attrs

Release 18.2.0

Sep 01, 2018

Contents

1 Getting Started 3

2 Day-to-Day Usage 5

3 Testimonials 7

4 Getting Help 9

5 Project Information 11

6 Full Table of Contents 25

7 Indices and tables 69

Release v18.2.0 (*What's new?*).

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

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

CHAPTER 1

Getting Started

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

```
$ 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, understand what drives attrs's design, and know what @attr.s and attr.ib() stand for.
- *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.
- Finally *Why not...* gives you a rundown of potential alternatives and why we think attrs is superior. Yes, we've heard about namedtuples 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 StackOverflow and someone will surely help you out!

CHAPTER 2

Day-to-Day Usage

- *Type Annotations* help you to write *correct* and *self-documenting* code. `attrs` has first class support for them and even allows you to drop the calls to `attr.ib()` on modern Python versions!
- 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.
- 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 3

Testimonials

Amber Hawkie Brown, Twisted Release Manager and Computer Owl:

> Writing a fully-functional class using attrs takes me less time than writing this testimonial.

Glyph Lefkowitz, creator of Twisted, Automat, and other open source software, in The One Python Library Everyone Needs:

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

Kenneth Reitz, author of Requests and Developer Advocate at DigitalOcean, (on paper no less!):

> attrs—classes for humans. I like it.

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

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

CHAPTER 4

Getting Help

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

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

CHAPTER 5

Project Information

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

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 License and Credits

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

5.1.1 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?

5.2 Backward Compatibility

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

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

Warning: The structure of the *attr.Attribute* class is exempt from this rule. It *will* change in the future, but since it should be considered read-only, that shouldn't matter.

However if you intend to build extensions on top of `attrs` you have to anticipate that.

5.3 How To Contribute

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

This document intends to make contribution more accessible by codifying tribal knowledge and expectations. Don't be afraid to open half-finished PRs, and ask questions if something is unclear!

5.3.1 Support

In case you'd like to help out but don't want to deal with GitHub, there's a great opportunity: help your fellow developers on StackOverflow!

The offical tag is `python-attrs` and helping out in support frees us up to improve `attrs` instead!

5.3.2 Workflow

- No contribution is too small! Please submit as many fixes for typos and grammar bloopers as you can!
- Try to limit each pull request to *one* change only.
- Since we squash on merge, it's up to you how you handle updates to the master branch. Whether you prefer to rebase on master or merge master into your branch, do whatever is more comfortable for you.
- *Always* add tests and docs for your code. This is a hard rule; patches with missing tests or documentation can't be merged.
- Make sure your changes pass our CI. You won't get any feedback until it's green unless you ask for it.
- Once you've addressed review feedback, make sure to bump the pull request with a short note, so we know you're done.
- Don't break backward compatibility.

5.3.3 Code

- Obey PEP 8 and PEP 257. We use the `"""`-on-separate-lines style for docstrings:

```
def func(x):
    """
    Do something.

    :param str x: A very important parameter.

    :rtype: str
    """
```

- If you add or change public APIs, tag the docstring using `.. versionadded:: 16.0.0 WHAT` or `.. versionchanged:: 16.2.0 WHAT`.

- We use isort to sort our imports, and we follow the Black code style with a line length of 79 characters. As long as you run our full tox suite before committing, or install our pre-commit hooks (ideally you'll do both – see below "Local Development Environment"), you won't have to spend any time on formatting your code at all. If you don't, CI will catch it for you – but that seems like a waste of your time!

5.3.4 Tests

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

```
x = f()

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

- To run the test suite, all you need is a recent tox. It will ensure the test suite runs with all dependencies against all Python versions just as it will on Travis CI. If you lack some Python versions, you can can always limit the environments like `tox -e py27,py35` (in that case you may want to look into pyenv, which makes it very easy to install many different Python versions in parallel).
- Write good test docstrings.
- To ensure new features work well with the rest of the system, they should be also added to our Hypothesis testing strategy, which is found in `tests/strategies.py`.
- If you've changed or added public APIs, please update our type stubs (files ending in `.pyi`).

