict` instance instead of sharing a common empty `dict` for all. The singleton empty `dict` is still enforced.

  #280

- `ctypes` is optional now however if it's missing, a bare `super()` will not work in slotted classes. This should only happen in special environments like Google App Engine.

  #284, #286

- The attribute redefinition feature introduced in 17.3.0 now takes into account if an attribute is redefined via multiple inheritance. In that case, the definition that is closer to the base of the class hierarchy wins.

  #285, #287

- Subclasses of `auto_attribs=True` can be empty now.

  #291, #292

- Equality tests are *much* faster now.

  #306

- All generated methods now have correct `__module__`, `__name__`, and (on Python 3) `__qualname__` attributes.

  #309

17.3.0 (2017-11-08)

Backward-incompatible Changes

- Attributes are no longer defined on the class body.

  This means that if you define a class `C` with an attribute `x`, the class will *not* have an attribute `x` for introspection. Instead of `C.x`, use `attr.fields(C).x` or look at `C.__attrs_attrs__`. The old behavior has been deprecated since version 16.1. (#253)

Changes

- `super()` and `__class__` now work with slotted classes on Python 3. (#102, #226, #269, #270, #272)
- Added *type* argument to `attr.ib()` and corresponding `type` attribute to `attr.Attribute`.

  This change paves the way for automatic type checking and serialization (though as of this release `attrs` does not make use of it). In Python 3.6 or higher, the value of `attr.Attribute.type` can alternately be set using variable type annotations (see PEP 526). (#151, #214, #215, #239)
- The combination of `str=True` and `slots=True` now works on Python 2. (#198)
- `attr.Factory` is hashable again. (#204)
- Subclasses now can overwrite attribute definitions of their base classes.

  That means that you can – for example – change the default value for an attribute by redefining it. (#221, #229)
- Added new option *auto_attribs* to `@attr.s` that allows to collect annotated fields without setting them to `attr.ib()`.

  Setting a field to an `attr.ib()` is still possible to supply options like validators. Setting it to any other value is treated like it was passed as `attr.ib(default=value)` – passing an instance of `attr.Factory` also works as expected. (#262, #277)
- Instances of classes created using `attr.make_class()` can now be pickled. (#282)

17.2.0 (2017-05-24)

Changes:

- Validators are hashable again. Note that validators may become frozen in the future, pending availability of no-overhead frozen classes. #192

17.1.0 (2017-05-16)

To encourage more participation, the project has also been moved into a dedicated GitHub organization and everyone is most welcome to join!

`attrs` also has a logo now!

Backward-incompatible Changes:

- `attrs` will set the `__hash__()` method to `None` by default now. The way hashes were handled before was in conflict with Python's specification. This *may* break some software although this breakage is most likely just surfacing of latent bugs. You can always make `attrs` create the `__hash__()` method using `@attr.s(hash=True)`. See #136 for the rationale of this change.

  Warning: Please *do not* upgrade blindly and *do* test your software! *Especially* if you use instances as dict keys or put them into sets!

- Correspondingly, `attr.ib`'s *hash* argument is `None` by default too and mirrors the *cmp* argument as it should.

Deprecations:

- `attr.assoc()` is now deprecated in favor of `attr.evolve()` and will stop working in 2018.

Changes:

- Fix default hashing behavior. Now *hash* mirrors the value of *cmp* and classes are unhashable by default. #136 #142
- Added `attr.evolve()` that, given an instance of an `attrs` class and field changes as keyword arguments, will instantiate a copy of the given instance with the changes applied. `evolve()` replaces `assoc()`, which is now deprecated. `evolve()` is significantly faster than `assoc()`, and requires the class have an initializer that can take the field values as keyword arguments (like `attrs` itself can generate). #116 #124 #135
- `FrozenInstanceError` is now raised when trying to delete an attribute from a frozen class. #118
- Frozen-ness of classes is now inherited. #128
- `__attrs_post_init__()` is now run if validation is disabled. #130
- Added `attr.validators.in_(options)` that, given the allowed `options`, checks whether the attribute value is in it. This can be used to check constants, enums, mappings, etc. #181
- Added `attr.validators.and_()` that composes multiple validators into one. #161

- For convenience, the *validator* argument of `@attr.s` now can take a list of validators that are wrapped using `and_()`. #138
- Accordingly, `attr.validators.optional()` now can take a list of validators too. #161
- Validators can now be defined conveniently inline by using the attribute as a decorator. Check out the validator examples to see it in action! #143
- `attr.Factory()` now has a *takes_self* argument that makes the initializer to pass the partially initialized instance into the factory. In other words you can define attribute defaults based on other attributes. #165 #189
- Default factories can now also be defined inline using decorators. They are *always* passed the partially initialized instance. #165
- Conversion can now be made optional using `attr.converters.optional()`. #105 #173
- `attr.make_class()` now accepts the keyword argument `bases` which allows for subclassing. #152
- Metaclasses are now preserved with `slots=True`. #155

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
- Added `@attr.s(str=True)` that will optionally create a `__str__()` method that is identical to `__repr__()`. This is mainly useful with `Exception`s 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 `Exception`s and were puzzled by the tracebacks: this option is for you.
- `__name__` is no longer overwritten with `__qualname__` for `attr.s(slots=True)` classes. #99

16.2.0 (2016-09-17)

Changes:

- Added `attr.astuple()` that – similarly to `attr.asdict()` – returns the instance as a tuple. #77
- Converters now work with frozen classes. #76
- Instantiation of `attrs` classes with converters is now significantly faster. #80
- Pickling now works with slotted classes. #81
- `attr.assoc()` now works with slotted 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 slotted 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 are no longer supported. 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 is no longer supported by the CPython core team. 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 slotted-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:

- Added 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:

- Added `attr.validators.optional()` that wraps other validators allowing attributes to be `None`. #16
- Multi-level inheritance now works. #24
- `__repr__()` now works with non-redecorated subclasses. #20

15.0.0 (2015-04-15)

Changes:

Initial release.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

a

INDEX