cattrs Documentation

Release 24.1.2

Tin Tvrtković

Sep 22, 2024

INTRODUCTION

Because validation belongs to the edges.

---

cattrs is a Swiss Army knife for (un)structuring and validating data in Python. In practice, that means it converts **unstructured dictionaries** into **proper classes** and back, while **validating** their contents.

cattrs works best with *attrs* classes, and dataclasses where simple (un-)structuring works out of the box, even for nested data, without polluting your data model with serialization details:

```
>>> from attrs import define
>>> from cattrs import structure, unstructure
>>> @define
... class C:
...     a: int
...     b: list[str]
>>> instance = structure({'a': 1, 'b': ['x', 'y']}, C)
>>> instance
C(a=1, b=['x', 'y'])
>>> unstructure(instance)
{'a': 1, 'b': ['x', 'y']}
```

However, *cattrs* does **much** more with a focus on **functional composition** and **not coupling** your data model to its serialization and validation rules.

To learn more on why to use *cattrs*, have a look at *Why cattrs?*, and if you're convinced jump right into *The Basics*!

CHAPTER

ONE

WHY *CATTRS*?

Python has a rich set of powerful, easy to use, built-in **unstructured** data types like dictionaries, lists and tuples. These data types effortlessly convert into common serialization formats like JSON, MessagePack, CBOR, YAML or TOML.

But the data that is used by your **business logic** should be **structured** into well-defined classes, since not all combinations of field names or values are valid inputs to your programs. The more trust you can have into the structure of your data, the simpler your code can be, and the fewer edge cases you have to worry about.

When you're handed unstructured data (by your network, file system, database, …), *cattrs* helps to convert this data into trustworthy structured data. When you have to convert your structured data into data types that other libraries can handle, *cattrs* turns your classes and enumerations into dictionaries, integers and strings.

attrs (and to a certain degree dataclasses) are excellent libraries for declaratively describing the structure of your data, but they're purposefully not serialization libraries. *cattrs* is there for you the moment your `attrs.asdict(your_instance)` and `YourClass(**data)` start failing you because you need more control over the conversion process.

1.1 Examples

cattrs works best with *attrs* classes, and dataclasses where simple (un-)structuring works out of the box, even for nested data, without polluting your data model with serialization details:

```
>>> from attrs import define
>>> from cattrs import structure, unstructure
>>> @define
... class C:
...     a: int
...     b: list[str]
>>> instance = structure({'a': 1, 'b': ['x', 'y']}, C)
>>> instance
C(a=1, b=['x', 'y'])
>>> unstructure(instance)
{'a': 1, 'b': ['x', 'y']}
```

Important: Note how the structuring and unstructuring details do **not** pollute your class, meaning: your data model. Any needs to configure the conversion are done within *cattrs* itself, not within your data model.

There are popular validation libraries for Python that couple your data model with its validation and serialization rules based on, for example, web APIs. We think that's the wrong approach. Validation and serializations are concerns of the edges of your program – not the core. They should neither apply design pressure on your business code, nor affect the performance of your code through unnecessary validation. In bigger real-world code bases it's also common for data coming from multiple sources that need different validation and serialization rules.

 You gotta keep 'em separated.

cattrs also works with the usual Python collection types like dictionaries, lists, or tuples when you want to **normalize** unstructured data data into a certain (still unstructured) shape. For example, to convert a list of a float, an int and a string into a tuple of ints:

```
>>> import cattrs

>>> cattrs.structure([1.0, 2, "3"], tuple[int, int, int])
(1, 2, 3)
```

Finally, here's a much more complex example, involving *attrs* classes where *cattrs* interprets the type annotations to structure and unstructure the data correctly, including Enums and nested data structures:

```
>>> from enum import unique, Enum
>>> from typing import Sequence
>>> from cattrs import structure, unstructure
>>> from attrs import define, field

>>> @unique
... class CatBreed(Enum):
...     SIAMESE = "siamese"
...     MAINE_COON = "maine_coon"
...     SACRED_BIRMAN = "birman"

>>> @define
... class Cat:
...     breed: CatBreed
...     names: Sequence[str]

>>> @define
... class DogMicrochip:
...     chip_id = field()  # Type annotations are optional, but recommended
...     time_chipped: float = field()

>>> @define
... class Dog:
...     cuteness: int
...     chip: DogMicrochip | None = None

>>> p = unstructure([Dog(cuteness=1, chip=DogMicrochip(chip_id=1, time_chipped=10.0)),
...                  Cat(breed=CatBreed.MAINE_COON, names=('Fluffly', 'Fluffer'))])

>>> p
[{'cuteness': 1, 'chip': {'chip_id': 1, 'time_chipped': 10.0}}, {'breed': 'maine_coon
↪', 'names': ['Fluffly', 'Fluffer']}]
>>> structure(p, list[Dog | Cat])
[Dog(cuteness=1, chip=DogMicrochip(chip_id=1, time_chipped=10.0)), Cat(breed=
↪<CatBreed.MAINE_COON: 'maine_coon'>, names=['Fluffly', 'Fluffer'])]
```

Tip: Consider unstructured data a low-level representation that needs to be converted to structured data to be handled, and use `structure()`. When you're done, `unstructure()` the data to its unstructured form and pass it along to another library or module.

1.2 Features

1.2.1 Recursive Unstructuring

- *attrs* classes and dataclasses are converted into dictionaries in a way similar to `attrs.asdict()`, or into tuples in a way similar to `attrs.astuple()`.
- Enumeration instances are converted to their values.
- Other types are let through without conversion. This includes types such as integers, dictionaries, lists and instances of non-*attrs* classes.
- Custom converters for any type can be registered using `register_unstructure_hook`.

1.2.2 Recursive Structuring

Converts unstructured data into structured data, recursively, according to your specification given as a type. The following types are supported:

- `typing.Optional[T]` and its 3.10+ form, `T | None`.
- `list[T]`, `typing.List[T]`, `typing.MutableSequence[T]`, `typing.Sequence[T]` convert to a lists.
- `tuple` and `typing.Tuple` (both variants, `tuple[T, ...]` and `tuple[X, Y, Z]`).
- `set[T]`, `typing.MutableSet[T]`, and `typing.Set[T]` convert to a sets.
- `frozenset[T]`, and `typing.FrozenSet[T]` convert to a frozensets.
- `dict[K, V]`, `typing.Dict[K, V]`, `typing.MutableMapping[K, V]`, and `typing.Mapping[K, V]` convert to a dictionaries.
- `typing.TypedDict`, ordinary and generic.
- `typing.NewType`
- PEP 695 type aliases on 3.12+
- *attrs* classes with simple attributes and the usual `__init__`[1].
- All *attrs* classes and dataclasses with the usual `__init__`, if their complex attributes have type metadata.
- Unions of supported *attrs* classes, given that all of the classes have a unique field.
- Unions of anything, if you provide a disambiguation function for it.
- Custom converters for any type can be registered using `register_structure_hook`.

[1] Simple attributes are attributes that can be assigned unstructured data, like numbers, strings, and collections of unstructured data.

1.2.3 Batteries Included

cattrs comes with pre-configured converters for a number of serialization libraries, including JSON (standard library, *orjson*, UltraJSON), *msgpack*, *cbor2*, *bson*, PyYAML, *tomlkit* and *msgspec* (supports only JSON at this time).

For details, see the cattrs.preconf package.

1.3 Design Decisions

cattrs is based on a few fundamental design decisions:

- Un/structuring rules are separate from the models. This allows models to have a one-to-many relationship with un/structuring rules, and to create un/structuring rules for models which you do not own and you cannot change. (*cattrs* can be configured to use un/structuring rules from models using the `use_class_methods` strategy.)
- Invent as little as possible; reuse existing ordinary Python instead. For example, *cattrs* did not have a custom exception type to group exceptions until the sanctioned Python `exceptiongroups`. A side-effect of this design decision is that, in a lot of cases, when you're solving *cattrs* problems you're actually learning Python instead of learning *cattrs*.
- Resist the temptation to guess. If there are two ways of solving a problem, *cattrs* should refuse to guess and let the user configure it themselves.

A foolish consistency is the hobgoblin of little minds, so these decisions can and are sometimes broken, but they have proven to be a good foundation.

1.4 Additional Documentation and Talks

- On structured and unstructured data, or the case for cattrs
- Why I use attrs instead of pydantic
- cattrs I: un/structuring speed
- Python has a macro language - it's Python (PyCon IT 2022)
- Intro to cattrs 23.1

CHAPTER

TWO

THE BASICS

All *cattrs* functionality is exposed through a `cattrs.Converter` object. A global converter is provided for convenience as `cattrs.global_converter` but more complex customizations should be performed on private instances, any number of which can be made.

2.1 Converters and Hooks

The core functionality of a converter is structuring and unstructuring data by composing *provided* and *custom handling functions*, called *hooks*.

To create a private converter, instantiate a `cattrs.Converter`. Converters are relatively cheap; users are encouraged to have as many as they need.

The two main methods, `structure` and `unstructure`, are used to convert between *structured* and *unstructured* data.

```
>>> from cattrs import structure, unstructure
>>> from attrs import define

>>> @define
... class Model:
...     a: int

>>> unstructure(Model(1))
{'a': 1}
>>> structure({"a": 1}, Model)
Model(a=1)
```

cattrs comes with a rich library of un/structuring hooks by default but it excels at composing custom hooks with built-in ones.

The simplest approach to customization is writing a new hook from scratch. For example, we can write our own hook for the `int` class and register it to a converter.

```
>>> from cattrs import Converter

>>> converter = Converter()

>>> @converter.register_structure_hook
... def int_hook(value, type) -> int:
...     if not isinstance(value, int):
...         raise ValueError('not an int!')
...     return value
```

Now, any other hook converting an `int` will use it.

Another approach to customization is wrapping (composing) an existing hook with your own function. A base hook can be obtained from a converter and then be subjected to the very rich machinery of function composition that Python offers.

```
>>> base_hook = converter.get_structure_hook(Model)

>>> @converter.register_structure_hook
... def my_model_hook(value, type) -> Model:
...     # Apply any preprocessing to the value.
...     result = base_hook(value, type)
...     # Apply any postprocessing to the model.
...     return result
```

(`cattrs.structure({}, Model)` is equivalent to `cattrs.get_structure_hook(Model)({}, Model)`.)

Now if we use this hook to structure a `Model`, through ✨the magic of function composition✨ that hook will use our old `int_hook`.

```
>>> converter.structure({"a": "1"}, Model)
  + Exception Group Traceback (most recent call last):
    |   File "...", line 22, in <module>
    |     base_hook({"a": "1"}, Model)
    |   File "<cattrs generated structure __main__.Model>", line 9, in structure_Model
    | cattrs.errors.ClassValidationError: While structuring Model (1 sub-exception)
    +-+---------------- 1 ----------------
      | Traceback (most recent call last):
      |   File "<cattrs generated structure __main__.Model>", line 5, in structure_
↪Model
      |   File "...", line 15, in my_int_hook
      |     raise ValueError("not an int!")
      | ValueError: not an int!
      | Structuring class Model @ attribute a
      +------------------------------------
```

To continue reading about customizing *cattrs*, see *Customizing (Un-)structuring*. More advanced structuring customizations are commonly called *Strategies*.

2.2 Global Converter

Global *cattrs* functions, such as *`cattrs.structure()`*, use a single *`global converter`*. Changes done to this global converter, such as registering new structure and unstructure hooks, affect all code using the global functions.

The following functions implicitly use this global converter:

- *`cattrs.structure()`*
- *`cattrs.unstructure()`*
- *`cattrs.get_structure_hook()`*
- *`cattrs.get_unstructure_hook()`*
- *`cattrs.structure_attrs_fromtuple()`*
- *`cattrs.structure_attrs_fromdict()`*

Changes made to the global converter will affect the behavior of these functions.

Larger applications are strongly encouraged to create and customize different, private instances of *cattrs.Converter*.

CHAPTER

THREE

BUILT-IN HOOKS

cattrs converters come with with a large repertoire of un/structuring hooks built-in. As always, complex hooks compose with simpler ones.

3.1 Primitive Values

3.1.1 `int`, `float`, `str`, `bytes`

When structuring, use any of these types to coerce the object to that type.

```
>>> cattrs.structure(1, str)
'1'
>>> cattrs.structure("1", float)
1.0
```

In case the conversion isn't possible the expected exceptions will be propagated out. The particular exceptions are the same as if you'd tried to do the coercion directly.

```
>>> cattrs.structure("not-an-int", int)
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'not-an-int'
```

Coercion is performed for performance and compatibility reasons. Any of these hooks can be overriden if pure validation is required instead.

```
>>> c = Converter()

>>> @c.register_structure_hook
... def validate(value, type) -> int:
...     if not isinstance(value, type):
...         raise ValueError(f'{value!r} not an instance of {type}')

>>> c.structure("1", int)
Traceback (most recent call last):
...
ValueError: '1' not an instance of <class 'int'>
```

When unstructuring, these types are passed through unchanged.

3.1.2 Enums

Enums are structured by their values, and unstructured to their values. This works even for complex values, like tuples.

```
>>> @unique
... class CatBreed(Enum):
...     SIAMESE = "siamese"
...     MAINE_COON = "maine_coon"
...     SACRED_BIRMAN = "birman"

>>> cattrs.structure("siamese", CatBreed)
<CatBreed.SIAMESE: 'siamese'>

>>> cattrs.unstructure(CatBreed.SIAMESE)
'siamese'
```

Again, in case of errors, the expected exceptions are raised.

3.1.3 `pathlib.Path`

`pathlib.Path` objects are structured using their string value, and unstructured into their string value.

```
>>> from pathlib import Path

>>> cattrs.structure("/root", Path)
PosixPath('/root')

>>> cattrs.unstructure(Path("/root"))
'/root'
```

In case the conversion isn't possible, the resulting exception is propagated out.

Added in version 23.1.0.

3.2 Collections and Related Generics

3.2.1 Optionals

`Optional` primitives and collections are supported out of the box. PEP 604 optionals (`T | None`) are also supported on Python 3.10 and later.

```
>>> cattrs.structure(None, int)
Traceback (most recent call last):
...
TypeError: int() argument must be a string, a bytes-like object or a number, not
↪'NoneType'

>>> print(cattrs.structure(None, int | None))
None
```

Bare `Optional` s (non-parameterized, just `Optional`, as opposed to `Optional[str]`) aren't supported; `Optional[Any]` should be used instead.

`Optionals` handling can be customized using *register_structure_hook()* and *register_unstructure_hook()*.

```
>>> converter = Converter()

>>> @converter.register_structure_hook
... def hook(val: Any, type: Any) -> str | None:
...     if val in ("", None):
...         return None
...     return str(val)
...

>>> print(converter.structure("", str | None))
None
```

3.2.2 Lists

Lists can be structured from any iterable object. Types converting to lists are:

- `typing.Sequence[T]`
- `typing.MutableSequence[T]`
- `typing.List[T]`
- `list[T]`

In all cases, a new list will be returned, so this operation can be used to copy an iterable into a list. A bare type, for example `Sequence` instead of `Sequence[int]`, is equivalent to `Sequence[Any]`.

```
>>> cattrs.structure((1, 2, 3), MutableSequence[int])
[1, 2, 3]
```

When unstructuring, lists are copied and their contents are handled according to their inner type. A useful use case for unstructuring collections is to create a deep copy of a complex or recursive collection.

3.2.3 Dictionaries

Dictionaries can be produced from other mapping objects. More precisely, the unstructured object must expose an `items()` method producing an iterable of key-value tuples, and be able to be passed to the `dict` constructor as an argument. Types converting to dictionaries are:

- `dict[K, V]` and `typing.Dict[K, V]`
- `collections.abc.MutableMapping[K, V]` and `typing.MutableMapping[K, V]`
- `collections.abc.Mapping[K, V]` and `typing.Mapping[K, V]`

In all cases, a new dict will be returned, so this operation can be used to copy a mapping into a dict. Any type parameters set to `typing.Any` will be passed through unconverted. If both type parameters are absent, they will be treated as `Any` too.

```
>>> from collections import OrderedDict
>>> cattrs.structure(OrderedDict([(1, 2), (3, 4)]), dict)
{1: 2, 3: 4}
```

Both keys and values are converted.

```
>>> cattrs.structure({1: None, 2: 2.0}, dict[str, Optional[int]])
{'1': None, '2': 2}
```

3.2.4 Virtual Subclasses of `abc.Mapping` and `abc.MutableMapping`

If a class declares itself a virtual subclass of `collections.abc.Mapping` or `collections.abc.MutableMapping` and its initializer accepts a dictionary, *cattrs* will be able to structure it by default.

3.2.5 Homogeneous and Heterogeneous Tuples

Homogeneous and heterogeneous tuples can be structured from iterable objects. Heterogeneous tuples require an iterable with the number of elements matching the number of type parameters exactly.

Use:

- `Tuple[A, B, C, D]`
- `tuple[A, B, C, D]`

Homogeneous tuples use:

- `Tuple[T, ...]`
- `tuple[T, ...]`

In all cases a tuple will be produced. Any type parameters set to `typing.Any` will be passed through unconverted.

```
>>> cattrs.structure([1, 2, 3], tuple[int, str, float])
(1, '2', 3.0)
```

When unstructuring, heterogeneous tuples unstructure into tuples since it's faster and virtually all serialization libraries support tuples natively.

See also:

Support for typing.NamedTuple.

Note: Structuring heterogenous tuples are not supported by the BaseConverter.

3.2.6 Deques

Deques can be structured from any iterable object. Types converting to deques are:

- `typing.Deque[T]`
- `collections.deque[T]`

In all cases, a new **unbounded** deque (`maxlen=None`) will be produced, so this operation can be used to copy an iterable into a deque. If you want to convert into bounded `deque`, registering a custom structuring hook is a good approach.

```
>>> from collections import deque
>>> cattrs.structure((1, 2, 3), deque[int])
deque([1, 2, 3])
```

Deques are unstructured into lists, or into deques when using the *`BaseConverter`*.

Added in version 23.1.0.

3.2.7 Sets and Frozensets

Sets and frozensets can be structured from any iterable object. Types converting to sets are:

- `typing.Set[T]`
- `typing.MutableSet[T]`
- `set[T]`

Types converting to frozensets are:

- `typing.FrozenSet[T]`
- `frozenset[T]`

In all cases, a new set or frozenset will be returned. A bare type, for example `MutableSet` instead of `MutableSet[int]`, is equivalent to `MutableSet[Any]`.

```
>>> cattrs.structure([1, 2, 3, 4], set)
{1, 2, 3, 4}
```

Sets and frozensets are unstructured into the same class.

3.2.8 Typed Dicts

TypedDicts can be structured from mapping objects, usually dictionaries.

```
>>> from typing import TypedDict

>>> class MyTypedDict(TypedDict):
...     a: int

>>> cattrs.structure({"a": "1"}, MyTypedDict)
{'a': 1}
```

Both *total* and *non-total* TypedDicts are supported, and inheritance between any combination works (except on 3.8 when `typing.TypedDict` is used, see below). Generic TypedDicts work on Python 3.11 and later, since that is the first Python version that supports them in general.

`typing.Required` and `typing.NotRequired` are supported.

On Python 3.8, using `typing_extensions.TypedDict` is recommended since `typing.TypedDict` doesn't support all necessary features so certain combinations of subclassing, totality and `typing.Required` won't work.

Similar to attrs *classes*, un/structuring can be customized using *`cattrs.gen.typeddicts.make_dict_structure_fn()`* and *`cattrs.gen.typeddicts.make_dict_unstructure_fn()`*.

```
>>> from typing import TypedDict
>>> from cattrs import Converter
>>> from cattrs.gen import override
>>> from cattrs.gen.typeddicts import make_dict_structure_fn

>>> class MyTypedDict(TypedDict):
...     a: int
...     b: int

>>> c = Converter()
>>> c.register_structure_hook(
```

(continues on next page)

(continued from previous page)

```
...     MyTypedDict,
...     make_dict_structure_fn(
...         MyTypedDict,
...         c,
...         a=override(rename="a-with-dash")
...     )
... )

>>> c.structure({"a-with-dash": 1, "b": 2}, MyTypedDict)
{'b': 2, 'a': 1}
```

TypedDicts unstructure into dictionaries, potentially unchanged (depending on the exact field types and registered hooks).

```
>>> from typing import TypedDict
>>> from datetime import datetime, timezone
>>> from cattrs import Converter

>>> class MyTypedDict(TypedDict):
...     a: datetime

>>> c = Converter()
>>> c.register_unstructure_hook(datetime, lambda d: d.timestamp())

>>> c.unstructure({"a": datetime(1970, 1, 1, tzinfo=timezone.utc)}, unstructure_
↪as=MyTypedDict)
{'a': 0.0}
```

Added in version 23.1.0.