5.3.5 Documentation

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

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

- If you start a new section, add two blank lines before and one blank line after the header, except if two headers follow immediately after each other:

```
Last line of previous section.


Header of New Top Section
-------------------------

Header of New Section
^^^^^^^^^^^^^^^^^^^^^

First line of new section.
```

- If you add a new feature, demonstrate its awesomeness on the examples page!

Changelog

If your change is noteworthy, there needs to be a changelog entry so our users can learn about it!

To avoid merge conflicts, we use the towncrier package to manage our changelog. `towncrier` uses independent files for each pull request – so called *news fragments* – instead of one monolithic changelog file. On release, those news fragments are compiled into our `CHANGELOG.rst`.

You don't need to install `towncrier` yourself, you just have to abide by a few simple rules:

- For each pull request, add a new file into `changelog.d` with a filename adhering to the `pr#.(change|deprecation|breaking).rst` schema: For example, `changelog.d/42.change.rst` for a non-breaking change that is proposed in pull request #42.
- As with other docs, please use semantic newlines within news fragments.
- Wrap symbols like modules, functions, or classes into double backticks so they are rendered in a monospace font.
- Wrap arguments into asterisks like in autodocs: *these* or *attributes*.
- If you mention functions or other callables, add parentheses at the end of their names: `attr.func()` or `attr.Class.method()`. This makes the changelog a lot more readable.
- Prefer simple past tense or constructions with "now". For example:
  - Added `attr.validators.func()`.
  - `attr.func()` now doesn't crash the Large Hadron Collider anymore when passed the *foobar* argument.
- If you want to reference multiple issues, copy the news fragment to another filename. `towncrier` will merge all news fragments with identical contents into one entry with multiple links to the respective pull requests.

Example entries:

```
Added ``attr.validators.func()``.
The feature really *is* awesome.
```

or:

```
``attr.func()`` now doesn't crash the Large Hadron Collider anymore when␣
↪passed the *foobar* argument.
The bug really *was* nasty.
```

`tox -e changelog` will render the current changelog to the terminal if you have any doubts.

5.3.6 Local Development Environment

You can (and should) run our test suite using tox. However, you'll probably want a more traditional environment as well. We highly recommend to develop using the latest Python 3 release because `attrs` tries to take advantage of modern features whenever possible.

First create a virtual environment. It's out of scope for this document to list all the ways to manage virtual environments in Python, but if you don't already have a pet way, take some time to look at tools like pew, virtualfish, and virtualenvwrapper.

Next, get an up to date checkout of the `attrs` repository:

```
$ git clone git@github.com:python-attrs/attrs.git
```

or if you want to use git via `https`:

```
$ git clone https://github.com/python-attrs/attrs.git
```

Change into the newly created directory and **after activating your virtual environment** install an editable version of `attrs` along with its tests and docs requirements:

```
$ cd attrs
$ pip install -e '.[dev]'
```

At this point,

```
$ python -m pytest
```

should work and pass, as should:

```
$ cd docs
$ make html
```

The built documentation can then be found in `docs/_build/html/`.

To avoid committing code that violates our style guide, we strongly advice you to install pre-commit[1] hooks:

```
$ pre-commit install
```

You can also run them anytime (as our tox does) using:

```
$ pre-commit run --all-files
```

5.3.7 Governance

`attrs` is maintained by team of volunteers that is always open to new members that share our vision of a fast, lean, and magic-free library that empowers programmers to write better code with less effort. If you'd like to join, just get a pull request merged and ask to be added in the very same pull request!

The simple rule is that everyone is welcome to review/merge pull requests of others but nobody is allowed to merge their own code.

Hynek Schlawack acts reluctantly as the BDFL and has the final say over design decisions.

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

Thank you for considering contributing to `attrs`!

5.4 Contributor Covenant Code of Conduct

5.4.1 Our Pledge

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

[1] pre-commit should have been installed into your virtualenv automatically when you ran `pip install -e '.[dev]'` above. If pre-commit is missing, it may be that you need to re-run `pip install -e '.[dev]'`.

5.4.2 Our Standards

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

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

Examples of unacceptable behavior by participants include:

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

5.4.3 Our Responsibilities

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

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

5.4.4 Scope

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

5.4.5 Enforcement

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

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

5.4.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at <https://www.contributor-covenant.org/version/1/4/code-of-conduct.html>.

5.5 Changelog

Versions follow CalVer with a strict backwards compatibility policy. The third digit is only for regressions.

5.5.1 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. #425
- 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

5.5.2 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 programatically.

  #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 incompatabilities.

  #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

5.5.3 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. Due to 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

5.5.4 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)

5.5.5 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

5.5.6 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

5.5.7 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

5.5.8 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

5.5.9 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

5.5.10 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.

5.5.11 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

5.5.12 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

5.5.13 15.0.0 (2015-04-15)

Changes:

Initial release.

CHAPTER 6

Full Table of Contents

6.1 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:

```
>>> import attr

>>> @attr.s
... class SomeClass(object):
...     a_number = attr.ib(default=42)
...     list_of_numbers = attr.ib(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

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

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

(continues on next page)

(continued from previous page)

```
>>> C = attr.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__`,
- a complete set of comparison methods,
- an initializer,
- and much more,

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

On Python 3.6 and later, you can often even drop the calls to `attr.ib()` by using type annotations.

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

6.1.1 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 `namedtuples` or `types.SimpleNamespace` but they're just a sub-genre of what `attrs` is good for.

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

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

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

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

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

6.1.2 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.

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

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

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

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

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

```
>>> from attr import attrs, attrib, Factory
>>> @attrs
... class SomeClass(object):
...     a_number = attrib(default=42)
...     list_of_numbers = attrib(default=Factory(list))
...
...     def hard_math(self, another_number):
...         return self.a_number + sum(self.list_of_numbers) * another_number
>>> SomeClass(1, [1, 2, 3])
SomeClass(a_number=1, list_of_numbers=[1, 2, 3])
```

Use whichever variant fits your taste better.

6.2 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!

6.2.1 ...tuples?

Readability

What makes more sense while debugging:

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

or:

```
(1, 2)
```

?

Let's add even more ambiguity:

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

or:

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

?

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

Don't get me started when you add nesting. If you've never 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.

6.2.2 . . . 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 `namedtuples` 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, `namedtuples` 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]

[1] The word is that `namedtuples` 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 `namedtuples` 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*.

- `namedtuples` have their methods *on your instances* whether you like it or not.[2]
- `namedtuples` 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* `Points` 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 `namedtuples` 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.

6.2.3 . . . 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 whose relevancy depends on your circumstances:

- `attrs` supports all mainstream Python versions, including CPython 2.7 and PyPy.
- 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, and `__slots__`, it permeates throughout all APIs.

  On the other hand, Data Classes currently do not offer any significant feature that `attrs` doesn't already have.

[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.

[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).

- `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 developement.

6.2.4 ...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.

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

is roughly

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

(continues on next page)

(continued from previous page)

```
...             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(object):
...     a = attr.ib()
...     b = attr.ib()
...
...     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.

6.3 attrs by Example

6.3.1 Basics

The simplest possible usage is:

```
>>> import attr
>>> @attr.s
... class Empty(object):
...     pass
>>> Empty()
Empty()
>>> Empty() == Empty()
True
>>> Empty() is Empty()
False
```

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
>>> C(x=1)
C(_x=1)
```

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

```
>>> @attr.s
... class C(object):
...     _x = attr.ib(init=False, default=42)
>>> C()
C(_x=42)
>>> C(23)
Traceback (most recent call last):
   ...
TypeError: __init__() takes exactly 1 argument (2 given)
```

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

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

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

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

The order of the attributes is defined by the MRO.

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

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

(continues on next page)

(continued from previous page)

```
...     class D(object):
...         pass
>>> C.D()
C.D()
```

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

Keyword-only Attributes

When using `attrs` on Python 3, you can also add keyword-only attributes:

```
>>> @attr.s
... class A:
...     a = attr.ib(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 `attr.s`, and will apply to all attributes:

```
>>> @attr.s(kw_only=True)
... class A:
...     a = attr.ib()
...     b = attr.ib()
>>> 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:

```
>>> @attr.s
... class A:
...     a = attr.ib(default=0)
>>> @attr.s
... class B(A):
...     b = attr.ib(kw_only=True)
>>> B(b=1)
B(a=0, b=1)
>>> B()
Traceback (most recent call last):
  ...
TypeError: B() missing 1 required keyword-only argument: 'b'
```

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

```
>>> @attr.s
... class A:
```