3.3 *attrs* Classes and Dataclasses

attrs classes and dataclasses work out of the box. The fields require type annotations (even if static type-checking is not being used), or they will be treated as *typing.Any*.

When structuring, given a mapping `d` and class `A`, *cattrs* will instantiate `A` with `d` unpacked.

```
>>> @define
... class A:
...     a: int
...     b: int

>>> cattrs.structure({'a': 1, 'b': '2'}, A)
A(a=1, b=2)
```

Tuples can be structured into classes using *`structure_attrs_fromtuple()`* (`fromtuple` as in the opposite of `attrs.astuple` and *`BaseConverter.unstructure_attrs_astuple()`*).

```
>>> @define
... class A:
...     a: str
...     b: int

>>> cattrs.structure_attrs_fromtuple(['string', '2'], A)
A(a='string', b=2)
```

Loading from tuples can be made the default by creating a new *Converter* with `unstruct_strat=cattr.UnstructureStrategy.AS_TUPLE`.

```
>>> converter = cattrs.Converter(unstruct_strat=cattr.UnstructureStrategy.AS_TUPLE)
>>> @define
... class A:
...     a: str
...     b: int

>>> converter.structure(['string', '2'], A)
A(a='string', b=2)
```

Structuring from tuples can also be made the default for specific classes only by registering a hook the usual way.

```
>>> converter = cattrs.Converter()

>>> @define
... class A:
...     a: str
...     b: int

>>> converter.register_structure_hook(A, converter.structure_attrs_fromtuple)
```

3.3.1 Generics

Generic *attrs* classes and dataclasses are fully supported, both using `typing.Generic` and PEP 695.

```
>>> @define
... class A[T]:
...    a: T

>>> cattrs.structure({"a": "1"}, A[int])
A(a=1)
```

3.3.2 Using Attribute Types and Converters

By default, *structure()* will use hooks registered using *register_structure_hook()* to convert values to the attribute type, and proceed to invoking any converters registered on attributes with `field`.

```
>>> from ipaddress import IPv4Address, ip_address
>>> converter = cattrs.Converter()

# Note: register_structure_hook has not been called, so this will fallback to 'ip_
↪address'
>>> @define
... class A:
...     a: IPv4Address = field(converter=ip_address)

>>> converter.structure({'a': '127.0.0.1'}, A)
A(a=IPv4Address('127.0.0.1'))
```

Priority is still given to hooks registered with *register_structure_hook()*, but this priority can be inverted by setting `prefer_attrib_converters` to `True`.

```
>>> converter = cattrs.Converter(prefer_attrib_converters=True)

>>> @define
... class A:
...     a: int = field(converter=lambda v: int(v) + 5)

>>> converter.structure({'a': '10'}, A)
A(a=15)
```

See also:

If an *attrs* or dataclass class uses inheritance and as such has one or several subclasses, it can be structured automatically to its exact subtype by using the *include subclasses* strategy.

3.4 Unions

Unions of `NoneType` and a single other type (also known as optionals) are supported by a *special case*.

3.4.1 Automatic Disambiguation

cattrs includes an opinionated strategy for automatically handling unions of *attrs* classes; see *Default Union Strategy* for details.

When unstructuring these kinds of unions, each union member will be unstructured according to the hook for that type.

3.4.2 Unions of Simple Types

cattrs comes with the *Union Passthrough*, which enables converters to structure unions of many primitive types and literals. This strategy can be applied to any converter, and is pre-applied to all *preconf* converters according to their underlying protocols.

3.5 Special Typing Forms

3.5.1 `typing.Any`

When structuring, use `typing.Any` to avoid applying any conversions to the object you're structuring; it will simply be passed through.

```
>>> cattrs.structure(1, Any)
1
>>> d = {1: 1}
>>> cattrs.structure(d, Any) is d
True
```

When unstructuring, `typing.Any` will make the value be unstructured according to its runtime class.

Changed in version 24.1.0: Previously, the unstructuring rules for `Any` were underspecified, leading to inconsistent behavior.

Changed in version 24.1.0: `typing_extensions.Any` is now also supported.

3.5.2 typing.Literal

When structuring, PEP 586 literals are validated to be in the allowed set of values.

```
>>> from typing import Literal

>>> cattrs.structure(1, Literal[1, 2])
1
```

When unstructuring, literals are passed through.

Added in version 1.7.0.

3.5.3 typing.NamedTuple

Named tuples with type hints (created from `typing.NamedTuple`) are supported. Named tuples are un/structured using tuples or lists by default.

The *`cattrs.cols`* module contains hook factories for un/structuring named tuples using dictionaries instead, *see here for details*.

Added in version 24.1.0.

3.5.4 typing.Final

PEP 591 Final attribute types (`Final[int]`) are supported and handled according to the inner type (in this case, `int`).

Added in version 23.1.0.

3.5.5 typing.Annotated

PEP 593 annotations (`typing.Annotated[type, ...]`) are supported and are handled using the first type present in the annotated type.

Added in version 1.4.0.

3.5.6 Type Aliases

Type aliases are supported on Python 3.12+ and are handled according to the rules for their underlying type. Their hooks can also be overriden using *Predicate Hooks*.

> **Warning:** Type aliases using `typing.TypeAlias` aren't supported since there is no way at runtime to distinguish them from their underlying types.

```
>>> from datetime import datetime, UTC

>>> type IsoDate = datetime

>>> converter = cattrs.Converter()
>>> converter.register_structure_hook_func(
...     lambda t: t is IsoDate, lambda v, _: datetime.fromisoformat(v)
```

(continues on next page)

(continued from previous page)

```
... )
>>> converter.register_unstructure_hook_func(
...     lambda t: t is IsoDate, lambda v: v.isoformat()
... )

>>> converter.structure("2022-01-01", IsoDate)
datetime.datetime(2022, 1, 1, 0, 0)
>>> converter.unstructure(datetime.now(UTC), unstructure_as=IsoDate)
'2023-11-20T23:10:46.728394+00:00'
```

Added in version 24.1.0.

3.5.7 typing.NewType

NewTypes are supported and are handled according to the rules for their underlying type. Their hooks can also be overriden using *Converter.register_structure_hook()*.

```
>>> from typing import NewType
>>> from datetime import datetime

>>> IsoDate = NewType("IsoDate", datetime)

>>> converter = cattrs.Converter()
>>> converter.register_structure_hook(IsoDate, lambda v, _: datetime.fromisoformat(v))

>>> converter.structure("2022-01-01", IsoDate)
datetime.datetime(2022, 1, 1, 0, 0)
```

Added in version 22.2.0.

3.5.8 typing.Protocol

Protocols cannot be structured by default and so require custom hooks.

Protocols are unstructured according to the actual runtime type of the value.

Added in version 1.9.0.

CHAPTER

FOUR

CUSTOMIZING (UN-)STRUCTURING

This section describes customizing the unstructuring and structuring processes in *cattrs*.

4.1 Custom (Un-)structuring Hooks

You can write your own structuring and unstructuring functions and register them for types using *Converter.register_structure_hook()* and *Converter.register_unstructure_hook()*. This approach is the most flexible but also requires the most amount of boilerplate.

register_structure_hook() and *register_unstructure_hook()* use a Python *singledispatch* under the hood. *singledispatch* is powerful and fast but comes with some limitations; namely that it performs checks using `issubclass()` which doesn't work with many Python types. Some examples of this are:

- various generic collections (`list[int]` is not a *subclass* of `list`)
- literals (`Literal[1]` is not a *subclass* of `Literal[1]`)
- generics (`MyClass[int]` is not a *subclass* of `MyClass`)
- protocols, unless they are `runtime_checkable`
- various modifiers, such as `Final` and `NotRequired`
- newtypes and 3.12 type aliases
- `typing.Annotated`

… and many others. In these cases, predicate functions should be used instead.

4.1.1 Use as Decorators

register_structure_hook() and *register_unstructure_hook()* can also be used as *decorators*. When used this way they behave a little differently.

register_structure_hook() will inspect the return type of the hook and register the hook for that type.

```
@converter.register_structure_hook
def my_int_hook(val: Any, _) -> int:
    """This hook will be registered for `int`s."""
    return int(val)
```

register_unstructure_hook() will inspect the type of the first argument and register the hook for that type.

```
from datetime import datetime

@converter.register_unstructure_hook
def my_datetime_hook(val: datetime) -> str:
    """This hook will be registered for `datetime`s."""
    return val.isoformat()
```

The non-decorator approach is still recommended when dealing with lambdas, hooks produced elsewhere, unannotated hooks and situations where type introspection doesn't work.

Added in version 24.1.0.

4.1.2 Predicate Hooks

A *predicate* is a function that takes a type and returns true or false depending on whether the associated hook can handle the given type.

The *register_unstructure_hook_func()* and *register_structure_hook_func()* are used to link un/structuring hooks to arbitrary types. These hooks are then called *predicate hooks*, and are very powerful.

Predicate hooks are evaluated after the *singledispatch* hooks. In the case where both a *singledispatch* hook and a predicate hook are present, the *singledispatch* hook will be used. Predicate hooks are checked in reverse order of registration, one-by-one, until a match is found.

The following example demonstrates a predicate that checks for the presence of an attribute on a class (`custom`), and then overrides the structuring logic.

```
>>> class D:
...     custom = True
...     def __init__(self, a):
...         self.a = a
...     def __repr__(self):
...         return f'D(a={self.a})'
...     @classmethod
...     def deserialize(cls, data):
...         return cls(data["a"])

>>> cattrs.register_structure_hook_func(
...     lambda cls: getattr(cls, "custom", False), lambda d, t: t.deserialize(d)
... )

>>> cattrs.structure({'a': 2}, D)
D(a=2)
```

4.1.3 Hook Factories

Hook factories are higher-order predicate hooks: they are functions that *produce* hooks. Hook factories are commonly used to create very optimized hooks by offloading part of the work into a separate, earlier step.

Hook factories are registered using *Converter.register_unstructure_hook_factory()* and *Converter.register_structure_hook_factory()*.

Here's an example showing how to use hook factories to apply the `forbid_extra_keys` to all attrs classes:

```
>>> from attrs import define, has
>>> from cattrs import Converter
>>> from cattrs.gen import make_dict_structure_fn

>>> c = Converter()

>>> c.register_structure_hook_factory(
...     has,
...     lambda cl: make_dict_structure_fn(cl, c, _cattrs_forbid_extra_keys=True)
... )

>>> @define
... class E:
...    an_int: int

>>> c.structure({"an_int": 1, "else": 2}, E)
Traceback (most recent call last):
...
cattrs.errors.ForbiddenExtraKeysError: Extra fields in constructor for E: else
```

Hook factories can receive the current converter by exposing an additional required parameter.

A complex use case for hook factories is described over at *Using Factory Hooks*.

Use as Decorators

register_unstructure_hook_factory() and *register_structure_hook_factory()* can also be used as decorators.

Here's an example of using an unstructure hook factory to handle unstructuring queues.

```
>>> from queue import Queue
>>> from typing import get_origin
>>> from cattrs import Converter

>>> c = Converter()

>>> @c.register_unstructure_hook_factory(lambda t: get_origin(t) is Queue)
... def queue_hook_factory(cl: Any, converter: Converter) -> Callable:
...     type_arg = get_args(cl)[0]
...     elem_handler = converter.get_unstructure_hook(type_arg)
...
...     def unstructure_hook(v: Queue) -> list:
...         res = []
...         while not v.empty():
...             res.append(elem_handler(v.get_nowait()))
...         return res
...
...     return unstructure_hook

>>> q = Queue()
>>> q.put(1)
>>> q.put(2)

>>> c.unstructure(q, unstructure_as=Queue[int])
[1, 2]
```

4.2 Customizing Collections

The *cattrs.cols* module contains predicates and hook factories useful for customizing collection handling. These hook factories can be wrapped to apply complex customizations.

Available predicates are:

- *is_any_set*
- *is_frozenset*
- *is_set*
- *is_sequence*
- *is_mapping*
- *is_namedtuple*

Tip: These predicates aren't *cattrs*-specific and may be useful in other contexts.

```
>>> from cattrs.cols import is_sequence

>>> is_sequence(list[str])
True
```

Available hook factories are:

- *iterable_unstructure_factory*
- *list_structure_factory*
- *namedtuple_structure_factory*
- *namedtuple_unstructure_factory*
- *namedtuple_dict_structure_factory*
- *namedtuple_dict_unstructure_factory*
- *mapping_structure_factory*

Additional predicates and hook factories will be added as requested.

For example, by default sequences are structured from any iterable into lists. This may be too lax, and additional validation may be applied by wrapping the default list structuring hook factory.

```
from cattrs.cols import is_sequence, list_structure_factory

c = Converter()

@c.register_structure_hook_factory(is_sequence)
def strict_list_hook_factory(type, converter):

    # First, we generate the default hook...
    list_hook = list_structure_factory(type, converter)

    # Then, we wrap it with a function of our own...
    def strict_list_hook(value, type):
        if not isinstance(value, list):
            raise ValueError("Not a list!")
```

(continues on next page)

(continued from previous page)

```
        return list_hook(value, type)

    # And finally, we return our own composite hook.
    return strict_list_hook
```

Now, all sequence structuring will be stricter:

```
>>> c.structure({"a", "b", "c"}, list[str])
Traceback (most recent call last):
    ...
ValueError: Not a list!
```

Added in version 24.1.0.

4.2.1 Customizing Named Tuples

Named tuples can be un/structured using dictionaries using the *namedtuple_dict_structure_factory* and *namedtuple_dict_unstructure_factory* hook factories.

To unstructure *all* named tuples into dictionaries:

```
>>> from typing import NamedTuple

>>> from cattrs.cols import is_namedtuple, namedtuple_dict_unstructure_factory
>>> c = Converter()

>>> c.register_unstructure_hook_factory(is_namedtuple, namedtuple_dict_unstructure_
↪factory)
<function namedtuple_dict_unstructure_factory at ...>

>>> class MyNamedTuple(NamedTuple):
...     a: int

>>> c.unstructure(MyNamedTuple(1))
{'a': 1}
```

To only un/structure *some* named tuples into dictionaries, change the predicate function when registering the hook factory:

```
>>> c.register_unstructure_hook_factory(
...     lambda t: t is MyNamedTuple,
...     namedtuple_dict_unstructure_factory,
... )
<function namedtuple_dict_unstructure_factory at ...>
```

4.3 Using `cattrs.gen` Generators

The *cattrs.gen* module allows for generating and compiling specialized hooks for unstructuring *attrs* classes, dataclasses and typed dicts. The default *Converter*, upon first encountering one of these types, will use the generation functions mentioned here to generate specialized hooks for it, register the hooks and use them.

One reason for generating these hooks in advance is that they can bypass a lot of *cattrs* machinery and be significantly faster than normal *cattrs*. The hooks are also good building blocks for more complex customizations.

Another reason is overriding behavior on a per-attribute basis.

Currently, the overrides only support generating dictionary un/structuring hooks (as opposed to tuples), and support `omit_if_default`, `forbid_extra_keys`, `rename` and `omit`.

4.3.1 omit_if_default

This override can be applied on a per-class or per-attribute basis. The generated unstructuring hook will skip unstructuring values that are equal to their default or factory values.

```
>>> from cattrs.gen import make_dict_unstructure_fn, override

>>> @define
... class WithDefault:
...     a: int
...     b: dict = Factory(dict)

>>> c = cattrs.Converter()
>>> c.register_unstructure_hook(WithDefault, make_dict_unstructure_fn(WithDefault, c,␣
↪b=override(omit_if_default=True)))
>>> c.unstructure(WithDefault(1))
{'a': 1}
```

Note that the per-attribute value overrides the per-class value. A side-effect of this is the ability to force the presence of a subset of fields. For example, consider a class with a `dateTime` field and a factory for it: skipping the unstructuring of the `dateTime` field would be inconsistent and based on the current time. So we apply the `omit_if_default` rule to the class, but not to the `dateTime` field.

Note:

```
The parameter to `make_dict_unstructure_function` is named ``_cattrs_omit_if_
↪default`` instead of just ``omit_if_default`` to avoid potential collisions with an␣
↪override for a field named ``omit_if_default``.
```

```
>>> from datetime import datetime
>>> from cattrs.gen import make_dict_unstructure_fn, override

>>> @define
... class TestClass:
...     a: Optional[int] = None
...     b: datetime = Factory(datetime.utcnow)

>>> c = cattrs.Converter()
>>> hook = make_dict_unstructure_fn(TestClass, c, _cattrs_omit_if_default=True,␣
↪b=override(omit_if_default=False))
>>> c.register_unstructure_hook(TestClass, hook)
>>> c.unstructure(TestClass())
{'b': ...}
```

This override has no effect when generating structuring functions.

4.3.2 forbid_extra_keys

By default *cattrs* is lenient in accepting unstructured input. If extra keys are present in a dictionary, they will be ignored when generating a structured object. Sometimes it may be desirable to enforce a stricter contract, and to raise an error when unknown keys are present - in particular when fields have default values this may help with catching typos. `forbid_extra_keys` can also be enabled (or disabled) on a per-class basis when creating structure hooks with *make_dict_structure_fn()*.

```
>>> from cattrs.gen import make_dict_structure_fn
>>>
>>> @define
... class TestClass:
...     number: int = 1
>>>
>>> c = cattrs.Converter(forbid_extra_keys=True)
>>> c.structure({"nummber": 2}, TestClass)
Traceback (most recent call last):
...
ForbiddenExtraKeyError: Extra fields in constructor for TestClass: nummber
>>> hook = make_dict_structure_fn(TestClass, c, _cattrs_forbid_extra_keys=False)
>>> c.register_structure_hook(TestClass, hook)
>>> c.structure({"nummber": 2}, TestClass)
TestClass(number=1)
```

This behavior can only be applied to classes or to the default for the *Converter*, and has no effect when generating unstructuring functions.

Changed in version 23.2.0: The value for the `make_dict_structure_fn._cattrs_forbid_extra_keys` parameter is now taken from the given converter by default.

4.3.3 rename

Using the rename override makes `cattrs` use the provided name instead of the real attribute name. This is useful if an attribute name is a reserved keyword in Python.

```
>>> from pendulum import DateTime
>>> from cattrs.gen import make_dict_unstructure_fn, make_dict_structure_fn, override

>>> @define
... class ExampleClass:
...     klass: Optional[int]

>>> c = cattrs.Converter()
>>> unst_hook = make_dict_unstructure_fn(ExampleClass, c, klass=override(rename="class
↪"))
>>> st_hook = make_dict_structure_fn(ExampleClass, c, klass=override(rename="class"))
>>> c.register_unstructure_hook(ExampleClass, unst_hook)
>>> c.register_structure_hook(ExampleClass, st_hook)
>>> c.unstructure(ExampleClass(1))
{'class': 1}
>>> c.structure({'class': 1}, ExampleClass)
ExampleClass(klass=1)
```

4.3.4 `omit`

This override can only be applied to individual attributes. Using the `omit` override will simply skip the attribute completely when generating a structuring or unstructuring function.

```
>>> from cattrs.gen import make_dict_unstructure_fn, override
>>>
>>> @define
... class ExampleClass:
...     an_int: int
>>>
>>> c = cattrs.Converter()
>>> unst_hook = make_dict_unstructure_fn(ExampleClass, c, an_int=override(omit=True))
>>> c.register_unstructure_hook(ExampleClass, unst_hook)
>>> c.unstructure(ExampleClass(1))
{}
```

4.3.5 `struct_hook` and `unstruct_hook`

By default, the generators will determine the right un/structure hook for each attribute of a class at time of generation according to the type of each individual attribute.