(continues on next page)

(continued from previous page)

```
...     a = attr.ib(default=0)
>>> @attr.s
... class B(A):
...     b = attr.ib()
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, convert=None,␣
↪metadata=mappingproxy({}), type=None, kw_only=False)
```

6.3.2 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):

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

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

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

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

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

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

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

6.3.3 Defaults

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

```
>>> import collections
>>> @attr.s
... class Connection(object):
...     socket = attr.ib()
...     @classmethod
...     def connect(cls, db_string):
...         # ... connect somehow to db_string ...
...         return cls(socket=42)
>>> @attr.s
... class ConnectionPool(object):
...     db_string = attr.ib()
...     pool = attr.ib(default=attr.Factory(collections.deque))
...     debug = attr.ib(default=False)
...     def get_connection(self):
...         try:
...             return self.pool.pop()
...         except IndexError:
...             if self.debug:
...                 print("New connection!")
...             return Connection.connect(self.db_string)
...     def free_connection(self, conn):
...         if self.debug:
...             print("Connection returned!")
...         self.pool.appendleft(conn)
...
>>> cp = ConnectionPool("postgres://localhost")
>>> cp
ConnectionPool(db_string='postgres://localhost', pool=deque([]), debug=False)
>>> conn = cp.get_connection()
>>> conn
Connection(socket=42)
>>> cp.free_connection(conn)
```

(continues on next page)

(continued from previous page)

```
>>> 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 a decorator. The method receives the partially initialized instance which enables you to base a default value on other attributes:

```
>>> @attr.s
... class C(object):
...     x = attr.ib(default=1)
...     y = attr.ib()
...     @y.default
...     def name_does_not_matter(self):
...         return self.x + 1
>>> C()
C(x=1, y=2)
```

And since the case of `attr.ib(default=attr.Factory(f))` is so common, `attrs` also comes with syntactic sugar for it:

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

6.3.4 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:

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

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

...or both at once:

```
>>> @attr.s
... class C(object):
...     x = attr.ib(validator=attr.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, kw_
↪only=False), <class 'int'>, '128')
>>> C(256)
Traceback (most recent call last):
   ...
ValueError: value out of bounds
```

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

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

Check out *Validators* for more details.

6.3.5 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.

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

Check out *Converters* for more details.

6.3.6 Metadata

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

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

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

If you're the author of a third-party library with `attrs` integration, please see *Extending Metadata*.

6.3.7 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:

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

If you don't mind annotating *all* attributes, you can even drop the *`attr.ib()`* and assign default values instead:

```
>>> import typing
>>> @attr.s(auto_attribs=True)
... class AutoC:
...     cls_var: typing.ClassVar[int] = 5  # this one is ignored
...     l: typing.List[int] = attr.Factory(list)
...     x: int = 1
```

(continues on next page)

(continued from previous page)

```
...     foo: str = attr.ib(
...         default="every attrib needs a type if auto_attribs=True"
...     )
...     bar: typing.Any = None
>>> attr.fields(AutoC).l.type
typing.List[int]
>>> attr.fields(AutoC).x.type
<class 'int'>
>>> attr.fields(AutoC).foo.type
<class 'str'>
>>> attr.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.

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.

6.3.8 Slots

Slotted classes have a bunch of advantages on CPython. Defining `__slots__` by hand is tedious, in `attrs` it's just a matter of passing `slots=True`:

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

6.3.9 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:

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

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

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

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

6.3.10 Other Goodies

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

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

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

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

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

```
>>> class D(object):
...     def __eq__(self, other):
...         return True  # arbitrary example
>>> C = attr.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 `@attr.s`. To do this, just define a `__attrs_post_init__` method in your class. It will get called at the end of the generated `__init__` method.

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

Finally, you can exclude single attributes from certain methods:

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

6.4 Type Annotations

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

On Python 3.6 and later, you can even drop the *`attr.ib()`*s if you're willing to annotate *all* attributes. That means that on modern Python versions, the declaration part of the example from the README can be simplified to:

```
>>> import attr
>>> import typing

>>> @attr.s(auto_attribs=True)
... class SomeClass:
...     a_number: int = 42
...     list_of_numbers: typing.List[int] = attr.Factory(list)

>>> sc = SomeClass(1, [1, 2, 3])
>>> sc
SomeClass(a_number=1, list_of_numbers=[1, 2, 3])
>>> attr.fields(SomeClass).a_number.type
<class 'int'>
```

You can still use *`attr.ib()`* for advanced features, but you don't have to.

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

6.4.1 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(object):
    a_number = attr.ib(default=42)  # type: int
    list_of_numbers = attr.ib(factory=list, type=typing.List[int])
```

The addition of static types is certainly one of the most exciting features in the Python ecosystem and helps you writing *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.

6.5 Initialization

In Python, instance intialization 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(object):
    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:

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

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

pt = Point.from_row(row)
```

Now you can instantiate `Points` 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 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 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.

6.5.1 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
>>> @attr.s
... class C(object):
...     _x = attr.ib()
>>> inspect.signature(C.__init__)
<Signature (self, x) -> 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:

```
>>> @attr.s
... class C(object):
...     _1 = attr.ib()
Traceback (most recent call last):
   ...
SyntaxError: invalid syntax
```

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

6.5.2 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:

```
>>> import attr
>>> @attr.s
... class C(object):
...     a = attr.ib(default=42)
...     b = attr.ib(default=attr.Factory(list))
...     c = attr.ib(factory=list)  # syntactic sugar for above
...     d = attr.ib()
...     @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:

```
>>> @attr.s
... class C(object):
...     x = attr.ib(default=[])
>>> i = C()
>>> j = C()
>>> i.x.append(42)
>>> j.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.

6.5.3 Validators

Another thing that definitely *does* belong into `__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.

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

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

Callables

If you want to re-use your validators, you should have a look at the `validator` argument to *attr.ib()*.

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.

Since the validators runs *after* the instance is initialized, you can refer to other attributes while validating:

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

This example also shows of some syntactic sugar for using the *attr.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 *attr.validate()* on any instance to verify that it's still valid:

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

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

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

```
>>> @attr.s
... class C(object):
...     x = attr.ib(validator=attr.validators.instance_of(int))
...     @x.validator
```

(continues on next page)

(continued from previous page)

```
...     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:

```
>>> attr.set_run_validators(False)
>>> C("128")
C(x='128')
>>> attr.set_run_validators(True)
>>> 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')
```

6.5.4 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.

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

(continues on next page)

(continued from previous page)

```
...     x = attr.ib(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'
```

6.5.5 Post-Init Hook

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 after your class is initialized. And for that `attrs` offers the `__attrs_post_init__` hook that is automatically detected and run after `attrs` is done initializing your instance:

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

```
>>> @attr.s(frozen=True)
... class FrozenBroken(object):
...     x = attr.ib()
...     y = attr.ib(init=False)
...     def __attrs_post_init__(self):
...         self.y = self.x + 1
>>> FrozenBroken(1)
Traceback (most recent call last):
   ...
attr.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__()`:

```
>>> @attr.s(frozen=True)
... class Frozen(object):
...     x = attr.ib()
...     y = attr.ib(init=False)
...     def __attrs_post_init__(self):
```

(continues on next page)

(continued from previous page)

```
...         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`.

6.6 Hashing

6.6.1 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 comparison by object identity: use `@attr.s(cmp=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 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` currently mimics 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(cmp=False)`. Setting `@attr.s(hash=False)` (that implies `cmp=True`) is almost certainly a *bug*.

2. If two object 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 fullfilled 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* responsibility to make sure that the object is not mutated.

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

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.

6.6.2 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`.

6.6.3 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.

6.7 API Reference

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

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

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

6.7.1 Core

`attr.s`(*these=None*, *repr_ns=None*, *repr=True*, *cmp=True*, *hash=None*, *init=True*, *slots=False*, *frozen=False*, *weakref_slot=True*, *str=False*, *auto_attribs=False*, *cache_hash=False*)

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

Parameters

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

  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.

- **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.
- **cmp** (*bool*) – Create `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, and `__ge__` methods that compare the class as if it were a tuple of its `attrs` attributes. But the attributes are *only* compared, if the types of both classes are *identical*!
- **hash** (`bool` or `None`) – If `None` (default), the `__hash__` method is generated according how *cmp* and *frozen* are set.