This process can be overriden by passing in the desired un/structure hook manually.

```
>>> from cattrs.gen import make_dict_structure_fn, override

>>> @define
... class ExampleClass:
...     an_int: int

>>> c = cattrs.Converter()
>>> st_hook = make_dict_structure_fn(
...     ExampleClass, c, an_int=override(struct_hook=lambda v, _: v + 1)
... )
>>> c.register_structure_hook(ExampleClass, st_hook)

>>> c.structure({"an_int": 1}, ExampleClass)
ExampleClass(an_int=2)
```

4.3.6 `use_alias`

By default, fields are un/structured to and from dictionary keys exactly matching the field names. *attrs* classes support *attrs* field aliases, which override the `__init__` parameter name for a given field. By generating your un/structure function with `_cattrs_use_alias=True`, *cattrs* will use the field alias instead of the field name as the un/structured dictionary key.

```
>>> from cattrs.gen import make_dict_structure_fn
>>>
>>> @define
... class AliasClass:
...    number: int = field(default=1, alias="count")
>>>
>>> c = cattrs.Converter()
>>> hook = make_dict_structure_fn(AliasClass, c, _cattrs_use_alias=True)
```

(continues on next page)

(continued from previous page)

```
>>> c.register_structure_hook(AliasClass, hook)
>>> c.structure({"count": 2}, AliasClass)
AliasClass(number=2)
```

Added in version 23.2.0.

4.3.7 include_init_false

By default, *attrs* fields defined as `init=False` are skipped when un/structuring. By generating your un/structure function with `_cattrs_include_init_false=True`, all `init=False` fields will be included for un/structuring.

```
>>> from cattrs.gen import make_dict_structure_fn
>>>
>>> @define
... class ClassWithInitFalse:
...     number: int = field(default=1, init=False)
>>>
>>> c = cattrs.Converter()
>>> hook = make_dict_structure_fn(ClassWithInitFalse, c, _cattrs_include_init_
↪false=True)
>>> c.register_structure_hook(ClassWithInitFalse, hook)
>>> c.structure({"number": 2}, ClassWithInitFalse)
ClassWithInitFalse(number=2)
```

A single attribute can be included by overriding it with `omit=False`.

```
>>> c = cattrs.Converter()
>>> hook = make_dict_structure_fn(ClassWithInitFalse, c, number=override(omit=False))
>>> c.register_structure_hook(ClassWithInitFalse, hook)
>>> c.structure({"number": 2}, ClassWithInitFalse)
ClassWithInitFalse(number=2)
```

Added in version 23.2.0.

CHAPTER

FIVE

STRATEGIES

cattrs ships with a number of *strategies* for customizing un/structuring behavior.

Strategies are prepackaged, high-level patterns for quickly and easily applying complex customizations to a converter.

5.1 Tagged Unions Strategy

Found at `cattrs.strategies.configure_tagged_union()`.

The *tagged union* strategy allows for un/structuring a union of classes by including an additional field (the *tag*) in the unstructured representation. Each tag value is associated with a member of the union.

```
>>> from cattrs.strategies import configure_tagged_union
>>> from cattrs import Converter
>>> converter = Converter()

>>> @define
... class A:
...     a: int

>>> @define
... class B:
...     b: str

>>> configure_tagged_union(A | B, converter)

>>> converter.unstructure(A(1), unstructure_as=A | B)
{'a': 1, '_type': 'A'}

>>> converter.structure({'a': 1, '_type': 'A'}, A | B)
A(a=1)
```

By default, the tag field name is `_type` and the tag value is the class name of the union member. Both the field name and value can be overriden.

The `tag_generator` parameter is a one-argument callable that will be called with every member of the union to generate a mapping of tag values to union members. Here are some common `tag_generator` uses:

Tag info available in	Recommended `tag_generator`
Name of the class	Use the default, or `lambda cl: cl.__name__`
A class variable (`classvar`)	`lambda cl: cl.classvar`
A dictionary (`mydict`)	`mydict.get` or `mydict.__getitem__`
An enum of possible values	Build a dictionary of classes to enum values and use it

The union members aren't required to be attrs classes or dataclasses, although those work automatically. They may be anything that cattrs can un/structure from/to a dictionary, for example a type with registered custom hooks.

A default member can be specified to be used if the tag is missing or is unknown. This is useful for evolving APIs in a backwards-compatible way; an endpoint taking class `A` can be changed to take `A | B` with `A` as the default (for old clients which do not send the tag).

This strategy only applies in the context of the union; the normal un/structuring hooks are left untouched. This also means union members can be reused in multiple unions easily.

```
# Unstructuring as a union.
>>> converter.unstructure(A(1), unstructure_as=A | B)
{'a': 1, '_type': 'A'}

# Unstructuring as just an `A`.
>>> converter.unstructure(A(1))
{'a': 1}
```

5.1.1 Real-life Case Study

The Apple App Store supports server callbacks, by which Apple sends a JSON payload to a URL of your choice. The payload can be interpreted as about a dozen different messages, based on the value of the `notificationType` field.

To keep the example simple we define two classes, one for the `REFUND` event and one for everything else.

```
@define
class Refund:
    originalTransactionId: str

@define
class OtherAppleNotification:
    notificationType: str

AppleNotification = Refund | OtherAppleNotification
```

Next, we use the *tagged unions* strategy to prepare our converter. The tag value for the `Refund` event is `REFUND`, and we can let the `OtherAppleNotification` class handle all the other cases. The `tag_generator` parameter is a callable, so we can give it the `get` method of a dictionary.

```
>>> from cattrs.strategies import configure_tagged_union

>>> c = Converter()
>>> configure_tagged_union(
...     AppleNotification,
...     c,
...     tag_name="notificationType",
...     tag_generator={Refund: "REFUND"}.get,
...     default=OtherAppleNotification
... )
```

The converter is now ready to start structuring Apple notifications.

```
>>> payload = {"notificationType": "REFUND", "originalTransactionId": "1"}
>>> notification = c.structure(payload, AppleNotification)

>>> match notification:
...     case Refund(txn_id):
```

(continues on next page)

(continued from previous page)

```
...         print(f"Refund for {txn_id}!")
...     case OtherAppleNotification(not_type):
...         print("Can't handle this yet")
Refund for 1!
```

Added in version 23.1.0.

5.2 Include Subclasses Strategy

Found at `cattrs.strategies.include_subclasses()`.

The *include subclass* strategy allows the un/structuring of a base class to an instance of itself or one of its descendants. Conceptually with this strategy, each time an un/structure operation for the base class is asked, `cattrs` machinery replaces that operation as if the union of the base class and its descendants had been asked instead.

```
>>> from attrs import define
>>> from cattrs.strategies import include_subclasses
>>> from cattrs import Converter

>>> @define
... class Parent:
...     a: int

>>> @define
... class Child(Parent):
...     b: str

>>> converter = Converter()
>>> include_subclasses(Parent, converter)

>>> converter.unstructure(Child(a=1, b="foo"), unstructure_as=Parent)
{'a': 1, 'b': 'foo'}

>>> converter.structure({'a': 1, 'b': 'foo'}, Parent)
Child(a=1, b='foo')
```

In the example above, we asked to unstructure then structure a `Child` instance as the `Parent` class and in both cases we correctly obtained back the unstructured and structured versions of the `Child` instance. If we did not apply the `include_subclasses` strategy, this is what we would have obtained:

```
>>> converter_no_subclasses = Converter()

>>> converter_no_subclasses.unstructure(Child(a=1, b="foo"), unstructure_as=Parent)
{'a': 1}

>>> converter_no_subclasses.structure({'a': 1, 'b': 'foo'}, Parent)
Parent(a=1)
```

Without the application of the strategy, in both unstructure and structure operations, we received a `Parent` instance.

Note: The handling of subclasses is an opt-in feature for two main reasons:

- Performance. While small and probably negligible in most cases the subclass handling incurs more function calls and has a performance impact.

- Customization. The specific handling of subclasses can be different from one situation to the other. In particular there is not apparent universal good defaults for disambiguating the union type. Consequently the decision is left to the user.

Warning: To work properly, all subclasses must be defined when the `include_subclasses` strategy is applied to a `converter`. If subclasses types are defined later, for instance in the context of a plug-in mechanism using inheritance, then those late defined subclasses will not be part of the subclasses union type and will not be un/structured as expected.

5.2.1 Customization

In the example shown in the previous section, the default options for `include_subclasses` work well because the `Child` class has an attribute that do not exist in the `Parent` class (the `b` attribute). The automatic union type disambiguation function which is based on finding unique fields for each type of the union works as intended.

Sometimes, more disambiguation customization is required. For instance, the unstructuring operation would have failed if `Child` did not have an extra attribute or if a sibling of `Child` had also a `b` attribute. For those cases, a callable of 2 positional arguments (a union type and a converter) defining a *tagged union strategy* can be passed to the `include_subclasses` strategy. *`configure_tagged_union()`* can be used as-is, but if you want to change its defaults, the partial function from the `functools` module in the standard library can come in handy.

```
>>> from functools import partial
>>> from attrs import define
>>> from cattrs.strategies import include_subclasses, configure_tagged_union
>>> from cattrs import Converter

>>> @define
... class Parent:
...     a: int

>>> @define
... class Child1(Parent):
...     b: str

>>> @define
... class Child2(Parent):
...     b: int

>>> converter = Converter()
>>> union_strategy = partial(configure_tagged_union, tag_name="type_name")
>>> include_subclasses(Parent, converter, union_strategy=union_strategy)

>>> converter.unstructure(Child1(a=1, b="foo"), unstructure_as=Parent)
{'a': 1, 'b': 'foo', 'type_name': 'Child1'}

>>> converter.structure({'a': 1, 'b': 1, 'type_name': 'Child2'}, Parent)
Child2(a=1, b=1)
```

Other customizations available see are (see *`include_subclasses()`*):

- The exact list of subclasses that should participate to the union with the `subclasses` argument.
- Attribute overrides that permit the customization of attributes un/structuring like renaming an attribute.

Here is an example involving both customizations:

```
>>> from attrs import define
>>> from cattrs.strategies import include_subclasses
>>> from cattrs import Converter, override

>>> @define
... class Parent:
...     a: int

>>> @define
... class Child(Parent):
...     b: str

>>> converter = Converter()
>>> include_subclasses(
...     Parent,
...     converter,
...     subclasses=(Parent, Child),
...     overrides={"b": override(rename="c")}
... )

>>> converter.unstructure(Child(a=1, b="foo"), unstructure_as=Parent)
{'a': 1, 'c': 'foo'}

>>> converter.structure({'a': 1, 'c': 'foo'}, Parent)
Child(a=1, b='foo')
```

Added in version 23.1.0.

5.3 Using Class-Specific Structure and Unstructure Methods

Found at `cattrs.strategies.use_class_methods()`.

This strategy allows for un/structuring logic on the models themselves. It can be applied for both structuring and unstructuring (also simultaneously).

If a class requires special handling for (un)structuring, you can add a dedicated (un)structuring method:

```
>>> from attrs import define
>>> from cattrs import Converter
>>> from cattrs.strategies import use_class_methods

>>> @define
... class MyClass:
...     a: int
...
...     @classmethod
...     def _structure(cls, data: dict):
...         return cls(data["b"] + 1)  # expecting "b", not "a"
...
...     def _unstructure(self):
...         return {"c": self.a - 1}  # unstructuring as "c", not "a"

>>> converter = Converter()
```

(continues on next page)

(continued from previous page)

```
>>> use_class_methods(converter, "_structure", "_unstructure")
>>> print(converter.structure({"b": 42}, MyClass))
MyClass(a=43)
>>> print(converter.unstructure(MyClass(42)))
{'c': 41}
```

Any class without a `_structure` or `_unstructure` method will use the default strategy for structuring or unstructuring, respectively. Feel free to use other names. The stategy can be applied multiple times (with different method names).

If you want to (un)structured nested objects, just append a converter parameter to your (un)structuring methods and you will receive the converter there:

```
>>> @define
... class Nested:
...     m: MyClass
...
...     @classmethod
...     def _structure(cls, data: dict, conv):
...         return cls(conv.structure(data["n"], MyClass))
...
...     def _unstructure(self, conv):
...         return {"n": conv.unstructure(self.m)}

>>> print(converter.structure({"n": {"b": 42}}, Nested))
Nested(m=MyClass(a=43))
>>> print(converter.unstructure(Nested(MyClass(42))))
{'n': {'c': 41}}
```

Added in version 23.2.0.

5.4 Union Passthrough

Found at `cattrs.strategies.configure_union_passthrough()`.

The *union passthrough* strategy enables a *Converter* to structure unions and subunions of given types.

A very common use case for *cattrs* is processing data created by other serialization libraries, such as *JSON* or *msgpack*. These libraries are able to directly produce values of unions inherent to the format. For example, every JSON library can differentiate between numbers, booleans, strings and null values since these values are represented differently in the wire format. This strategy enables *cattrs* to offload the creation of these values to an underlying library and just validate the final value. So, *cattrs* preconfigured JSON converters can handle the following type:

- `bool | int | float | str | None`

Continuing the JSON example, this strategy also enables structuring subsets of unions of these values. Accordingly, here are some examples of subset unions that are also supported:

- `bool | int`
- `int | str`
- `int | float | str`

The strategy also supports types including one or more Literals of supported types. For example:

- `Literal["admin", "user"] | int`

- `Literal[True] | str | int | float`

The strategy also supports NewTypes of these types. For example:

```
>>> from typing import NewType

>>> UserId = NewType("UserId", int)

>>> converter.loads("12", UserId)
12
```

Unions containing unsupported types can be handled if at least one union type is supported by the strategy; the supported union types will be checked before the rest (referred to as the *spillover*) is handed over to the converter again.

For example, if `A` and `B` are arbitrary *attrs* classes, the union `Literal[10] | A | B` cannot be handled directly by a JSON converter. However, the strategy will check if the value being structured matches `Literal[10]` (because this type *is* supported) and, if not, will pass it back to the converter to be structured as `A | B` (where a different strategy can handle it).

The strategy is designed to run in *O(1)* at structure time; it doesn't depend on the size of the union and the ordering of union members.

This strategy has been preapplied to the following preconfigured converters:

- `BsonConverter`
- *`Cbor2Converter`*
- *`JsonConverter`*
- *`MsgpackConverter`*
- `MsgspecJsonConverter`
- *`OrjsonConverter`*
- *`PyyamlConverter`*
- *`TomlkitConverter`*
- *`UjsonConverter`*

Added in version 23.2.0.

CHAPTER

SIX

RECIPES

This page contains a collection of recipes for custom un-/structuring mechanisms.

6.1 Switching Initializers

When structuring *attrs* classes, *cattrs* uses the classes' `__init__` method to instantiate objects by default. In certain situations, you might want to deviate from this behavior and use alternative initializers instead.

For example, consider the following `Point` class describing points in 2D space, which offers two `classmethods` for alternative creation:

```
>>> import math
>>> from attrs import define

>>> @define
... class Point:
...     """A point in 2D space."""
...     x: float
...     y: float
...
...     @classmethod
...     def from_tuple(cls, coordinates: tuple[float, float]) -> "Point":
...         """Create a point from a tuple of Cartesian coordinates."""
...         return Point(*coordinates)
...
...     @classmethod
...     def from_polar(cls, radius: float, angle: float) -> "Point":
...         """Create a point from its polar coordinates."""
...         return Point(radius * math.cos(angle), radius * math.sin(angle))
```

6.1.1 Selecting an Alternative Initializer

A simple way to *statically* set one of the `classmethods` as initializer is to register a structuring hook that holds a reference to the respective callable:

```
>>> from inspect import signature
>>> from typing import Callable, TypedDict

>>> from cattrs import Converter
>>> from cattrs.dispatch import StructureHook
```

(continues on next page)

(continued from previous page)

```
>>> def signature_to_typed_dict(fn: Callable) -> type[TypedDict]:
...     """Create a TypedDict reflecting a callable's signature."""
...     params = {p: t.annotation for p, t in signature(fn).parameters.items()}
...     return TypedDict(f"{fn.__name__}_args", params)
...

>>> def make_initializer_from(fn: Callable, conv: Converter) -> StructureHook:
...     """Return a structuring hook from a given callable."""
...     td = signature_to_typed_dict(fn)
...     td_hook = conv.get_structure_hook(td)
...     return lambda v, _: fn(**td_hook(v, td))
```

Now, you can easily structure `Points` from the specified alternative representation:

```
>>> c = Converter()
>>> c.register_structure_hook(Point, make_initializer_from(Point.from_polar, c))

>>> p0 = Point(1.0, 0.0)
>>> p1 = c.structure({"radius": 1.0, "angle": 0.0}, Point)
>>> assert p0 == p1
```

6.1.2 Dynamically Switching Between Initializers

In some cases, even more flexibility is required and the selection of the initializer must happen at runtime, requiring a dynamic approach. A typical scenario would be when object structuring happens behind an API and you want to let the user specify which representation of the object they wish to provide in their serialization string.

In such situations, the following hook factory can help you achieve your goal:

```
>>> from inspect import signature
>>> from typing import Callable, TypedDict

>>> from cattrs import Converter
>>> from cattrs.dispatch import StructureHook

>>> def signature_to_typed_dict(fn: Callable) -> type[TypedDict]:
...     """Create a TypedDict reflecting a callable's signature."""
...     params = {p: t.annotation for p, t in signature(fn).parameters.items()}
...     return TypedDict(f"{fn.__name__}_args", params)

>>> T = TypeVar("T")
>>> def make_initializer_selection_hook(
...     initializer_key: str,
...     converter: Converter,
... ) -> StructureHook:
...     """Return a structuring hook that dynamically switches between initializers.""
↪"
...
...     def select_initializer_hook(specs: dict, cls: type[T]) -> T:
...         """Deserialization with dynamic initializer selection."""
...
...         # If no initializer keyword is specified, use regular __init__
...         if initializer_key not in specs:
...             return converter.structure_attrs_fromdict(specs, cls)
...
```

(continues on next page)

(continued from previous page)

```
...         # Otherwise, call the specified initializer with deserialized arguments
...         specs = specs.copy()
...         initializer_name = specs.pop(initializer_key)
...         initializer = getattr(cls, initializer_name)
...         td = signature_to_typed_dict(initializer)
...         td_hook = converter.get_structure_hook(td)
...         return initializer(**td_hook(specs, td))
...
...     return select_initializer_hook
```

Specifying the key that determines the initializer to be used now lets you dynamically select the `classmethod` as part of the object specification itself:

```
>>> c = Converter()
>>> c.register_structure_hook(Point, make_initializer_selection_hook("initializer",␣
↪c))

>>> p0 = Point(1.0, 0.0)
>>> p1 = c.structure({"initializer": "from_polar", "radius": 1.0, "angle": 0.0},␣
↪Point)
>>> p2 = c.structure({"initializer": "from_tuple", "coordinates": (1.0, 0.0)}, Point)
>>> assert p0 == p1 == p2
```

CHAPTER

SEVEN

VALIDATION

cattrs has a detailed validation mode since version 22.1.0, and this mode is enabled by default. When running under detailed validation, the structuring hooks are slightly slower but produce richer and more precise error messages. Unstructuring hooks are not affected.

7.1 Detailed Validation

Added in version 22.1.0.

In detailed validation mode, any structuring errors will be grouped and raised together as a *cattrs.BaseValidationError*, which is a PEP 654 ExceptionGroup. ExceptionGroups are special exceptions which contain lists of other exceptions, which may themselves be other ExceptionGroups. In essence, ExceptionGroups are trees of exceptions.

When structuring a class, *cattrs* will gather any exceptions on a field-by-field basis and raise them as a *cattrs.ClassValidationError*, which is a subclass of *BaseValidationError*.

When structuring sequences and mappings, *cattrs* will gather any exceptions on a key- or index-basis and raise them as a *cattrs.IterableValidationError*, which is a subclass of *BaseValidationError*.

The exceptions will also have their `__notes__` attributes set, as per PEP 678, showing the field, key or index for each inner exception.