  1. If *both* are True, `attrs` will generate a `__hash__` for you.
  2. If *cmp* is True and *frozen* is False, `__hash__` will be set to None, marking it unhashable (which it is).
  3. If *cmp* 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 the Python documentation 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_post_init__` method exists on the class, it will be called after the class is fully initialized.
- **slots** (*bool*) – Create a slots-style class that's more memory-efficient. See *Slots* for further ramifications.
- **frozen** (*bool*) – Make instances immutable after initialization. If someone attempts to modify a frozen instance, *attr.exceptions.FrozenInstanceError* is raised.

  Please note:

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

Attributes annotated as `typing.ClassVar` are **ignored**.

- **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, then no attributes of this class which participate in hash code computation may be mutated after object creation.

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.

New in version 18.2.0: *kw_only*

New in version 18.2.0: *cache_hash*

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

For example:

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

`attr.ib`(*default=NOTHING*, *validator=None*, *repr=True*, *cmp=True*, *hash=None*, *init=True*, *convert=None*, *metadata=None*, *type=None*, *converter=None*, *factory=None*, *kw_only=False*)

Create a new attribute on a class.

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

Parameters

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

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

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

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

- **factory** (*callable*) – Syntactic sugar for default=attr.Factory(callable).

- **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** (*bool*) – Include this attribute in the generated __repr__ method.

- **cmp** (*bool*) – Include this attribute in the generated comparison methods (__eq__ et al).

- **hash** (bool or None) – Include this attribute in the generated __hash__ method. If None (default), mirror *cmp*'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 converter 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).

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 *cmp* 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*

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

class attr.**Attribute**(*name*, *default*, *validator*, *repr*, *cmp*, *hash*, *init*, *convert=None*, *metadata=None*, *type=None*, *converter=None*, *kw_only=False*)

Read-only representation of an attribute.

Attribute name The name of the attribute.

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

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

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

- *fields()* returns a tuple of them.
- Validators get them passed as the first argument.

Warning: You should never instantiate this class yourself!

```
>>> import attr
>>> @attr.s
... class C(object):
...     x = attr.ib()
>>> attr.fields(C).x
Attribute(name='x', default=NOTHING, validator=None, repr=True, cmp=True,␣
↪hash=None, init=True, metadata=mappingproxy({}), type=None, converter=None, kw_
↪only=False)
```

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

Stores a factory callable.

If passed as the default value to *attr.ib()*, 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(object):
...     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})
```

exception attr.exceptions.**FrozenInstanceError**
A frozen/immutable instance has been attempted to be modified.

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

New in version 16.1.0.

exception attr.exceptions.**AttrsAttributeNotFoundError**
An `attrs` function couldn't find an attribute that the user asked for.

New in version 16.2.0.

exception attr.exceptions.**NotAnAttrsClassError**
A non-`attrs` class has been passed into an `attrs` function.

New in version 16.2.0.

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

New in version 17.3.0.

For example:

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

6.7.2 Helpers

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

attr.**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 *attr.Attribute*

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

For example:

```
>>> @attr.s
... class C(object):
...     x = attr.ib()
...     y = attr.ib()
>>> attr.fields(C)
(Attribute(name='x', default=NOTHING, validator=None, repr=True, cmp=True,␣
↪hash=None, init=True, metadata=mappingproxy({}), type=None, converter=None, kw_
↪only=False), Attribute(name='y', default=NOTHING, validator=None, repr=True,␣
↪cmp=True, hash=None, init=True, metadata=mappingproxy({}), type=None,␣
↪converter=None, kw_only=False))
>>> attr.fields(C)[1]
Attribute(name='y', default=NOTHING, validator=None, repr=True, cmp=True,␣
↪hash=None, init=True, metadata=mappingproxy({}), type=None, converter=None, kw_
↪only=False)
>>> attr.fields(C).y is attr.fields(C)[1]
True
```

attr.**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 *attr.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(object):
...     x = attr.ib()
...     y = attr.ib()
>>> attr.fields_dict(C)
{'x': Attribute(name='x', default=NOTHING, validator=None, repr=True, cmp=True,␣
↪hash=None, init=True, metadata=mappingproxy({}), type=None, converter=None, kw_
↪only=False), 'y': Attribute(name='y', default=NOTHING, validator=None,␣
↪repr=True, cmp=True, hash=None, init=True, metadata=mappingproxy({}), type=None,
↪ converter=None, kw_only=False)}
>>> attr.fields_dict(C)['y']
Attribute(name='y', default=NOTHING, validator=None, repr=True, cmp=True,␣
↪hash=None, init=True, metadata=mappingproxy({}), type=None, converter=None, kw_
↪only=False)
>>> attr.fields_dict(C)['y'] is attr.fields(C).y
True
```

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

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

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

Return type bool

For example:

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

attr.**asdict**(*inst*, *recurse=True*, *filter=None*, *dict_factory=<class 'dict'>*, *retain_collection_types=False*)
Return the attrs attribute values of *inst* as a dict.

Optionally recurse into other attrs-decorated classes.

Parameters

- **inst** – Instance of an attrs-decorated class.
- **recurse** (*bool*) – Recurse into classes that are also attrs-decorated.
- **filter** (*callable*) – A callable whose return code determines whether an attribute or element is included (True) or dropped (False). Is called with the *attr.Attribute* as the first argument and the value as the second argument.
- **dict_factory** (*callable*) – A callable to produce dictionaries from. For example, to produce ordered dictionaries instead of normal Python dictionaries, pass in collections.OrderedDict.
- **retain_collection_types** (*bool*) – Do not convert to list when encountering an attribute whose type is tuple or set. Only meaningful if recurse is True.

Return type return type of *dict_factory*

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

New in version 16.0.0: *dict_factory*

New in version 16.1.0: *retain_collection_types*

For example:

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

attr.**astuple**(*inst*, *recurse=True*, *filter=None*, *tuple_factory=<class 'tuple'>*, *retain_collection_types=False*)
Return the attrs attribute values of *inst* as a tuple.

Optionally recurse into other attrs-decorated classes.

Parameters

- **inst** – Instance of an `attrs`-decorated class.
- **recurse** (*bool*) – Recurse into classes that are also `attrs`-decorated.
- **filter** (*callable*) – A callable whose return code determines whether an attribute or element is included (`True`) or dropped (`False`). Is called with the *attr.Attribute* as the first argument and the value as the second argument.
- **tuple_factory** (*callable*) – A callable to produce tuples from. For example, to produce lists instead of tuples.
- **retain_collection_types** (*bool*) – Do not convert to `list` or `dict` when encountering an attribute which type is `tuple`, `dict` or `set`. Only meaningful if `recurse` is `True`.

Return type return type of *tuple_factory*

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

New in version 16.2.0.

For example:

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

`attrs` includes some handy helpers for filtering:

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

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

Return type `callable`

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

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

Return type `callable`

See *Converting to Collections Types* for examples.

`attr.`**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:

```
>>> @attr.s
... class C(object):
...     x = attr.ib()
...     y = attr.ib()
>>> i1 = C(1, 2)
>>> i1
C(x=1, y=2)
>>> i2 = attr.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.`**`validate`**`(`*`inst`*`)`
Validate all attributes on *inst* that have a validator.

Leaves all exceptions through.

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

For example:

```
>>> @attr.s
... class C(object):
...     x = attr.ib(validator=attr.validators.instance_of(int))
>>> i = C(1)
>>> i.x = "1"
>>> attr.validate(i)
Traceback (most recent call last):
   ...
TypeError: ("'x' must be <type 'int'> (got '1' that is a <type 'str'>).",␣
↪Attribute(name='x', default=NOTHING, validator=<instance_of validator for type
↪<type 'int'>>, repr=True, cmp=True, hash=None, init=True, type=None, kw_
↪only=False), <type 'int'>, '1')
```

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

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

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

6.7.3 Validators

`attrs` comes with some common validators in the `attrs.validators` module:

attr.validators.**instance_of**(*type*)

A validator that raises a `TypeError` if the initializer is called with a wrong type for this particular attribute (checks are performed using `isinstance()` therefore it's also valid to pass a tuple of types).

Parameters **type** (*type or tuple of types*) – The type to check for.

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

For example:

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

attr.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 *attr.Attribute*), the expected options, and the value it got.

New in version 17.1.0.

For example:

```
>>> import enum
>>> class State(enum.Enum):
...     ON = "on"
...     OFF = "off"
>>> @attr.s
... class C(object):
...     state = attr.ib(validator=attr.validators.in_(State))
...     val = attr.ib(validator=attr.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')
>>> C(State.ON, 4)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
   ...
ValueError: 'val' must be in [1, 2, 3] (got 4)
```

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

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

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

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

attr.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 *attr.ib()*'s validator argument.

Thus the following two statements are equivalent:

```
x = attr.ib(validator=attr.validators.and_(v1, v2, v3))
x = attr.ib(validator=[v1, v2, v3])
```

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

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

Parameters **validator** (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:

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

6.7.4 Converters

attr.converters.**optional**(*converter*)
A converter that allows an attribute to be optional. An optional attribute is one which can be set to None.

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

New in version 17.1.0.

For example:

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

attr.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 *attr.Factory* is supported, however the takes_self option is *not*.
- **factory** (*callable*) – A callable that takes not 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 *attr.Factory* is passed with takes_self=True.

New in version 18.2.0.

For example:

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

6.7.5 Deprecated APIs

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

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 *evolve()* instead.

6.8 Extending

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

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

```
>>> import attr
>>> def print_attrs(cls):
...     print(cls.__attrs_attrs__)
>>> @print_attrs
... @attr.s
... class C(object):
...     a = attr.ib()
(Attribute(name='a', default=NOTHING, validator=None, repr=True, cmp=True, hash=None,␣
↪init=True, metadata=mappingproxy({}), type=None, converter=None, kw_only=False),)
```

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

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

is just syntactic sugar for:

```
def original_f():
   pass

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

6.8.1 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.

6.8.2 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:

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

6.8.3 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

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

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

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

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

6.9 How Does It Work?

6.9.1 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 `@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.

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

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

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

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

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

6.9.2 Immutability

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

Depending on whether a class is a dict class or a slots 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.

Slots Classes

Slots 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)"
.....................................
Median +- std dev: 378 ns +- 12 ns

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

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

Summary

You should avoid to instantiate lots of frozen slotted classes (i.e. `@attr.s(slots=True, frozen=True)`) 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).

6.10 Glossary

dict classes A regular class whose attributes are stored in the `__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`.

slotted classes A class that has no `__dict__` attribute and defines its attributes in a `__slots__` attribute instead. In `attrs`, they are created by passing `slots=True` to `@attr.s`.

Their main advantage is that they use less memory on CPython[1].

However they also come with a bunch of 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__`:

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

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

- 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 slot classes.
- Using `pickle` with slotted classes requires pickle protocol 2 or greater. Python 2 uses protocol 0 by default so the protocol needs to be specified. Python 3 uses protocol 3 by default. You can support protocol 0 and 1 by implementing `__getstate__` and `__setstate__` methods yourself. Those methods are created for frozen slotted classes because they won't pickle otherwise. Think twice before using `pickle` though.
- 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 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 `__init_subclass__` do not work with slotted classes.

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

CHAPTER 7

Indices and tables

- genindex
- search

Index

A

and_() (in module attr.validators), 62
asdict() (in module attr), 58
assoc() (in module attr), 63
astuple() (in module attr), 58
Attribute (class in attr), 54
AttrsAttributeNotFoundError, 56

D

default_if_none() (in module attr.converters), 63
DefaultAlreadySetError, 56
dict classes, **67**

E

evolve() (in module attr), 59
exclude() (in module attr.filters), 59

F

Factory (class in attr), 55
fields() (in module attr), 56
fields_dict() (in module attr), 57
FrozenInstanceError, 56

G

get_run_validators() (in module attr), 60

H

has() (in module attr), 57

I

ib() (in module attr), 52
in_() (in module attr.validators), 61
include() (in module attr.filters), 59
instance_of() (in module attr.validators), 60

M

make_class() (in module attr), 55

N

NotAnAttrsClassError, 56

O

optional() (in module attr.converters), 63
optional() (in module attr.validators), 62

P

provides() (in module attr.validators), 62
Python Enhancement Proposals
    PEP 526, 42
    PEP 557, 29

S

s() (in module attr), 50
set_run_validators() (in module attr), 60
slotted classes, **67**

U

UnannotatedAttributeError, 56

V

validate() (in module attr), 60