A simple example involving a class containing a list and a dictionary:

```
@define
class Class:
    a_list: list[int]
    a_dict: dict[str, int]

>>> structure({"a_list": ["a"], "a_dict": {"str": "a"}}, Class)
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |   File "/Users/tintvrtkovic/pg/cattrs/src/cattr/converters.py", line 276, in␣
↪structure
  |     return self._structure_func.dispatch(cl)(obj, cl)
  |   File "<cattrs generated structure __main__.Class>", line 14, in structure_Class
  |     if errors: raise __c_cve('While structuring Class', errors, __cl)
  | cattrs.errors.ClassValidationError: While structuring Class
  +-+---------------- 1 ----------------
    | Exception Group Traceback (most recent call last):
    |   File "<cattrs generated structure __main__.Class>", line 5, in structure_Class
    |     res['a_list'] = __c_structure_a_list(o['a_list'], __c_type_a_list)
    |   File "/Users/tintvrtkovic/pg/cattrs/src/cattr/converters.py", line 457, in _
```

(continues on next page)

(continued from previous page)

```
↪structure_list
    |     raise IterableValidationError(
    | cattrs.errors.IterableValidationError: While structuring list[int]
    | Structuring class Class @ attribute a_list
    +-+---------------- 1 ----------------
      | Traceback (most recent call last):
      |   File "/Users/tintvrtkovic/pg/cattrs/src/cattr/converters.py", line 450, in _
↪structure_list
      |     res.append(handler(e, elem_type))
      |   File "/Users/tintvrtkovic/pg/cattrs/src/cattr/converters.py", line 375, in _
↪structure_call
      |     return cl(obj)
      | ValueError: invalid literal for int() with base 10: 'a'
      | Structuring list[int] @ index 0
      +------------------------------------
    +---------------- 2 ----------------
    | Exception Group Traceback (most recent call last):
    |   File "<cattrs generated structure __main__.Class>", line 10, in structure_
↪Class
    |     res['a_dict'] = __c_structure_a_dict(o['a_dict'], __c_type_a_dict)
    |   File "", line 17, in structure_mapping
    | cattrs.errors.IterableValidationError: While structuring dict
    | Structuring class Class @ attribute a_dict
    +-+---------------- 1 ----------------
      | Traceback (most recent call last):
      |   File "", line 5, in structure_mapping
      | ValueError: invalid literal for int() with base 10: 'a'
      | Structuring mapping value @ key 'str'
      +------------------------------------
```

7.1.1 Transforming Exceptions into Error Messages

Added in version 23.1.0.

ExceptionGroup stack traces are useful while developing, but sometimes a more compact representation of validation errors is required. *cattrs* provides a helper function, *cattrs.transform_error()*, which transforms validation errors into lists of error messages.

The example from the previous paragraph produces the following error messages:

```
>>> from cattrs import structure, transform_error

>>> try:
...     structure({"a_list": ["a"], "a_dict": {"str": "a"}}, Class)
... except Exception as exc:
...     print(transform_error(exc))
['invalid value for type, expected int @ $.a_list[0]', "invalid value for type,␣
↪expected int @ $.a_dict['str']"]
```

A small number of built-in exceptions are converted into error messages automatically. This can be further customized by providing *cattrs.transform_error()* with a function that it can use to turn individual, non-ExceptionGroup exceptions into error messages. A useful pattern is wrapping the default, *cattrs.v.format_exception()* function.

```
>>> from cattrs.v import format_exception

>>> def my_exception_formatter(exc: BaseException, type) -> str:
...     if isinstance(exc, MyInterestingException):
...         return "My error message"
...     return format_exception(exc, type)

>>> try:
...     structure(..., Class)
... except Exception as exc:
...     print(transform_error(exc, format_exception=my_exception_formatter))
```

If even more customization is required, *cattrs.transform_error()* can be copied over into your codebase and adjusted as needed.

7.2 Non-detailed Validation

Non-detailed validation can be enabled by initializing any of the converters with `detailed_validation=False`. In this mode, any errors during un/structuring will bubble up directly as soon as they happen.

CHAPTER

EIGHT

PRECONFIGURED CONVERTERS

The *cattrs.preconf* package contains factories for preconfigured converters, specifically adjusted for particular serialization libraries.

For example, to get a converter configured for BSON:

```
>>> from cattrs.preconf.bson import make_converter

>>> converter = make_converter() # Takes the same parameters as the `cattrs.Converter`
```

Converters obtained this way can be customized further, just like any other converter.

These converters support all *default hooks* and the following additional classes and type annotations, both for structuring and unstructuring:

- `datetime.datetime`, `datetime.date`

Added in version 22.1.0: All preconf converters now have `loads` and `dumps` methods, which combine un/structuring and the de/serialization logic from their underlying libraries.

```
>>> from cattrs.preconf.json import make_converter

>>> converter = make_converter()

>>> @define
... class Test:
...     a: int

>>> converter.dumps(Test(1))
'{"a": 1}'
```

Particular libraries may have additional constraints documented below.

Third-party libraries can be specified as optional (extra) dependencies on *cattrs* during installation. Optional install targets should match the name of the *cattrs.preconf* modules.

```
# Using pip
$ pip install cattrs[ujson]

# Using pdm
$ pdm add cattrs[orjson]

# Using poetry
$ poetry add --extras tomlkit cattrs
```

8.1 Standard Library *json*

Found at `cattrs.preconf.json`.

Bytes are serialized as base 85 strings. Counters are serialized as dictionaries. Sets are serialized as lists, and deserialized back into sets. `datetime` s and `date` s are serialized as ISO 8601 strings.

8.2 *orjson*

Found at `cattrs.preconf.orjson`.

Bytes are un/structured as base 85 strings. Sets are unstructured into lists, and structured back into sets. `datetime` s and `date` s are passed through to be unstructured into RFC 3339 by *orjson* itself. Typed named tuples are unstructured into ordinary tuples, and then into JSON arrays by *orjson*.

orjson doesn't support integers less than -9223372036854775808, and greater than 9223372036854775807. *orjson* only supports mappings with string keys so mappings will have their keys stringified before serialization, and destringified during deserialization.

8.3 *msgspec*

Found at `cattrs.preconf.msgspec`. Only JSON functionality is currently available, other formats supported by msgspec to follow in the future.

msgspec structs are supported, but not composable - a struct will be handed over to *msgspec* directly, and *msgspec* will handle and all of its fields, recursively. *cattrs* may get more sophisticated handling of structs in the future.

msgspec strict mode is used by default. This can be customized by changing the `encoder` attribute on the converter.

What *cattrs* calls *unstructuring* and *structuring*, *msgspec* calls `to_builtins` and `convert`. What *cattrs* refers to as *dumping* and *loading*, *msgspec* refers to as `encoding` and `decoding`.

Compatibility notes:

- Bytes are un/structured as base 64 strings directly by *msgspec* itself.
- *msgspec* encodes special float values (`NaN, Inf, -Inf`) as `null`.
- `datetime` s and `date` s are passed through to be unstructured into RFC 3339 by *msgspec* itself.
- *attrs* classes, dataclasses and sequences are handled directly by *msgspec* if possible, otherwise by the normal *cattrs* machinery. This means it's possible the validation errors produced may be *msgspec* validation errors instead of *cattrs* validation errors.

This converter supports `get_loads_hook()` and `get_dumps_hook()`. These are factories for dumping and loading functions (as opposed to unstructuring and structuring); the hooks returned by this may be further optimized to offload as much work as possible to *msgspec*.

```
>>> from cattrs.preconf.msgspec import make_converter

>>> @define
... class Test:
...     a: int

>>> converter = make_converter()
>>> dumps = converter.get_dumps_hook(A)
```

(continues on next page)

(continued from previous page)

```
>>> dumps(Test(1))   # Will use msgspec directly.
b'{"a":1}'
```

Due to its complexity, this converter is currently *provisional* and may slightly change as the best integration patterns are discovered.

msgspec doesn't support PyPy.

Added in version 24.1.0.

8.4 *ujson*

Found at *cattrs.preconf.ujson*.

Bytes are serialized as base 85 strings. Sets are serialized as lists, and deserialized back into sets. `datetime` s and `date` s are serialized as ISO 8601 strings.

ujson doesn't support integers less than -9223372036854775808, and greater than 9223372036854775807, nor does it support `float('inf')`.

8.5 *msgpack*

Found at *cattrs.preconf.msgpack*.

Sets are serialized as lists, and deserialized back into sets. `datetime` s are serialized as UNIX timestamp float values. `date` s are serialized as midnight-aligned UNIX timestamp float values.

msgpack doesn't support integers less than -9223372036854775808, and greater than 18446744073709551615.

When parsing msgpack data from bytes, the library needs to be passed `strict_map_key=False` to get the full range of compatibility.

8.6 *cbor2*

Found at *cattrs.preconf.cbor2*.

cbor2 implements a fully featured CBOR encoder with several extensions for handling shared references, big integers, rational numbers and so on.

Sets are serialized and deserialized to sets. Tuples are serialized as lists.

`datetime` s are serialized as a text string by default (CBOR Tag 0). Use keyword argument `datetime_as_timestamp=True` to encode as UNIX timestamp integer/float (CBOR Tag 1) **note:** this replaces timezone information as UTC.

`date` s are serialized as ISO 8601 strings.

Use keyword argument `canonical=True` for efficient encoding to the smallest binary output.

Floats can be forced to smaller output by casting to lower-precision formats by casting to `numpy` floats (and back to Python floats). Example: `float(np.float32(value))` or `float(np.float16(value))`

Added in version 23.1.0.

8.7 *bson*

Found at `cattrs.preconf.bson`. Tested against the *bson* module bundled with the *pymongo* library, not the standalone PyPI *bson* package.

Sets are serialized as lists, and deserialized back into sets.

bson doesn't support integers less than -9223372036854775808 or greater than 9223372036854775807 (64-bit signed). *bson* does not support null bytes in mapping keys. *bson* only supports mappings with string keys so mappings will have their keys stringified before serialization, and destringified during deserialization. The *bson* datetime representation doesn't support microsecond accuracy. `date` s are serialized as ISO 8601 strings.

When encoding and decoding, the library needs to be passed `codec_options=bson.CodecOptions(tz_aware=True)` to get the full range of compatibility.

8.8 *pyyaml*

Found at *cattrs.preconf.pyyaml*.

Frozensets are serialized as lists, and deserialized back into frozensets. `date` s are serialized as ISO 8601 strings. Typed named tuples are unstructured into ordinary tuples, and then into YAML arrays by *pyyaml*.

8.9 *tomlkit*

Found at *cattrs.preconf.tomlkit*.

Bytes are serialized as base 85 strings. Sets are serialized as lists, and deserialized back into sets. Tuples are serialized as lists, and deserialized back into tuples. *tomlkit* only supports mappings with string keys so mappings will have their keys stringified before serialization, and destringified during deserialization. `date` s are serialized as ISO 8601 strings.

CHAPTER

NINE

HANDLING UNIONS

cattrs is able to handle simple unions of *attrs* classes and dataclasses *automatically*. More complex cases require converter customization (since there are many ways of handling unions).

cattrs also comes with a number of optional strategies to help handle unions:

- *tagged unions strategy* mentioned below
- *union passthrough strategy*, which is preapplied to all the *preconfigured* converters

9.1 Default Union Strategy

For convenience, *cattrs* includes a default union structuring strategy which is a little more opinionated.

Given a union of several *attrs* classes and/or dataclasses, the default union strategy will attempt to handle it in several ways.

First, it will look for `Literal` fields. If *all members* of the union contain a literal field, *cattrs* will generate a disambiguation function based on the field.

```
from typing import Literal

@define
class ClassA:
    field_one: Literal["one"]

@define
class ClassB:
    field_one: Literal["two"] = "two"
```

In this case, a payload containing `{"field_one": "one"}` will produce an instance of `ClassA`.

Note: The following snippet can be used to disable the use of literal fields, restoring legacy behavior.

```
from functools import partial
from cattrs.disambiguators import is_supported_union

converter.register_structure_hook_factory(
    is_supported_union,
    partial(converter._gen_attrs_union_structure, use_literals=False),
)
```

If there are no appropriate fields, the strategy will examine the classes for **unique required fields**.

So, given a union of `ClassA` and `ClassB`:

```
@define
class ClassA:
    field_one: str
    field_with_default: str = "a default"

@define
class ClassB:
    field_two: str
```

the strategy will determine that if a payload contains the key `field_one` it should be handled as `ClassA`, and if it contains the key `field_two` it should be handled as `ClassB`. The field `field_with_default` will not be considered since it has a default value, so it gets treated as optional.

Changed in version 23.2.0: Literals can now be potentially used to disambiguate.

Changed in version 24.1.0: Dataclasses are now supported in addition to *attrs* classes.

9.2 Unstructuring Unions with Extra Metadata

Note: *cattrs* comes with the *tagged unions strategy* for handling this exact use-case since version 23.1. The example below has been left here for educational purposes, but you should prefer the strategy.

Let's assume a simple scenario of two classes, `ClassA` and `ClassB`, both of which have no distinct fields and so cannot be used automatically with *cattrs*.

```
@define
class ClassA:
    a_string: str

@define
class ClassB:
    a_string: str
```

A naive approach to unstructuring either of these would yield identical dictionaries, and not enough information to re-structure the classes.

```
>>> converter.unstructure(ClassA("test"))
{'a_string': 'test'}  # Is this ClassA or ClassB? Who knows!
```

What we can do is ensure some extra information is present in the unstructured data, and then use that information to help structure later.

First, we register an unstructure hook for the `Union[ClassA, ClassB]` type.

```
>>> converter.register_unstructure_hook(
...     Union[ClassA, ClassB],
...     lambda o: {"_type": type(o).__name__,  **converter.unstructure(o)}
... )
>>> converter.unstructure(ClassA("test"), unstructure_as=Union[ClassA, ClassB])
{'_type': 'ClassA', 'a_string': 'test'}
```

Note that when unstructuring, we had to provide the `unstructure_as` parameter or *cattrs* would have just applied the usual unstructuring rules to `ClassA`, instead of our special union hook.

Now that the unstructured data contains some information, we can create a structuring hook to put it to use:

```
>>> converter.register_structure_hook(
...     Union[ClassA, ClassB],
...     lambda o, _: converter.structure(o, ClassA if o["_type"] == "ClassA" else␣
↪ClassB)
... )
>>> converter.structure({"_type": "ClassA", "a_string": "test"}, Union[ClassA,␣
↪ClassB])
ClassA(a_string='test')
```

CHAPTER

TEN

ADVANCED EXAMPLES

This section covers advanced use examples of *cattrs* features.

10.1 Using Factory Hooks

For this example, let's assume you have some attrs classes with snake case attributes, and you want to un/structure them as camel case.

Warning: A simpler and better approach to this problem is to simply make your class attributes camel case. However, this is a good example of the power of hook factories and *cattrs'* composition-based design.

Here's our simple data model:

```
@define
class Inner:
    a_snake_case_int: int
    a_snake_case_float: float
    a_snake_case_str: str

@define
class Outer:
    a_snake_case_inner: Inner
```

Let's examine our options one by one, starting with the simplest: writing manual un/structuring hooks.

We just write the code by hand and register it:

```
def unstructure_inner(inner):
    return {
        "aSnakeCaseInt": inner.a_snake_case_int,
        "aSnakeCaseFloat": inner.a_snake_case_float,
        "aSnakeCaseStr": inner.a_snake_case_str
    }

>>> converter.register_unstructure_hook(Inner, unstructure_inner)
```

(Let's skip the other unstructure hook and 2 structure hooks due to verbosity.)

This will get us where we want to go, but the drawbacks are immediately obvious: we'd need to write a ton of code ourselves, wasting effort, increasing our maintenance burden and risking bugs. Obviously this won't do.

Why write code when we can write code to write code for us? In this case this code has already been written for you. *cattrs* contains a module, `cattrs.gen`, with functions to automatically generate hooks exactly like this. These functions also take parameters to customize the generated hooks.

We can generate and register the renaming hooks we need:

```
>>> from cattrs.gen import make_dict_unstructure_fn, override

>>> converter.register_unstructure_hook(
...     Inner,
...      make_dict_unstructure_fn(
...          Inner,
...          converter,
...          a_snake_case_int=override(rename="aSnakeCaseInt"),
...          a_snake_case_float=override(rename="aSnakeCaseFloat"),
...          a_snake_case_str=override(rename="aSnakeCaseStr"),
...      )
...  )
```

(Again skipping the other hooks due to verbosity.)

This is still too verbose and manual for our tastes, so let's automate it further. We need a way to convert snake case identifiers to camel case, so let's grab one from Stack Overflow:

```
def to_camel_case(snake_str: str) -> str:
    components = snake_str.split("_")
    return components[0] + "".join(x.title() for x in components[1:])
```

We can combine this with `attrs.fields` to save us some typing:

```
from attrs import fields
from cattrs.gen import make_dict_unstructure_fn, override

converter.register_unstructure_hook(
    Inner,
    make_dict_unstructure_fn(
        Inner,
        converter,
        **{a.name: override(rename=to_camel_case(a.name)) for a in fields(Inner)}
    )
)

converter.register_unstructure_hook(
    Outer,
    make_dict_unstructure_fn(
        Outer,
        converter,
        **{a.name: override(rename=to_camel_case(a.name)) for a in fields(Outer)}
    )
)
```

(Skipping the structuring hooks due to verbosity.)

Now we're getting somewhere, but we still need to do this for each class separately. The final step is using hook factories instead of hooks directly.

Hook factories are functions that return hooks. They are also registered using predicates instead of being attached to classes directly, like normal un/structure hooks. Predicates are functions that given a type return a boolean whether they handle it.

We want our hook factories to trigger for all *attrs* classes, so we need a predicate to recognize whether a type is an *attrs* class. Luckily, *attrs* comes with `attrs.has`, which is exactly this.

As the final step, we can combine all of this into two hook factories:

```
from attrs import has, fields
from cattrs import Converter
from cattrs.gen import make_dict_unstructure_fn, make_dict_structure_fn, override

converter = Converter()

def to_camel_case(snake_str: str) -> str:
    components = snake_str.split("_")
    return components[0] + "".join(x.title() for x in components[1:])

def to_camel_case_unstructure(cls):
    return make_dict_unstructure_fn(
        cls,
        converter,
        **{
            a.name: override(rename=to_camel_case(a.name))
            for a in fields(cls)
        }
    )

def to_camel_case_structure(cls):
    return make_dict_structure_fn(
        cls,
        converter,
        **{
            a.name: override(rename=to_camel_case(a.name))
            for a in fields(cls)
        }
    )

converter.register_unstructure_hook_factory(
    has, to_camel_case_unstructure
)
converter.register_structure_hook_factory(
    has, to_camel_case_structure
)
```

The `converter` instance will now un/structure every attrs class to camel case. Nothing has been omitted from this final example; it's complete.

10.2 Using Fallback Key Names

Sometimes when structuring data, the input data may be in multiple formats that need to be converted into a common attribute.

Consider an example where a data store creates a new schema version and renames a key (ie, `{'old_field': 'value1'}` in v1 becomes `{'new_field': 'value1'}` in v2), while also leaving existing records in the system with the V1 schema. Both keys should convert to the same field.

Here, builtin customizations such as *rename* are insufficient - *cattrs* cannot structure both `old_field` and `new_field` into a single field using `rename`, at least not on the same converter.

In order to support both fields, you can apply a little preprocessing to the default *cattrs* structuring hooks. One approach is to write the following decorator and apply it to your class.

```
from attrs import define
from cattrs import Converter
from cattrs.gen import make_dict_structure_fn

converter = Converter()


def fallback_field(
    converter_arg: Converter,
    old_to_new_field: dict[str, str]
):
    def decorator(cls):
        struct = make_dict_structure_fn(cls, converter_arg)

        def structure(d, cl):
            for k, v in old_to_new_field.items():
                if k in d:
                    d[v] = d[k]

            return struct(d, cl)

        converter_arg.register_structure_hook(cls, structure)

        return cls

    return decorator


@fallback_field(converter, {"old_field": "new_field"})
@define
class MyInternalAttr:
    new_field: str
```

cattrs will now structure both key names into `new_field` on your class.

```
converter.structure({"new_field": "foo"}, MyInternalAttr)
converter.structure({"old_field": "foo"}, MyInternalAttr)
```

CHAPTER

ELEVEN

CONVERTERS IN-DEPTH

Converters are registries of rules *cattrs* uses to perform function composition and generate its un/structuring functions.

Currently, a converter contains the following state:

- a registry of unstructure hooks, backed by a singledispatch and a *`FunctionDispatch`*, wrapped in a cache.
- a registry of structure hooks, backed by a different singledispatch and `FunctionDispatch`, and a different cache.
- a `detailed_validation` flag (defaulting to true), determining whether the converter uses *detailed validation*.
- a reference to *`an unstructuring strategy`* (either AS_DICT or AS_TUPLE).
- a `prefer_attrib_converters` flag (defaulting to false), determining whether to favor *attrs* converters over normal *cattrs* machinery when structuring *attrs* classes
- a `dict_factory` callable, a legacy parameter used for creating `dicts` when dumping *attrs* classes using `AS_DICT`.

Converters may be cloned using the *`Converter.copy()`* method. The new copy may be changed through the `copy` arguments, but will retain all manually registered hooks from the original.

11.1 Customizing Collection Unstructuring

Important: This feature is supported for Python 3.9 and later.

Tip: See *Customizing Collections* for a more modern and more powerful way of customizing collection handling.

Overriding collection unstructuring in a generic way can be a very useful feature. A common example is using a JSON library that doesn't support sets, but expects lists and tuples instead.

Using ordinary unstructuring hooks for this is unwieldy due to the semantics of singledispatch; in other words, you'd need to register hooks for all specific types of set you're using (`set[int]`, `set[float]`, `set[str]`...), which is not useful.

Function-based hooks can be used instead, but come with their own set of challenges - they're complicated to write efficiently.

The *`Converter`* supports easy customizations of collection unstructuring using its `unstruct_collection_overrides` parameter. For example, to unstructure all sets into lists, use the following:

```
>>> from collections.abc import Set
>>> converter = cattrs.Converter(unstruct_collection_overrides={Set: list})

>>> converter.unstructure({1, 2, 3})
[1, 2, 3]
```

Going even further, the `Converter` contains heuristics to support the following Python types, in order of decreasing generality:

- `typing.Sequence`, `typing.MutableSequence`, `list`, `deque`, `tuple`
- `typing.Set`, `frozenset`, `typing.MutableSet`, `set`
- `typing.Mapping`, `typing.MutableMapping`, `dict`, `defaultdict`, `collections.OrderedDict`, `collections.Counter`

For example, if you override the unstructure type for `Sequence`, but not for `MutableSequence`, `list` or `tuple`, the override will also affect those types. An easy way to remember the rule:

- all `MutableSequence` s are `Sequence` s, so the override will apply
- all `list` s are `MutableSequence` s, so the override will apply
- all `tuple` s are `Sequence` s, so the override will apply

If, however, you override only `MutableSequence`, fields annotated as `Sequence` will not be affected (since not all sequences are mutable sequences), and fields annotated as tuples will not be affected (since tuples are not mutable sequences in the first place).

Similar logic applies to the set and mapping hierarchies.

Make sure you're using the types from `collections.abc` on Python 3.9+, and from `typing` on older Python versions.

11.2 Fallback Hook Factories

By default, when a *converter* cannot handle a type it will:

- when unstructuring, pass the value through unchanged
- when structuring, raise a *cattrs.errors.StructureHandlerNotFoundError* asking the user to add configuration

These behaviors can be customized by providing custom *hook factories* when creating the converter.

```
>>> from pickle import dumps

>>> class Unsupported:
...     """An artisinal (non-attrs) class, unsupported by default."""

>>> converter = Converter(unstructure_fallback_factory=lambda _: dumps)
>>> instance = Unsupported()
>>> converter.unstructure(instance)
b'\x80\x04\x95\x18\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__\x94\x8c\x04Test\x94\
↪x93\x94)\x81\x94.'
```

This also enables converters to be chained.

```
>>> parent = Converter()

>>> child = Converter(
...     unstructure_fallback_factory=parent.get_unstructure_hook,
...     structure_fallback_factory=parent.get_structure_hook,
... )
```

Added in version 23.2.0.

11.3 cattrs.Converter

The *Converter* is a converter variant that automatically generates, compiles and caches specialized structuring and unstructuring hooks for *attrs* classes, dataclasses and TypedDicts.

`Converter` differs from the *cattrs.BaseConverter* in the following ways:

- structuring and unstructuring of *attrs* classes is slower the first time, but faster every subsequent time
- structuring and unstructuring can be customized
- support for *attrs* classes with PEP563 (postponed) annotations
- support for generic *attrs* classes
- support for easy overriding collection unstructuring

The *Converter* used to be called `GenConverter`, and that alias is still present for backwards compatibility.

11.4 cattrs.BaseConverter

The *BaseConverter* is a simpler and slower converter variant. It does no code generation, so it may be faster on first-use which can be useful in specific cases, like CLI applications where startup time is more important than throughput.

CHAPTER

TWELVE

CATTRS

12.1 cattrs package

cattrs.**structure**(*obj*, *cl*)

Convert unstructured Python data structures to structured data.

Parameters

- **obj** (*Any*)
- **cl** (*type[T]*)

Return type

T

cattrs.**unstructure**(*obj*, *unstructure_as=None*)

Parameters

- **obj** (*Any*)
- **unstructure_as** (*Any*)

Return type

Any

cattrs.**get_structure_hook**(*type*, *cache_result=True*)

Get the structure hook for the given type.

This hook can be manually called, or composed with other functions and re-registered.

If no hook is registered, the converter structure fallback factory will be used to produce one.

Parameters

- **cache** – Whether to cache the returned hook.
- **type** (*Any*)
- **cache_result** (*bool*)

Return type

Callable[[*Any*, *Any*], *Any*]

Added in version 24.1.0.

cattrs.**get_unstructure_hook**(*type*, *cache_result=True*)

Get the unstructure hook for the given type.

This hook can be manually called, or composed with other functions and re-registered.

If no hook is registered, the converter unstructure fallback factory will be used to produce one.

Parameters

- **cache** – Whether to cache the returned hook.
- **type** (*Any*)
- **cache_result** (*bool*)

Return type
Callable[[*Any*], *Any*]

Added in version 24.1.0.

cattrs.**register_structure_hook_func**(*check_func*, *func*)

Register a class-to-primitive converter function for a class, using a function to check if it's a match.

Parameters

- **check_func** (*Callable[[Any], bool]*)
- **func** (*Callable[[Any, Any], Any]*)

Return type
None

cattrs.**register_structure_hook**(*cl*, *func=None*)

Register a primitive-to-class converter function for a type.

The converter function should take two arguments:

- a Python object to be converted,
- the type to convert to

and return the instance of the class. The type may seem redundant, but is sometimes needed (for example, when dealing with generic classes).

This method may be used as a decorator. In this case, the decorated hook must have a return type annotation, and this annotation will be used as the type for the hook.

Changed in version 24.1.0: This method may now be used as a decorator.

Parameters

- **cl** (*Any*)
- **func** (*Callable[[Any, Any], Any] | None*)

Return type
None

cattrs.**register_unstructure_hook_func**(*check_func*, *func*)

Register a class-to-primitive converter function for a class, using a function to check if it's a match.

Parameters

- **check_func** (*Callable[[Any], bool]*)
- **func** (*Callable[[Any], Any]*)

Return type
None

cattrs.**register_unstructure_hook**(*cls=None*, *func=None*)

Register a class-to-primitive converter function for a class.

The converter function should take an instance of the class and return its Python equivalent.

May also be used as a decorator. When used as a decorator, the first argument annotation from the decorated function will be used as the type to register the hook for.

Changed in version 24.1.0: This method may now be used as a decorator.

Parameters

- **cls** (*Any*)
- **func** (*UnstructureHook | None*)

Return type
Callable[[UnstructureHook]] | None

cattrs.**structure_attrs_fromdict**(*obj*, *cl*)

Instantiate an attrs class from a mapping (dict).

Parameters

- **obj** (*Mapping[str, Any]*)
- **cl** (*type[T]*)

Return type
T

cattrs.**structure_attrs_fromtuple**(*obj*, *cl*)

Load an attrs class from a sequence (tuple).

Parameters

- **obj** (*tuple[Any, ...]*)
- **cl** (*type[T]*)

Return type
T

cattrs.**global_converter: Final = <cattrs.converters.Converter object>**

The global converter. Prefer creating your own if customizations are required.

class cattrs.**BaseConverter**(*dict_factory=<class 'dict'>*, *unstruct_strat=UnstructureStrategy.AS_DICT*, *prefer_attrib_converters=False*, *detailed_validation=True*, *unstructure_fallback_factory=<function BaseConverter.<lambda>>*, *structure_fallback_factory=<function BaseConverter.<lambda>>*)

Bases: object

Converts between structured and unstructured data.

Parameters

- **detailed_validation** (*bool*) – Whether to use a slightly slower mode for detailed validation errors.
- **unstructure_fallback_factory** (*HookFactory[UnstructureHook]*) – A hook factory to be called when no registered unstructuring hooks match.

- **structure_fallback_factory** (*HookFactory[StructureHook]*) – A hook factory to be called when no registered structuring hooks match.
- **dict_factory** (*Callable[[], Any]*)
- **unstruct_strat** (UnstructureStrategy)
- **prefer_attrib_converters** (*bool*)

Added in version 23.2.0: *unstructure_fallback_factory*

Added in version 23.2.0: *structure_fallback_factory*

detailed_validation

unstructure(*obj*, *unstructure_as=None*)

Parameters

- **obj** (*Any*)
- **unstructure_as** (*Any*)

Return type
Any

property unstruct_strat: *UnstructureStrategy*

The default way of unstructuring `attrs` classes.

register_unstructure_hook(*cls: UnstructureHookT*) → UnstructureHookT

register_unstructure_hook(*cls: Any*, *func: UnstructureHook*) → None

Register a class-to-primitive converter function for a class.

The converter function should take an instance of the class and return its Python equivalent.

May also be used as a decorator. When used as a decorator, the first argument annotation from the decorated function will be used as the type to register the hook for.

Changed in version 24.1.0: This method may now be used as a decorator.

register_unstructure_hook_func(*check_func*, *func*)

Register a class-to-primitive converter function for a class, using a function to check if it's a match.

Parameters

- **check_func** (*Callable[[Any], bool]*)
- **func** (*Callable[[Any], Any]*)

Return type
None

register_unstructure_hook_factory(*predicate: Predicate*) → Callable[[AnyUnstructureHookFactoryBase], AnyUnstructureHookFactoryBase]

register_unstructure_hook_factory(*predicate: Predicate*, *factory: UnstructureHookFactory*) → UnstructureHookFactory

register_unstructure_hook_factory(*predicate: Predicate*, *factory: ExtendedUnstructureHookFactory[BaseConverter]*) → ExtendedUnstructureHookFactory[*BaseConverter*]

Register a hook factory for a given predicate.

The hook factory may expose an additional required parameter. In this case, the current converter will be provided to the hook factory as that parameter.

May also be used as a decorator.

Parameters

- **predicate** – A function that, given a type, returns whether the factory can produce a hook for that type.
- **factory** – A callable that, given a type, produces an unstructuring hook for that type. This unstructuring hook will be cached.

Changed in version 24.1.0: This method may now be used as a decorator. The factory may also receive the converter as a second, required argument.

get_unstructure_hook(*type*, *cache_result=True*)

Get the unstructure hook for the given type.

This hook can be manually called, or composed with other functions and re-registered.

If no hook is registered, the converter unstructure fallback factory will be used to produce one.

Parameters

- **cache** – Whether to cache the returned hook.
- **type** (*Any*)
- **cache_result** (*bool*)

Return type
Callable[[*Any*], *Any*]

Added in version 24.1.0.

register_structure_hook(*cl: StructureHookT*) → StructureHookT

register_structure_hook(*cl: Any*, *func: StructureHook*) → None

Register a primitive-to-class converter function for a type.

The converter function should take two arguments:

- a Python object to be converted,
- the type to convert to

and return the instance of the class. The type may seem redundant, but is sometimes needed (for example, when dealing with generic classes).

This method may be used as a decorator. In this case, the decorated hook must have a return type annotation, and this annotation will be used as the type for the hook.

Changed in version 24.1.0: This method may now be used as a decorator.

register_structure_hook_func(*check_func*, *func*)

Register a class-to-primitive converter function for a class, using a function to check if it's a match.

Parameters

- **check_func** (*Callable[[Any], bool]*)
- **func** (*Callable[[Any, Any], Any]*)

Return type
None

register_structure_hook_factory(*predicate: Predicate*) → Callable[[AnyStructureHookFactoryBase], AnyStructureHookFactoryBase]

register_structure_hook_factory(*predicate: Predicate*, *factory: StructureHookFactory*) → StructureHookFactory

register_structure_hook_factory(*predicate: Predicate*, *factory: ExtendedStructureHookFactory[BaseConverter]*) → ExtendedStructureHookFactory[*BaseConverter*]

Register a hook factory for a given predicate.

The hook factory may expose an additional required parameter. In this case, the current converter will be provided to the hook factory as that parameter.

May also be used as a decorator.

Parameters

- **predicate** – A function that, given a type, returns whether the factory can produce a hook for that type.
- **factory** – A callable that, given a type, produces a structuring hook for that type. This structuring hook will be cached.

Changed in version 24.1.0: This method may now be used as a decorator. The factory may also receive the converter as a second, required argument.

structure(*obj*, *cl*)

Convert unstructured Python data structures to structured data.

Parameters

- **obj** (*Any*)
- **cl** (*type[T]*)

Return type
T

get_structure_hook(*type*, *cache_result=True*)

Get the structure hook for the given type.

This hook can be manually called, or composed with other functions and re-registered.

If no hook is registered, the converter structure fallback factory will be used to produce one.

Parameters

- **cache** – Whether to cache the returned hook.
- **type** (*Any*)
- **cache_result** (*bool*)

Return type
Callable[[*Any*, *Any*], *Any*]

Added in version 24.1.0.

unstructure_attrs_asdict(*obj*)

Our version of *attrs.asdict*, so we can call back to us.

Parameters
obj (*Any*)

Return type
dict[str, *Any*]

unstructure_attrs_astuple(*obj*)

Our version of *attrs.astuple*, so we can call back to us.

Parameters
obj (*Any*)

Return type
tuple[*Any*, …]

structure_attrs_fromtuple(*obj*, *cl*)

Load an attrs class from a sequence (tuple).

Parameters

- **obj** (*tuple[Any, ...]*)
- **cl** (*type[T]*)

Return type
T

structure_attrs_fromdict(*obj*, *cl*)

Instantiate an attrs class from a mapping (dict).

Parameters

- **obj** (*Mapping[str, Any]*)
- **cl** (*type[T]*)

Return type
T

copy(*dict_factory=None*, *unstruct_strat=None*, *prefer_attrib_converters=None*, *detailed_validation=None*)

Create a copy of the converter, keeping all existing custom hooks.

Parameters

- **detailed_validation** (*bool | None*) – Whether to use a slightly slower mode for detailed validation errors.
- **dict_factory** (*Callable[[], Any] | None*)
- **unstruct_strat** (*UnstructureStrategy | None*)
- **prefer_attrib_converters** (*bool | None*)

Return type
BaseConverter

class cattrs.**Converter**(*dict_factory=<class 'dict'>*, *unstruct_strat=UnstructureStrategy.AS_DICT*, *omit_if_default=False*, *forbid_extra_keys=False*, *type_overrides={}*, *unstruct_collection_overrides={}*, *prefer_attrib_converters=False*, *detailed_validation=True*, *unstructure_fallback_factory=<function Converter.<lambda>>*, *structure_fallback_factory=<function Converter.<lambda>>*)

Bases: *BaseConverter*

A converter which generates specialized un/structuring functions.

Parameters

- **detailed_validation** (*bool*) – Whether to use a slightly slower mode for detailed validation errors.
- **unstructure_fallback_factory** (*HookFactory[UnstructureHook]*) – A hook factory to be called when no registered unstructuring hooks match.
- **structure_fallback_factory** (*HookFactory[StructureHook]*) – A hook factory to be called when no registered structuring hooks match.
- **dict_factory** (*Callable[[], Any]*)
- **unstruct_strat** (UnstructureStrategy)
- **omit_if_default** (*bool*)
- **forbid_extra_keys** (*bool*)
- **type_overrides** (*Mapping[type, AttributeOverride]*)
- **unstruct_collection_overrides** (*Mapping[type, Callable]*)
- **prefer_attrib_converters** (*bool*)

Added in version 23.2.0: *unstructure_fallback_factory*

Added in version 23.2.0: *structure_fallback_factory*

omit_if_default

forbid_extra_keys

type_overrides

register_unstructure_hook_factory(*predicate: Predicate*) → Callable[[AnyUnstructureHookFactory], AnyUnstructureHookFactory]

register_unstructure_hook_factory(*predicate: Predicate*, *factory: UnstructureHookFactory*) → UnstructureHookFactory

register_unstructure_hook_factory(*predicate: Predicate*, *factory: ExtendedUnstructureHookFactory[*Converter*]*) → ExtendedUnstructureHookFactory[*Converter*]

Register a hook factory for a given predicate.

The hook factory may expose an additional required parameter. In this case, the current converter will be provided to the hook factory as that parameter.

May also be used as a decorator.

Parameters

- **predicate** – A function that, given a type, returns whether the factory can produce a hook for that type.
- **factory** – A callable that, given a type, produces an unstructuring hook for that type. This unstructuring hook will be cached.

Changed in version 24.1.0: This method may now be used as a decorator. The factory may also receive the converter as a second, required argument.

register_structure_hook_factory(*predicate: Predicate*) → Callable[[AnyStructureHookFactory], AnyStructureHookFactory]

register_structure_hook_factory(*predicate: Predicate*, *factory: StructureHookFactory*) → StructureHookFactory

register_structure_hook_factory(*predicate: Predicate*, *factory: ExtendedStructureHookFactory[Converter]*) → ExtendedStructureHookFactory[*Converter*]

Register a hook factory for a given predicate.

The hook factory may expose an additional required parameter. In this case, the current converter will be provided to the hook factory as that parameter.

May also be used as a decorator.

Parameters

- **predicate** – A function that, given a type, returns whether the factory can produce a hook for that type.
- **factory** – A callable that, given a type, produces a structuring hook for that type. This structuring hook will be cached.

Changed in version 24.1.0: This method may now be used as a decorator. The factory may also receive the converter as a second, required argument.

get_structure_newtype(*type*)

Parameters
type (*type[T]*)

Return type
Callable[[*Any*, *Any*], *T*]

gen_unstructure_annotated(*type*)

gen_structure_annotated(*type*)

A hook factory for annotated types.

Return type
Callable

gen_unstructure_typeddict(*cl*)

Generate a TypedDict unstructure function.

Also apply converter-scored modifications.

Parameters
cl (*Any*)

Return type
Callable[[dict], dict]

gen_unstructure_attrs_fromdict(*cl*)

Parameters
cl (*type[T]*)

Return type
Callable[[*T*], dict[str, *Any*]]

gen_unstructure_optional(*cl*)

Generate an unstructuring hook for optional types.

Parameters
cl (*type[T]*)

Return type
Callable[[*T*], *Any*]

gen_structure_typeddict(*cl*)

Generate a TypedDict structure function.

Also apply converter-scored modifications.

Parameters
cl (*Any*)

Return type
Callable[[dict, *Any*], dict]

gen_structure_attrs_fromdict(*cl*)

Parameters
cl (*type[T]*)

Return type
Callable[[*Mapping*[str, *Any*], *Any*], *T*]

gen_unstructure_iterable(*cl*, *unstructure_to=None*)

Parameters

- **cl** (*Any*)
- **unstructure_to** (*Any*)

Return type
Callable[[*Iterable*[*Any*]], *Any*]

gen_unstructure_hetero_tuple(*cl*, *unstructure_to=None*)

Parameters

- **cl** (*Any*)
- **unstructure_to** (*Any*)

Return type
Callable[[*Tuple*[*Any*, ...]], *Any*]

gen_unstructure_mapping(*cl*, *unstructure_to=None*, *key_handler=None*)

Parameters

- **cl** (*Any*)
- **unstructure_to** (*Any*)
- **key_handler** (*Callable[[Any, Any | None], Any] | None*)

Return type
Callable[[*Mapping*[*Any*, *Any*]], *Any*]

gen_structure_counter(*cl*)

Parameters
cl (*Any*)

Return type
Callable[[*Mapping*[*Any*, *Any*], *Any*], *T*]

gen_structure_mapping(*cl*)

Parameters
cl (*Any*)

Return type
Callable[[*Mapping*[*Any*, *Any*], *Any*], *T*]

copy(*dict_factory=None*, *unstruct_strat=None*, *omit_if_default=None*, *forbid_extra_keys=None*, *type_overrides=None*, *unstruct_collection_overrides=None*, *prefer_attrib_converters=None*, *detailed_validation=None*)

Create a copy of the converter, keeping all existing custom hooks.

Parameters

- **detailed_validation** (*bool | None*) – Whether to use a slightly slower mode for detailed validation errors.
- **dict_factory** (*Callable[[], Any] | None*)
- **unstruct_strat** (*UnstructureStrategy | None*)
- **omit_if_default** (*bool | None*)
- **forbid_extra_keys** (*bool | None*)
- **type_overrides** (*Mapping[type, AttributeOverride] | None*)
- **unstruct_collection_overrides** (*Mapping[type, Callable] | None*)
- **prefer_attrib_converters** (*bool | None*)

Return type
Converter

class cattrs.**AttributeValidationNote**(*string*, *name*, *type*)

Bases: str

Attached as a note to an exception when an attribute fails structuring.

Parameters

- **string** (*str*)
- **name** (*str*)
- **type** (*Any*)

Return type
AttributeValidationNote

name: str

type: Any

exception cattrs.**BaseValidationError**(*message*, *excs*, *cl*)

Bases: ExceptionGroup

Parameters
cl (*Type*)

cl: Type

derive(*excs*)

exception cattrs.**ClassValidationError**(*message*, *excs*, *cl*)

Bases: *BaseValidationError*

Raised when validating a class if any attributes are invalid.

Parameters
cl (*Type*)

group_exceptions()

Split the exceptions into two groups: with and without validation notes.

Return type
Tuple[*List*[*Tuple*[Exception, AttributeValidationNote]], *List*[Exception]]

exception cattrs.**ForbiddenExtraKeysError**(*message*, *cl*, *extra_fields*)

Bases: Exception

Raised when *forbid_extra_keys* is activated and such extra keys are detected during structuring.

The attribute *extra_fields* is a sequence of those extra keys, which were the cause of this error, and *cl* is the class which was structured with those extra keys.

Parameters

- **message** (*str | None*)
- **cl** (*Type*)
- **extra_fields** (*Set[str]*)

Return type
None

cattrs.**GenConverter**

alias of *Converter*

exception cattrs.**IterableValidationError**(*message*, *excs*, *cl*)

Bases: *BaseValidationError*

Raised when structuring an iterable.

Parameters
cl (*Type*)

group_exceptions()

Split the exceptions into two groups: with and without validation notes.

Return type
Tuple[*List*[*Tuple*[Exception, IterableValidationNote]], *List*[Exception]]

class cattrs.**IterableValidationNote**(*string*, *index*, *type*)

Bases: str

Attached as a note to an exception when an iterable element fails structuring.

Parameters

- **string** (*str*)
- **index** (*int | str*)
- **type** (*Any*)

Return type
IterableValidationNote

type: Any

cattrs.**override**(*omit_if_default=None*, *rename=None*, *omit=None*, *struct_hook=None*, *unstruct_hook=None*)

Override how a particular field is handled.

Parameters

- **omit** (*bool | None*) – Whether to skip the field or not. *None* means apply default handling.
- **omit_if_default** (*bool | None*)
- **rename** (*str | None*)
- **struct_hook** (*Callable[[Any, Any], Any] | None*)
- **unstruct_hook** (*Callable[[Any], Any] | None*)

Return type
AttributeOverride

exception cattrs.**StructureHandlerNotFoundError**(*message*, *type_*)

Bases: Exception

Error raised when structuring cannot find a handler for converting inputs into type_.

Parameters

- **message** (*str*)
- **type_** (*Type*)

Return type
None

cattrs.**transform_error**(*exc*, *path='$'*, *format_exception=<function format_exception>*)

Transform an exception into a list of error messages.

To get detailed error messages, the exception should be produced by a converter with *detailed_validation* set.

By default, the error messages are in the form of *{description} @ {path}*.

While traversing the exception and subexceptions, the path is formed:

- by appending *.{field_name}* for fields in classes
- by appending *[{int}]* for indices in iterables, like lists
- by appending *[{str}]* for keys in mappings, like dictionaries

Parameters

- **exc** (ClassValidationError | IterableValidationError | *BaseException*) – The exception to transform into error messages.
- **path** (*str*) – The root path to use.
- **format_exception** (*Callable[[BaseException, type | None], str]*) – A callable to use to transform *Exceptions* into string descriptions of errors.

Return type
List[str]

Added in version 23.1.0.

class cattrs.**UnstructureStrategy**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

attrs classes unstructuring strategies.

AS_DICT = 'asdict'

AS_TUPLE = 'astuple'

12.1.1 Subpackages

cattrs.gen package

cattrs.gen.**make_dict_unstructure_fn**(*cl*, *converter*, *_cattrs_omit_if_default=False*, *_cattrs_use_linecache=True*, *_cattrs_use_alias=False*, *_cattrs_include_init_false=False*, ***kwargs*)

Generate a specialized dict unstructuring function for an attrs class or a dataclass.

Any provided overrides are attached to the generated function under the *overrides* attribute.

Parameters

- **_cattrs_omit_if_default** (*bool*) – if true, attributes equal to their default values will be omitted in the result dictionary.
- **_cattrs_use_alias** (*bool*) – If true, the attribute alias will be used as the dictionary key by default.
- **_cattrs_include_init_false** (*bool*) – If true, _attrs_ fields marked as *init=False* will be included.
- **cl** (*type[T]*)
- **converter** (BaseConverter)
- **_cattrs_use_linecache** (*bool*)
- **kwargs** (*AttributeOverride*)

Return type
Callable[[T], dict[str, Any]]

Added in version 23.2.0: *_cattrs_use_alias*

Added in version 23.2.0: *_cattrs_include_init_false*

cattrs.gen.**make_dict_structure_fn**(*cl*, *converter*, *_cattrs_forbid_extra_keys='from_converter'*, *_cattrs_use_linecache=True*, *_cattrs_prefer_attrib_converters='from_converter'*, *_cattrs_detailed_validation='from_converter'*, *_cattrs_use_alias=False*, *_cattrs_include_init_false=False*, ***kwargs*)

Generate a specialized dict structuring function for an attrs class or dataclass.

Any provided overrides are attached to the generated function under the *overrides* attribute.

Parameters

- **_cattrs_forbid_extra_keys** (*bool | Literal['from_converter']*) – Whether the structuring function should raise a *ForbiddenExtraKeysError* if unknown keys are encountered.
- **_cattrs_use_linecache** (*bool*) – Whether to store the source code in the Python linecache.
- **_cattrs_prefer_attrib_converters** (*bool | Literal['from_converter']*) – If an _attrs_ converter is present on a field, use it instead of processing the field normally.
- **_cattrs_detailed_validation** (*bool | Literal['from_converter']*) – Whether to use a slower mode that produces more detailed errors.
- **_cattrs_use_alias** (*bool*) – If true, the attribute alias will be used as the dictionary key by default.
- **_cattrs_include_init_false** (*bool*) – If true, _attrs_ fields marked as *init=False* will be included.
- **cl** (*type[T]*)
- **converter** (BaseConverter)
- **kwargs** (*AttributeOverride*)

Return type
DictStructureFn[T]

Added in version 23.2.0: *_cattrs_use_alias*

Added in version 23.2.0: *_cattrs_include_init_false*

Changed in version 23.2.0: The *_cattrs_forbid_extra_keys* and *_cattrs_detailed_validation* parameters take their values from the given converter by default.

Changed in version 24.1.0: The *_cattrs_prefer_attrib_converters* parameter takes its value from the given converter by default.

cattrs.gen.**make_iterable_unstructure_fn**(*cl*, *converter*, *unstructure_to=None*)

A hook factory for unstructuring iterables.

Parameters

- **unstructure_to** (*Any*) – Force unstructuring to this type, if provided.
- **cl** (*Any*)
- **converter** (BaseConverter)

Return type
UnstructureHook

cattrs.gen.**make_hetero_tuple_unstructure_fn**(*cl*, *converter*, *unstructure_to=None*, *type_args=None*)

Generate a specialized unstructure function for a heterogenous tuple.

Parameters

- **type_args** (*tuple | None*) – If provided, override the type arguments.
- **cl** (*Any*)
- **converter** (BaseConverter)
- **unstructure_to** (*Any*)

Return type
HeteroTupleUnstructureFn

cattrs.gen.**make_mapping_unstructure_fn**(*cl*, *converter*, *unstructure_to=None*, *key_handler=None*)

Generate a specialized unstructure function for a mapping.

Parameters

- **cl** (*Any*)
- **converter** (BaseConverter)
- **unstructure_to** (*Any*)
- **key_handler** (*Callable[[Any, Any | None], Any] | None*)

Return type
MappingUnstructureFn

cattrs.gen.**make_mapping_structure_fn**(*cl*, *converter*, *structure_to=<class 'dict'>*, *key_type=NOTHING*, *val_type=NOTHING*, *detailed_validation=True*)

Generate a specialized structure function for a mapping.

Parameters

- **cl** (*type[T]*)
- **converter** (BaseConverter)
- **structure_to** (*type*)
- **detailed_validation** (*bool*)

Return type
MappingStructureFn[T]

cattrs.gen.**make_dict_unstructure_fn_from_attrs**(*attrs*, *cl*, *converter*, *typevar_map={}*, *_cattrs_omit_if_default=False*, *_cattrs_use_linecache=True*, *_cattrs_use_alias=False*, *_cattrs_include_init_false=False*, ***kwargs*)

Generate a specialized dict unstructuring function for a list of attributes.

Usually used as a building block by more specialized hook factories.

Any provided overrides are attached to the generated function under the *overrides* attribute.

Parameters

- **cl** (*type*) – The class for which the function is generated; used mostly for its name, module name and qualname.
- **_cattrs_omit_if_default** (*bool*) – if true, attributes equal to their default values will be omitted in the result dictionary.
- **_cattrs_use_alias** (*bool*) – If true, the attribute alias will be used as the dictionary key by default.
- **_cattrs_include_init_false** (*bool*) – If true, _attrs_ fields marked as *init=False* will be included.
- **attrs** (*list[Attribute]*)
- **converter** (BaseConverter)
- **typevar_map** (*dict[str, Any]*)

- **_cattrs_use_linecache** (*bool*)
- **kwargs** (*AttributeOverride*)

Return type
Callable[[T], dict[str, Any]]

Added in version 24.1.0.

cattrs.gen.**make_dict_structure_fn_from_attrs**(*attrs*, *cl*, *converter*, *typevar_map={}*, *_cattrs_forbid_extra_keys='from_converter'*, *_cattrs_use_linecache=True*, *_cattrs_prefer_attrib_converters='from_converter'*, *_cattrs_detailed_validation='from_converter'*, *_cattrs_use_alias=False*, *_cattrs_include_init_false=False*, ***kwargs*)

Generate a specialized dict structuring function for a list of attributes.

Usually used as a building block by more specialized hook factories.

Any provided overrides are attached to the generated function under the *overrides* attribute.

Parameters

- **_cattrs_forbid_extra_keys** (*bool | Literal['from_converter']*) – Whether the structuring function should raise a *ForbiddenExtraKeysError* if unknown keys are encountered.
- **_cattrs_use_linecache** (*bool*) – Whether to store the source code in the Python linecache.
- **_cattrs_prefer_attrib_converters** (*bool | Literal['from_converter']*) – If an _attrs_ converter is present on a field, use it instead of processing the field normally.
- **_cattrs_detailed_validation** (*bool | Literal['from_converter']*) – Whether to use a slower mode that produces more detailed errors.
- **_cattrs_use_alias** (*bool*) – If true, the attribute alias will be used as the dictionary key by default.
- **_cattrs_include_init_false** (*bool*) – If true, _attrs_ fields marked as *init=False* will be included.
- **attrs** (*list[Attribute]*)
- **cl** (*type*)
- **converter** (BaseConverter)
- **typevar_map** (*dict[str, Any]*)
- **kwargs** (*AttributeOverride*)

Return type
DictStructureFn[T]

Added in version 24.1.0.

Submodules

cattrs.gen.typeddicts module

cattrs.gen.typeddicts.**make_dict_unstructure_fn**(*cl*, *converter*, *_cattrs_use_linecache=True*, ***kwargs*)

Generate a specialized dict unstructuring function for a TypedDict.

Parameters

- **cl** (*type[T]*) – A *TypedDict* class.
- **converter** (BaseConverter) – A Converter instance to use for unstructuring nested fields.
- **kwargs** (*AttributeOverride*) – A mapping of field names to an *AttributeOverride*, for customization.
- **_cattrs_detailed_validation** – Whether to store the generated code in the _linecache_, for easier debugging and better stack traces.
- **_cattrs_use_linecache** (*bool*)

Return type
Callable[[T], dict[str, Any]]

cattrs.gen.typeddicts.**make_dict_structure_fn**(*cl*, *converter*, *_cattrs_forbid_extra_keys='from_converter'*, *_cattrs_use_linecache=True*, *_cattrs_detailed_validation='from_converter'*, ***kwargs*)

Generate a specialized dict structuring function for typed dicts.

Parameters

- **cl** (*Any*) – A *TypedDict* class.
- **converter** (BaseConverter) – A Converter instance to use for structuring nested fields.
- **kwargs** (*AttributeOverride*) – A mapping of field names to an *AttributeOverride*, for customization.
- **_cattrs_detailed_validation** (*bool | Literal['from_converter']*) – Whether to use a slower mode that produces more detailed errors.
- **_cattrs_forbid_extra_keys** (*bool | Literal['from_converter']*) – Whether the structuring function should raise a *ForbiddenExtraKeysError* if unknown keys are encountered.
- **_cattrs_detailed_validation** – Whether to store the generated code in the _linecache_, for easier debugging and better stack traces.
- **_cattrs_use_linecache** (*bool*)

Return type
Callable[[dict, Any], Any]

Changed in version 23.2.0: The *_cattrs_forbid_extra_keys* and *_cattrs_detailed_validation* parameters take their values from the given converter by default.

cattrs.preconf package

cattrs.preconf.**validate_datetime**(*v*, *_*)

cattrs.preconf.**wrap**(*_*)

Wrap a *Converter* *__init__* in a type-safe way.

Parameters

_ (*Callable[[~P], Any]*)

Return type

Callable[[*Callable*[[…], *T*]], *Callable*[[~P], *T*]]

Submodules

cattrs.preconf.bson module

cattrs.preconf.cbor2 module

Preconfigured converters for cbor2.

class cattrs.preconf.cbor2.**Cbor2Converter**(*dict_factory=<class 'dict'>*, *unstruct_strat=UnstructureStrategy.AS_DICT*, *omit_if_default=False*, *forbid_extra_keys=False*, *type_overrides={}*, *unstruct_collection_overrides={}*, *prefer_attrib_converters=False*, *detailed_validation=True*, *unstructure_fallback_factory=<function Converter.<lambda>>*, *structure_fallback_factory=<function Converter.<lambda>>*)

Bases: *Converter*

Parameters

- **detailed_validation** (*bool*) – Whether to use a slightly slower mode for detailed validation errors.
- **unstructure_fallback_factory** (*HookFactory[UnstructureHook]*) – A hook factory to be called when no registered unstructuring hooks match.
- **structure_fallback_factory** (*HookFactory[StructureHook]*) – A hook factory to be called when no registered structuring hooks match.
- **dict_factory** (*Callable[[], Any]*)
- **unstruct_strat** (UnstructureStrategy)
- **omit_if_default** (*bool*)
- **forbid_extra_keys** (*bool*)
- **type_overrides** (*Mapping[type, AttributeOverride]*)
- **unstruct_collection_overrides** (*Mapping[type, Callable]*)
- **prefer_attrib_converters** (*bool*)

Added in version 23.2.0: *unstructure_fallback_factory*

Added in version 23.2.0: *structure_fallback_factory*

dumps(*obj*, *unstructure_as=None*, ***kwargs*)

Parameters

- **obj** (*Any*)
- **unstructure_as** (*Any*)
- **kwargs** (*Any*)

Return type
bytes

loads(*data*, *cl*, ***kwargs*)

Parameters

- **data** (*bytes*)
- **cl** (*Type[T]*)
- **kwargs** (*Any*)

Return type
T

omit_if_default

forbid_extra_keys

type_overrides

cattrs.preconf.cbor2.**configure_converter**(*converter*)

Configure the converter for use with the cbor2 library.

- datetimes are serialized as timestamp floats
- sets are serialized as lists

Parameters
converter (BaseConverter)

cattrs.preconf.cbor2.**make_converter**(**args*, ***kwargs*)

Parameters

- **args** (*Any*)
- **kwargs** (*Any*)

Return type
Cbor2Converter

cattrs.preconf.json module

Preconfigured converters for the stdlib json.

class cattrs.preconf.json.**JsonConverter**(*dict_factory=<class 'dict'>*, *unstruct_strat=UnstructureStrategy.AS_DICT*, *omit_if_default=False*, *forbid_extra_keys=False*, *type_overrides={}*, *unstruct_collection_overrides={}*, *prefer_attrib_converters=False*, *detailed_validation=True*, *unstructure_fallback_factory=<function Converter.<lambda>>*, *structure_fallback_factory=<function Converter.<lambda>>*)

Bases: *Converter*

Parameters

- **detailed_validation** (*bool*) – Whether to use a slightly slower mode for detailed validation errors.
- **unstructure_fallback_factory** (*HookFactory[UnstructureHook]*) – A hook factory to be called when no registered unstructuring hooks match.
- **structure_fallback_factory** (*HookFactory[StructureHook]*) – A hook factory to be called when no registered structuring hooks match.
- **dict_factory** (*Callable[[], Any]*)
- **unstruct_strat** (UnstructureStrategy)
- **omit_if_default** (*bool*)
- **forbid_extra_keys** (*bool*)
- **type_overrides** (*Mapping[type, AttributeOverride]*)
- **unstruct_collection_overrides** (*Mapping[type, Callable]*)
- **prefer_attrib_converters** (*bool*)

Added in version 23.2.0: *unstructure_fallback_factory*

Added in version 23.2.0: *structure_fallback_factory*

dumps(*obj*, *unstructure_as=None*, ***kwargs*)

Parameters

- **obj** (*Any*)
- **unstructure_as** (*Any*)
- **kwargs** (*Any*)

Return type
str

loads(*data*, *cl*, ***kwargs*)

Parameters

- **data** (*bytes | str*)
- **cl** (*Type[T]*)

- **kwargs** (*Any*)

Return type
T

omit_if_default

forbid_extra_keys

type_overrides

cattrs.preconf.json.**configure_converter**(*converter*)

Configure the converter for use with the stdlib json module.

- bytes are serialized as base85 strings
- datetimes are serialized as ISO 8601
- counters are serialized as dicts
- sets are serialized as lists
- union passthrough is configured for unions of strings, bools, ints, floats and None

Parameters
converter (BaseConverter)

cattrs.preconf.json.**make_converter**(**args*, ***kwargs*)

Parameters

- **args** (*Any*)
- **kwargs** (*Any*)

Return type
JsonConverter

cattrs.preconf.msgpack module

Preconfigured converters for msgpack.

class cattrs.preconf.msgpack.**MsgpackConverter**(*dict_factory=<class 'dict'>*, *unstruct_strat=UnstructureStrategy.AS_DICT*, *omit_if_default=False*, *forbid_extra_keys=False*, *type_overrides={}*, *unstruct_collection_overrides={}*, *prefer_attrib_converters=False*, *detailed_validation=True*, *unstructure_fallback_factory=<function Converter.<lambda>>*, *structure_fallback_factory=<function Converter.<lambda>>*)

Bases: *Converter*

Parameters

- **detailed_validation** (*bool*) – Whether to use a slightly slower mode for detailed validation errors.

- **unstructure_fallback_factory** (*HookFactory[UnstructureHook]*) – A hook factory to be called when no registered unstructuring hooks match.
- **structure_fallback_factory** (*HookFactory[StructureHook]*) – A hook factory to be called when no registered structuring hooks match.
- **dict_factory** (*Callable[[], Any]*)
- **unstruct_strat** (UnstructureStrategy)
- **omit_if_default** (*bool*)
- **forbid_extra_keys** (*bool*)
- **type_overrides** (*Mapping[type, AttributeOverride]*)
- **unstruct_collection_overrides** (*Mapping[type, Callable]*)
- **prefer_attrib_converters** (*bool*)

Added in version 23.2.0: *unstructure_fallback_factory*

Added in version 23.2.0: *structure_fallback_factory*

dumps(*obj*, *unstructure_as=None*, ***kwargs*)

Parameters

- **obj** (*Any*)
- **unstructure_as** (*Any*)
- **kwargs** (*Any*)

Return type
bytes

loads(*data*, *cl*, ***kwargs*)

Parameters

- **data** (*bytes*)
- **cl** (*Type[T]*)
- **kwargs** (*Any*)

Return type
T

omit_if_default

forbid_extra_keys

type_overrides

cattrs.preconf.msgpack.**configure_converter**(*converter*)

Configure the converter for use with the msgpack library.

- datetimes are serialized as timestamp floats
- sets are serialized as lists

Parameters
converter (BaseConverter)

cattrs.preconf.msgpack.**make_converter**(**args*, ***kwargs*)

Parameters

- **args** (*Any*)
- **kwargs** (*Any*)

Return type
MsgpackConverter

cattrs.preconf.msgspec module

cattrs.preconf.orjson module

Preconfigured converters for orjson.

class cattrs.preconf.orjson.**OrjsonConverter**(*dict_factory=<class 'dict'>*, *unstruct_strat=UnstructureStrategy.AS_DICT*, *omit_if_default=False*, *forbid_extra_keys=False*, *type_overrides={}*, *unstruct_collection_overrides={}*, *prefer_attrib_converters=False*, *detailed_validation=True*, *unstructure_fallback_factory=<function Converter.<lambda>>*, *structure_fallback_factory=<function Converter.<lambda>>*)

Bases: *Converter*

Parameters

- **detailed_validation** (*bool*) – Whether to use a slightly slower mode for detailed validation errors.
- **unstructure_fallback_factory** (*HookFactory[UnstructureHook]*) – A hook factory to be called when no registered unstructuring hooks match.
- **structure_fallback_factory** (*HookFactory[StructureHook]*) – A hook factory to be called when no registered structuring hooks match.
- **dict_factory** (*Callable[[], Any]*)
- **unstruct_strat** (UnstructureStrategy)
- **omit_if_default** (*bool*)
- **forbid_extra_keys** (*bool*)
- **type_overrides** (*Mapping[type, AttributeOverride]*)
- **unstruct_collection_overrides** (*Mapping[type, Callable]*)
- **prefer_attrib_converters** (*bool*)

Added in version 23.2.0: *unstructure_fallback_factory*

Added in version 23.2.0: *structure_fallback_factory*

dumps(*obj*, *unstructure_as=None*, ***kwargs*)

Parameters

- **obj** (*Any*)
- **unstructure_as** (*Any*)
- **kwargs** (*Any*)

Return type
bytes

loads(*data*, *cl*)

Parameters

- **data** (*bytes | bytearray | memoryview | str*)
- **cl** (*Type[T]*)

Return type
T

omit_if_default

forbid_extra_keys

type_overrides

cattrs.preconf.orjson.**configure_converter**(*converter*)

Configure the converter for use with the orjson library.

- bytes are serialized as base85 strings
- datetimes and dates are passed through to be serialized as RFC 3339 by orjson
- typed namedtuples are serialized as lists
- sets are serialized as lists
- string enum mapping keys have special handling
- mapping keys are coerced into strings when unstructuring

Parameters
converter (BaseConverter)

cattrs.preconf.orjson.**make_converter**(**args*, ***kwargs*)

Parameters

- **args** (*Any*)
- **kwargs** (*Any*)

Return type
OrjsonConverter

cattrs.preconf.pyyaml module

Preconfigured converters for pyyaml.

cattrs.preconf.pyyaml.**validate_date**(*v*, *_*)

class cattrs.preconf.pyyaml.**PyyamlConverter**(*dict_factory=<class 'dict'>*, *unstruct_strat=UnstructureStrategy.AS_DICT*, *omit_if_default=False*, *forbid_extra_keys=False*, *type_overrides={}*, *unstruct_collection_overrides={}*, *prefer_attrib_converters=False*, *detailed_validation=True*, *unstructure_fallback_factory=<function Converter.<lambda>>*, *structure_fallback_factory=<function Converter.<lambda>>*)

Bases: *Converter*

Parameters

- **detailed_validation** (*bool*) – Whether to use a slightly slower mode for detailed validation errors.
- **unstructure_fallback_factory** (*HookFactory[UnstructureHook]*) – A hook factory to be called when no registered unstructuring hooks match.
- **structure_fallback_factory** (*HookFactory[StructureHook]*) – A hook factory to be called when no registered structuring hooks match.
- **dict_factory** (*Callable[[], Any]*)
- **unstruct_strat** (UnstructureStrategy)
- **omit_if_default** (*bool*)
- **forbid_extra_keys** (*bool*)
- **type_overrides** (*Mapping[type, AttributeOverride]*)
- **unstruct_collection_overrides** (*Mapping[type, Callable]*)
- **prefer_attrib_converters** (*bool*)

Added in version 23.2.0: *unstructure_fallback_factory*

Added in version 23.2.0: *structure_fallback_factory*

dumps(*obj*, *unstructure_as=None*, ***kwargs*)

Parameters

- **obj** (*Any*)
- **unstructure_as** (*Any*)
- **kwargs** (*Any*)

Return type
str

loads(*data*, *cl*)

Parameters

- **data** (*str*)

- **cl** (*Type[T]*)

Return type
T

omit_if_default

forbid_extra_keys

type_overrides

cattrs.preconf.pyyaml.**configure_converter**(*converter*)

Configure the converter for use with the pyyaml library.

- frozensets are serialized as lists
- string enums are converted into strings explicitly
- datetimes and dates are validated
- typed namedtuples are serialized as lists

Parameters
converter (BaseConverter)

cattrs.preconf.pyyaml.**make_converter**(**args*, ***kwargs*)

Parameters

- **args** (*Any*)
- **kwargs** (*Any*)

Return type
PyyamlConverter

cattrs.preconf.tomlkit module

Preconfigured converters for tomlkit.

class cattrs.preconf.tomlkit.**TomlkitConverter**(*dict_factory=<class 'dict'>*, *unstruct_strat=UnstructureStrategy.AS_DICT*, *omit_if_default=False*, *forbid_extra_keys=False*, *type_overrides={}*, *unstruct_collection_overrides={}*, *prefer_attrib_converters=False*, *detailed_validation=True*, *unstructure_fallback_factory=<function Converter.<lambda>>*, *structure_fallback_factory=<function Converter.<lambda>>*)

Bases: *Converter*

Parameters

- **detailed_validation** (*bool*) – Whether to use a slightly slower mode for detailed validation errors.
- **unstructure_fallback_factory** (*HookFactory[UnstructureHook]*) – A hook factory to be called when no registered unstructuring hooks match.

- **structure_fallback_factory** (*HookFactory[StructureHook]*) – A hook factory to be called when no registered structuring hooks match.
- **dict_factory** (*Callable[[], Any]*)
- **unstruct_strat** (UnstructureStrategy)
- **omit_if_default** (*bool*)
- **forbid_extra_keys** (*bool*)
- **type_overrides** (*Mapping[type, AttributeOverride]*)
- **unstruct_collection_overrides** (*Mapping[type, Callable]*)
- **prefer_attrib_converters** (*bool*)

Added in version 23.2.0: *unstructure_fallback_factory*

Added in version 23.2.0: *structure_fallback_factory*

dumps(*obj*, *unstructure_as=None*, ***kwargs*)

Parameters

- **obj** (*Any*)
- **unstructure_as** (*Any*)
- **kwargs** (*Any*)

Return type
str

loads(*data*, *cl*)

Parameters

- **data** (*str*)
- **cl** (*Type[T]*)

Return type
T

omit_if_default

forbid_extra_keys

type_overrides

cattrs.preconf.tomlkit.**configure_converter**(*converter*)

Configure the converter for use with the tomlkit library.

- bytes are serialized as base85 strings
- sets are serialized as lists
- tuples are serializas as lists
- mapping keys are coerced into strings when unstructuring

Parameters
converter (BaseConverter)

cattrs.preconf.tomlkit.**make_converter**(**args*, ***kwargs*)

Parameters

- **args** (*Any*)
- **kwargs** (*Any*)

Return type
TomlkitConverter

cattrs.preconf.ujson module

Preconfigured converters for ujson.

class cattrs.preconf.ujson.**UjsonConverter**(*dict_factory=<class 'dict'>*, *unstruct_strat=UnstructureStrategy.AS_DICT*, *omit_if_default=False*, *forbid_extra_keys=False*, *type_overrides={}*, *unstruct_collection_overrides={}*, *prefer_attrib_converters=False*, *detailed_validation=True*, *unstructure_fallback_factory=<function Converter.<lambda>>*, *structure_fallback_factory=<function Converter.<lambda>>*)

Bases: *Converter*

Parameters

- **detailed_validation** (*bool*) – Whether to use a slightly slower mode for detailed validation errors.
- **unstructure_fallback_factory** (*HookFactory[UnstructureHook]*) – A hook factory to be called when no registered unstructuring hooks match.
- **structure_fallback_factory** (*HookFactory[StructureHook]*) – A hook factory to be called when no registered structuring hooks match.
- **dict_factory** (*Callable[[], Any]*)
- **unstruct_strat** (UnstructureStrategy)
- **omit_if_default** (*bool*)
- **forbid_extra_keys** (*bool*)
- **type_overrides** (*Mapping[type, AttributeOverride]*)
- **unstruct_collection_overrides** (*Mapping[type, Callable]*)
- **prefer_attrib_converters** (*bool*)

Added in version 23.2.0: *unstructure_fallback_factory*

Added in version 23.2.0: *structure_fallback_factory*

dumps(*obj*, *unstructure_as=None*, ***kwargs*)

Parameters

- **obj** (*Any*)
- **unstructure_as** (*Any*)

- **kwargs** (*Any*)

Return type
str

loads(*data*, *cl*, ***kwargs*)

Parameters

- **data** (*AnyStr*)
- **cl** (*Type[T]*)
- **kwargs** (*Any*)

Return type
T

omit_if_default

forbid_extra_keys

type_overrides

cattrs.preconf.ujson.**configure_converter**(*converter*)

Configure the converter for use with the ujson library.

- bytes are serialized as base64 strings
- datetimes are serialized as ISO 8601
- sets are serialized as lists

Parameters
converter (BaseConverter)

cattrs.preconf.ujson.**make_converter**(**args*, ***kwargs*)

Parameters

- **args** (*Any*)
- **kwargs** (*Any*)

Return type
UjsonConverter

cattrs.strategies package

High level strategies for converters.

cattrs.strategies.**configure_tagged_union**(*union*, *converter*, *tag_generator=<function default_tag_generator>*, *tag_name='_type'*, *default=NOTHING*)

Configure the converter so that *union* (which should be a union) is un/structured with the help of an additional piece of data in the unstructured payload, the tag.

Parameters

- **converter** (BaseConverter) – The converter to apply the strategy to.

- **tag_generator** (*Callable[[Type], str]*) – A *tag_generator* function is used to map each member of the union to a tag, which is then included in the unstructured payload. The default tag generator returns the name of the class.
- **tag_name** (*str*) – The key under which the tag will be set in the unstructured payload. By default, *'_type'*.
- **default** (*Type | Literal[_Nothing.NOTHING]*) – An optional class to be used if the tag information is not present when structuring.
- **union** (*Any*)

Return type
None

The tagged union strategy currently only works with the dict un/structuring base strategy.

Added in version 23.1.0.

cattrs.strategies.**configure_union_passthrough**(*union*, *converter*)

Configure the converter to support validating and passing through unions of the provided types and their subsets.

For example, all mature JSON libraries natively support producing unions of ints, floats, Nones, and strings. Using this strategy, a converter can be configured to efficiently validate and pass through unions containing these types.

The most important point is that another library (in this example the JSON library) handles producing the union, and the converter is configured to just validate it.

Literals of provided types are also supported, and are checked by value.

NewTypes of provided types are also supported.

The strategy is designed to be O(1) in execution time, and independent of the ordering of types in the union.

If the union contains a class and one or more of its subclasses, the subclasses will also be included when validating the superclass.

Added in version 23.2.0.

Parameters

- **union** (*Any*)
- **converter** (BaseConverter)

Return type
None

cattrs.strategies.**include_subclasses**(*cl*, *converter*, *subclasses=None*, *union_strategy=None*, *overrides=None*)

Configure the converter so that the attrs/dataclass *cl* is un/structured as if it was a union of itself and all its subclasses that are defined at the time when this strategy is applied.

Parameters

- **cl** (*type*) – A base *attrs* or *dataclass* class.
- **converter** (*C*) – The *Converter* on which this strategy is applied. Do note that the strategy does not work for a *cattrs.BaseConverter*.
- **subclasses** (*tuple[type, ...] | None*) – A tuple of sublcasses whose ancestor is *cl*. If left as *None*, subclasses are detected using recursively the *__subclasses__* method of *cl* and its descendents.

- **union_strategy** (*Callable[[Any, C], Any] | None*) – A callable of two arguments passed by position (*subclass_union*, *converter*) that defines the union strategy to use to disambiguate the subclasses union. If *None* (the default), the automatic unique field disambiguation is used which means that every single subclass participating in the union must have an attribute name that does not exist in any other sibling class.
- **overrides** (*dict[str, AttributeOverride] | None*) – a mapping of *cl* attribute names to overrides (instantiated with `cattrs.gen.override()`) to customize un/structuring.

Return type
None

Added in version 23.1.0.

Changed in version 24.1.0: When overrides are not provided, hooks for individual classes are retrieved from the converter instead of generated with no overrides, using converter defaults.

`cattrs.strategies.`**use_class_methods**(*converter*, *structure_method_name=None*, *unstructure_method_name=None*)

Configure the converter such that dedicated methods are used for (un)structuring the instance of a class if such methods are available. The default (un)structuring will be applied if such an (un)structuring methods cannot be found.

Parameters

- **converter** (BaseConverter) – The *Converter* on which this strategy is applied. You can use *cattrs.BaseConverter* or any other derived class.
- **structure_method_name** (*str | None*) – Optional string with the name of the class method which should be used for structuring. If not provided, no class method will be used for structuring.
- **unstructure_method_name** (*str | None*) – Optional string with the name of the class method which should be used for unstructuring. If not provided, no class method will be used for unstructuring.

Return type
None

If you want to (un)structured nested objects, just append a converter parameter to your (un)structuring methods and you will receive the converter there.

Added in version 23.2.0.

12.1.2 Submodules

12.1.3 cattrs.cols module

Utility functions for collections.

`cattrs.cols.`**is_any_set**(*type*)

A predicate function for both mutable and frozensets.

Return type
bool

cattrs.cols.**is_frozenset**(*type*)

A predicate function for frozensets.

Matches built-in frozensets and frozensets from the typing module.

Parameters
type (*Any*)

Return type
bool

cattrs.cols.**is_namedtuple**(*type*)

A predicate function for named tuples.

Parameters
type (*Any*)

Return type
bool

cattrs.cols.**is_mapping**(*type*)

A predicate function for mappings.

Parameters
type (*Any*)

Return type
bool

cattrs.cols.**is_set**(*type*)

A predicate function for (mutable) sets.

Matches built-in sets and sets from the typing module.

Parameters
type (*Any*)

Return type
bool

cattrs.cols.**is_sequence**(*type*)

A predicate function for sequences.

Matches lists, sequences, mutable sequences, deques and homogenous tuples.

Parameters
type (*Any*)

Return type
bool

cattrs.cols.**iterable_unstructure_factory**(*cl*, *converter*, *unstructure_to=None*)

A hook factory for unstructuring iterables.

Parameters

- **unstructure_to** (*Any*) – Force unstructuring to this type, if provided.
- **cl** (*Any*)
- **converter** (BaseConverter)

Return type
UnstructureHook

cattrs.cols.**list_structure_factory**(*type*, *converter*)

A hook factory for structuring lists.

Converts any given iterable into a list.

Parameters

- **type** (*type*)
- **converter** (BaseConverter)

Return type
StructureHook

cattrs.cols.**namedtuple_structure_factory**(*cl*, *converter*)

A hook factory for structuring namedtuples from iterables.

Parameters

- **cl** (*type[tuple]*)
- **converter** (BaseConverter)

Return type
StructureHook

cattrs.cols.**namedtuple_unstructure_factory**(*cl*, *converter*, *unstructure_to=None*)

A hook factory for unstructuring namedtuples.

Parameters

- **unstructure_to** (*Any*) – Force unstructuring to this type, if provided.
- **cl** (*type[tuple]*)
- **converter** (BaseConverter)

Return type
UnstructureHook

cattrs.cols.**namedtuple_dict_structure_factory**(*cl*, *converter*, *detailed_validation='from_converter'*, *forbid_extra_keys=False*, *use_linecache=True*, */*, ***kwargs*)

A hook factory for hooks structuring namedtuples from dictionaries.

Parameters

- **forbid_extra_keys** (*bool*) – Whether the hook should raise a *ForbiddenExtraKeysError* if unknown keys are encountered.
- **use_linecache** (*bool*) – Whether to store the source code in the Python linecache.
- **cl** (*type[tuple]*)
- **converter** (BaseConverter)
- **detailed_validation** (*bool | Literal['from_converter']*)
- **kwargs** (*AttributeOverride*)

Return type
StructureHook

Added in version 24.1.0.

cattrs.cols.**namedtuple_dict_unstructure_factory**(*cl*, *converter*, *omit_if_default=False*, *use_linecache=True*, */*, ***kwargs*)

A hook factory for hooks unstructuring namedtuples to dictionaries.

Parameters

- **omit_if_default** (*bool*) – When true, attributes equal to their default values will be omitted in the result dictionary.
- **use_linecache** (*bool*) – Whether to store the source code in the Python linecache.
- **cl** (*type[tuple]*)
- **converter** (BaseConverter)
- **kwargs** (*AttributeOverride*)

Return type
UnstructureHook

Added in version 24.1.0.

cattrs.cols.**mapping_structure_factory**(*cl*, *converter*, *structure_to=<class 'dict'>*, *key_type=NOTHING*, *val_type=NOTHING*, *detailed_validation=True*)

Generate a specialized structure function for a mapping.

Parameters

- **cl** (*type[T]*)
- **converter** (BaseConverter)
- **structure_to** (*type*)
- **detailed_validation** (*bool*)

Return type
MappingStructureFn[T]

12.1.4 cattrs.disambiguators module

Utilities for union (sum type) disambiguation.

cattrs.disambiguators.**is_supported_union**(*typ*)

Whether the type is a union of attrs classes.

Parameters
typ (*Any*)

Return type
bool

cattrs.disambiguators.**create_default_dis_func**(*converter*, **classes*, *use_literals=True*, *overrides='from_converter'*)

Given attrs classes or dataclasses, generate a disambiguation function.

The function is based on unique fields without defaults or unique values.

Parameters

- **use_literals** (*bool*) – Whether to try using fields annotated as literals for disambiguation.

- **overrides** (*dict[str, AttributeOverride] | Literal['from_converter']*) – Attribute overrides to apply.
- **converter** (BaseConverter)
- **classes** (*type[AttrsInstance]*)

Return type
Callable[[Mapping[Any, Any]], type[Any] | None]

Changed in version 24.1.0: Dataclasses are now supported.

12.1.5 cattrs.dispatch module

class cattrs.dispatch.**FunctionDispatch**(*converter*, *handler_pairs=NOTHING*)

Bases: object

FunctionDispatch is similar to functools.singledispatch, but instead dispatches based on functions that take the type of the first argument in the method, and return True or False.

objects that help determine dispatch should be instantiated objects.

Parameters

- **converter** (BaseConverter) – A converter to be used for factories that require converters.
- **handler_pairs** (*list[tuple[Predicate, Callable[[Any, Any], Any], bool, bool]]*)

Changed in version 24.1.0: Support for factories that require converters, hence this requires a converter when creating.

Method generated by attrs for class FunctionDispatch.

register(*predicate*, *func*, *is_generator=False*, *takes_converter=False*)

Parameters

- **predicate** (*Callable[[Any], bool]*)
- **func** (*Callable[[...], Any]*)

Return type
None

dispatch(*typ*)

Return the appropriate handler for the object passed.

Parameters
typ (*Any*)

Return type
Callable[[…], *Any*] | None

get_num_fns()

Return type
int

copy_to(*other*, *skip=0*)

Parameters

- **other** (FunctionDispatch)
- **skip** (*int*)

Return type
None

class cattrs.dispatch.**MultiStrategyDispatch**(*fallback_factory*, *converter*)

Bases: Generic[Hook]

MultiStrategyDispatch uses a combination of exact-match dispatch, singledispatch, and FunctionDispatch.

Parameters

- **converter** (BaseConverter) – A converter to be used for factories that require converters.
- **fallback_factory** (*HookFactory[Hook]*) – A hook factory to be called when a hook cannot be produced.

Changed in version 23.2.0: Fallbacks are now factories.

Changed in version 24.1.0: Support for factories that require converters, hence this requires a converter when creating.

dispatch: Callable[[TargetType, *BaseConverter*], Hook]

dispatch_without_caching(*typ*)

Dispatch on the type but without caching the result.

Parameters
typ (*Any*)

Return type
Hook

register_cls_list(*cls_and_handler*, *direct=False*)

Register a class to direct or singledispatch.

Parameters
direct (*bool*)

Return type
None

register_func_list(*pred_and_handler*)

Register a predicate function to determine if the handler should be used for the type.

Parameters
pred_and_handler (*list[tuple[Predicate, Any] | tuple[Predicate, Any, bool] | tuple[Predicate, Callable[[Any, BaseConverter], Any], Literal['extended']]]*) – The list of predicates and their associated handlers. If a handler is registered in *extended* mode, it's a factory that requires a converter.

clear_direct()

Clear the direct dispatch.

Return type
None

clear_cache()

Clear all caches.

Return type
None

get_num_fns()

Return type
int

copy_to(*other*, *skip=0*)

Parameters

- **other** (MultiStrategyDispatch)
- **skip** (*int*)

Return type
None

12.1.6 cattrs.errors module

exception cattrs.errors.**StructureHandlerNotFoundError**(*message*, *type_*)

Bases: Exception

Error raised when structuring cannot find a handler for converting inputs into `type_`.

Parameters

- **message** (*str*)
- **type_** (*Type*)

Return type
None

exception cattrs.errors.**BaseValidationError**(*message*, *excs*, *cl*)

Bases: ExceptionGroup

Parameters
cl (*Type*)

cl: Type

derive(*excs*)

class cattrs.errors.**IterableValidationNote**(*string*, *index*, *type*)

Bases: str

Attached as a note to an exception when an iterable element fails structuring.

Parameters

- **string** (*str*)
- **index** (*int | str*)
- **type** (*Any*)

Return type
IterableValidationNote

type: Any

exception cattrs.errors.**IterableValidationError**(*message*, *excs*, *cl*)

Bases: *BaseValidationError*

Raised when structuring an iterable.

Parameters
cl (*Type*)

group_exceptions()

Split the exceptions into two groups: with and without validation notes.

Return type
Tuple[*List*[*Tuple*[Exception, IterableValidationNote]], *List*[Exception]]

class cattrs.errors.**AttributeValidationNote**(*string*, *name*, *type*)

Bases: str

Attached as a note to an exception when an attribute fails structuring.

Parameters

- **string** (*str*)
- **name** (*str*)
- **type** (*Any*)

Return type
AttributeValidationNote

name: str

type: Any

exception cattrs.errors.**ClassValidationError**(*message*, *excs*, *cl*)

Bases: *BaseValidationError*

Raised when validating a class if any attributes are invalid.

Parameters
cl (*Type*)

group_exceptions()

Split the exceptions into two groups: with and without validation notes.

Return type
Tuple[*List*[*Tuple*[Exception, AttributeValidationNote]], *List*[Exception]]

exception cattrs.errors.**ForbiddenExtraKeysError**(*message*, *cl*, *extra_fields*)

Bases: Exception

Raised when *forbid_extra_keys* is activated and such extra keys are detected during structuring.

The attribute *extra_fields* is a sequence of those extra keys, which were the cause of this error, and *cl* is the class which was structured with those extra keys.

Parameters

- **message** (*str | None*)
- **cl** (*Type*)
- **extra_fields** (*Set[str]*)

Return type
None

12.1.7 cattrs.fns module

Useful internal functions.

cattrs.fns.**Predicate**

A predicate function determines if a type can be handled.

alias of `Callable`[[`Any`], `bool`]

cattrs.fns.**identity**(*obj*)

The identity function.

Parameters
obj (*T*)

Return type
T

cattrs.fns.**raise_error**(*_*, *cl*)

At the bottom of the condition stack, we explode if we can't handle it.

Parameters
cl (*Type*)

Return type
NoReturn

12.1.8 cattrs.v module

Cattrs validation.

cattrs.v.**format_exception**(*exc*, *type*)

The default exception formatter, handling the most common exceptions.

The following exceptions are handled specially:

- *KeyErrors* (*required field missing*)
- *ValueErrors* (*invalid value for type, expected <type>* or just *invalid value*)
- *TypeErrors* (*invalid value for type, expected <type>* and a couple special cases for iterables)
- *cattrs.ForbiddenExtraKeysError*
- some *AttributeErrors* (special cased for structing mappings)

Parameters

- **exc** (*BaseException*)
- **type** (*type | None*)

Return type
str

cattrs.v.**transform_error**(*exc*, *path='$'*, *format_exception=<function format_exception>*)

Transform an exception into a list of error messages.

To get detailed error messages, the exception should be produced by a converter with *detailed_validation* set.

By default, the error messages are in the form of *{description} @ {path}*.

While traversing the exception and subexceptions, the path is formed:

- by appending *.{field_name}* for fields in classes
- by appending *[{int}]* for indices in iterables, like lists
- by appending *[{str}]* for keys in mappings, like dictionaries

Parameters

- **exc** (*ClassValidationError | IterableValidationError | BaseException*) – The exception to transform into error messages.
- **path** (*str*) – The root path to use.
- **format_exception** (*Callable[[BaseException, type | None], str]*) – A callable to use to transform *Exceptions* into string descriptions of errors.

Return type
List[str]

Added in version 23.1.0.

CHAPTER

THIRTEEN

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

13.1 Types of Contributions

13.1.1 Report Bugs

Report bugs at https://github.com/python-attrs/cattrs/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

13.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

13.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

13.1.4 Write Documentation

cattrs could always use more documentation, whether as part of the official cattrs docs, in docstrings, or even on the web in blog posts, articles, and such.

13.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/python-attrs/cattrs/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

13.2 Get Started!

Ready to contribute? Here's how to set up *cattrs* for local development.

1. Fork the `cattrs` repo on GitHub.
2. Clone your fork locally::

```
$ git clone git@github.com:your_name_here/cattrs.git
```

3. Install your local copy into a virtualenv. Assuming you have PDM installed, this is how you set up your fork for local development::

```
$ cd cattrs/
$ pdm install -d -G :all
```

4. Create a branch for local development::

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox::

```
$ make lint
$ make test
$ tox
```

6. Commit your changes and push your branch to GitHub::

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

13.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for all supported Python versions. Check https://github.com/python-attrs/cattrs/actions and make sure that the tests pass for all supported Python versions.
4. Don't forget to add a line to HISTORY.md.

13.4 Tips

To run a subset of tests:

```
$ pdm run pytest tests.test_unstructure
```

CHAPTER

FOURTEEN

BENCHMARKING

cattrs includes a benchmarking suite to help detect performance regressions and guide performance optimizations.

The suite is based on pytest and pytest-benchmark. Benchmarks are similar to tests, with the exception of being stored in the `bench/` directory and being used to verify performance instead of correctness.

14.1 A Sample Workflow

First, ensure the system you're benchmarking on is as stable as possible. For example, the pyperf library has a `system tune` command that can tweak CPU frequency governors. You also might want to quit as many applications as possible and run the benchmark suite on isolated CPU cores (`taskset` can be used for this purpose on Linux).

Then, generate a baseline using `make bench`. This will run the benchmark suite and save it into a file.

Following that, implement the changes you have in mind. Run the test suite to ensure correctness. Then, compare the performance of the new code to the saved baseline using `make bench-cmp`. If the code is still correct but faster, congratulations!

CHAPTER

FIFTEEN

HISTORY

This project adheres to Calendar Versioning. The first number of the version is the year. The second number is incremented with each release, starting at 1 for each year. The third number is for emergencies when we need to start branches for older releases.

Our backwards-compatibility policy can be found here.

15.1 24.1.2 (2024-09-22)

- Fix *BaseConverter.register_structure_hook()* and *BaseConverter.register_unstructure_hook()* type hints. (#581 #582)

15.2 24.1.1 (2024-09-11)

- Fix *BaseConverter.register_structure_hook_factory()* and *BaseConverter.register_unstructure_hook_factory()* type hints. (#578 #579)

15.3 24.1.0 (2024-08-28)

- **Potentially breaking**: Unstructuring hooks for `typing.Any` are consistent now: values are unstructured using their runtime type. Previously this behavior was underspecified and inconsistent, but followed this rule in the majority of cases. Reverting old behavior is very dependent on the actual case; ask on the issue tracker if in doubt. (#473)
- **Minor change**: Heterogeneous tuples are now unstructured into tuples instead of lists by default; this is significantly faster and widely supported by serialization libraries. (#486)
- **Minor change**: *cattrs.gen.make_dict_structure_fn()* will use the value for the `prefer_attrib_converters` parameter from the given converter by default now. If you're using this function directly, the old behavior can be restored by passing in the desired values explicitly. (#527 #528)
- Introduce *BaseConverter.get_structure_hook()* and *BaseConverter.get_unstructure_hook()* methods. (#432 #472)
- *BaseConverter.register_structure_hook()*, *BaseConverter.register_unstructure_hook()*, *BaseConverter.register_unstructure_hook_factory()* and *BaseConverter.register_structure_hook_factory()* can now be used as decorators and have gained new features. See here and here for more details. (#487)
- Introduce and document the *cattrs.cols* module for better collection customizations. (#504 #540)

- Enhance the *cattrs.cols.is_mapping()* predicate function to also cover virtual subclasses of `abc.Mapping`. This enables map classes from libraries such as *immutables* or *sortedcontainers* to structure out-of-the-box. (#555 #556)
- Introduce the *msgspec* `preconf converter`. Only JSON is supported for now, with other formats supported by *msgspec* to come later. (#481)
- The default union handler now properly takes renamed fields into account. (#472)
- The default union handler now also handles dataclasses. (#426 #477)
- Add support for PEP 695 type aliases. (#452)
- Add support for PEP 696 `TypeVar`s with defaults. (#512)
- Add support for named tuples with type metadata (`typing.NamedTuple`). (#425 #491)
- Add support for optionally un/unstructuring named tuples using dictionaries. (#425 #549)
- The `include_subclasses` strategy now fetches the member hooks from the converter (making use of converter defaults) if overrides are not provided, instead of generating new hooks with no overrides. (#429 #472)
- The preconf `make_converter` factories are now correctly typed. (#481)
- The *orjson preconf converter* now passes through dates and datetimes to orjson while unstructuring, greatly improving speed. (#463)
- *cattrs.gen* generators now attach metadata to the generated functions, making them introspectable. (#472)
- Structure hook factories in *cattrs.gen* now handle recursive classes better. (#540)
- The tagged union strategy now leaves the tags in the payload unless `forbid_extra_keys` is set. (#533 #534)
- More robust support for `Annotated` and `NotRequired` in TypedDicts. (#450)
- `typing_extensions.Literal` is now automatically structured, just like `typing.Literal`. (#460 #467)
- `typing_extensions.Any` is now supported and handled like `typing.Any`. (#488 #490)
- `Optional` types can now be consistently customized using `register_structure_hook` and `register_unstructure_hook`. (#529 #530)
- The BaseConverter now properly generates detailed validation errors for mappings. (#496)
- PEP 695 generics are now tested. (#452)
- Imports are now sorted using Ruff.
- Tests are run with the pytest-xdist plugin by default.
- Rework the introductory parts of the documentation, introducing the Basics section. (#472)
- The documentation has been significantly reworked. (#473)
- The docs now use the Inter font.
- Make type annotations for `include_subclasses` and `tagged_union` strategies more lenient. (#431)

15.4 23.2.3 (2023-11-30)

- Fix a regression when unstructuring dictionary values typed as `Any`. (#453 #462)
- Fix a regression when unstructuring unspecialized generic classes. (#465 #466)
- Optimize function source code caching. (#445 #464)
- Generate unique files only in case of linecache enabled. (#445 #441)

15.5 23.2.2 (2023-11-21)

- Fix a regression when unstructuring `Any | None`. (#453 #454)

15.6 23.2.1 (2023-11-18)

- Fix unnecessary `typing_extensions` import on Python 3.11. (#446 #447)

15.7 23.2.0 (2023-11-17)

- **Potentially breaking**: skip *attrs* fields marked as `init=False` by default. This change is potentially breaking for unstructuring. See here for instructions on how to restore the old behavior. (#40 #395)
- **Potentially breaking**: *cattrs.gen.make_dict_structure_fn()* and *cattrs.gen.typeddicts.make_dict_structure_fn()* will use the values for the `detailed_validation` and `forbid_extra_keys` parameters from the given converter by default now. If you're using these functions directly, the old behavior can be restored by passing in the desired values directly. (#410 #411)
- **Potentially breaking**: The default union structuring strategy will also use fields annotated as `typing.Literal` to help guide structuring. See here for instructions on how to restore the old behavior. (#391)
- Python 3.12 is now supported. Python 3.7 is no longer supported; use older releases there. (#424)
- Implement the `union passthrough` strategy, enabling much richer union handling for preconfigured converters. Learn more here.
- Introduce the `use_class_methods` strategy. Learn more here. (#405)
- The `omit` parameter of *cattrs.override()* is now of type `bool | None` (from `bool`). `None` is the new default and means to apply default *cattrs* handling to the attribute, which is to omit the attribute if it's marked as `init=False`, and keep it otherwise.
- Converters can now be initialized with custom fallback hook factories for un/structuring. (#331 #441)
- Add support for `date` to preconfigured converters. (#420)
- Add support for `datetime.date`s to the PyYAML preconfigured converter. (#393)
- Fix *format_exception()* parameter working for recursive calls to *transform_error*. (#389)
- *attrs* aliases are now supported, although aliased fields still map to their attribute name instead of their alias by default when un/structuring. (#322 #391)
- Fix TypedDicts with periods in their field names. (#376 #377)
- Optimize and improve unstructuring of `Optional` (unions of one type and `None`). (#380 #381)

- Fix *format_exception* and *transform_error* type annotations.
- Improve the implementation of `cattrs._compat.is_typeddict`. The implementation is now simpler, and relies on fewer private implementation details from `typing` and typing_extensions. (#384)
- Improve handling of TypedDicts with forward references.
- Speed up generated *attrs* and TypedDict structuring functions by changing their signature slightly. (#388)
- Fix copying of converters with function hooks. (#398 #399)
- Broaden *loads'* type definition for the preconf orjson converter. (#400)
- *AttributeValidationNote* and *IterableValidationNote* are now picklable. (#408)
- Fix structuring `Final` lists. (#412)
- Fix certain cases of structuring `Annotated` types. (#418)
- Fix the tagged union strategy to work with `forbid_extra_keys`. (#402 #443)
- Use PDM instead of Poetry.
- *cattrs* is now linted with Ruff.
- Remove some unused lines in the unstructuring code. (#416)
- Fix handling classes inheriting from non-generic protocols. (#374 #436)
- The documentation Makefile now supports the `htmlview` and `htmllive` targets. (#442)
- *cattrs* is now published using PyPI Trusted Publishers, and `main` branch commits are automatically deployed to Test PyPI.

15.8 23.1.2 (2023-06-02)

- Improve `typing_extensions` version bound. (#372)

15.9 23.1.1 (2023-05-30)

- Add `typing_extensions` as a direct dependency on 3.10. (#369 #370)

15.10 23.1.0 (2023-05-30)

- Introduce the `tagged_union` strategy. (#318 #317)
- Introduce the `cattrs.transform_error` helper function for formatting validation exceptions. (258 342)
- Add support for `typing.TypedDict` and `typing_extensions.TypedDict`. (#296 #364)
- Add support for `typing.Final`. (#340 #349)
- Introduce `override.struct_hook` and `override.unstruct_hook`. Learn more here. (#326)
- Fix generating structuring functions for types with angle brackets (<>) and pipe symbols (|) in the name. (#319 #327)
- `pathlib.Path` is now supported by default. (#81)
- Add `cbor2` serialization library to the `cattrs.preconf` package.

- Add optional dependencies for `cattrs.preconf` third-party libraries. (#337)
- All preconf converters now allow overriding the default `unstruct_collection_overrides` in `make_converter`. (#350 #353)
- Subclasses structuring and unstructuring is now supported via a custom `include_subclasses` strategy. (#312)
- Add support for `typing_extensions.Annotated` when the python version is less than `3.9`. (#366)
- Add unstructuring and structuring support for the standard library `deque`. (#355)

15.11 22.2.0 (2022-10-03)

- *Potentially breaking*: `cattrs.Converter` has been renamed to `cattrs.BaseConverter`, and `cattrs.GenConverter` to `cattrs.Converter`. The `GenConverter` name is still available for backwards compatibility, but is deprecated. If you were depending on functionality specific to the old `Converter`, change your import to `from cattrs import BaseConverter`.
- NewTypes are now supported by the `cattrs.Converter`. (#255 #94 #297)
- `cattrs.Converter` and `cattrs.BaseConverter` can now copy themselves using the `copy` method. (#284)
- Python 3.11 support.
- cattrs now supports un/structuring `kw_only` fields on attrs classes into/from dictionaries. (#247)
- PyPy support (and tests, using a minimal Hypothesis profile) restored. (#253)
- Fix propagating the `detailed_validation` flag to mapping and counter structuring generators.
- Fix `typing.Set` applying too broadly when used with the `GenConverter.unstruct_collection_overrides` parameter on Python versions below 3.9. Switch to `typing.AbstractSet` on those versions to restore the old behavior. (#264)
- Uncap the required Python version, to avoid problems detailed here (#275)
- Fix `Converter.register_structure_hook_factory` and `cattrs.gen.make_dict_unstructure_fn` type annotations. (#281)
- Expose all error classes in the `cattr.errors` namespace. Note that it is deprecated, just use `cattrs.errors`. (#252)
- Fix generating structuring functions for types with quotes in the name. (#291 #277)
- Fix usage of notes for the final version of PEP 678, supported since `exceptiongroup>=1.0.0rc4`. (#303)

15.12 22.1.0 (2022-04-03)

- *cattrs* now uses the CalVer versioning convention.
- *cattrs* now has a detailed validation mode, which is enabled by default. Learn more here. The old behavior can be restored by creating the converter with `detailed_validation=False`.
- *attrs* and dataclass structuring is now ~25% faster.
- Fix an issue structuring bare `typing.List` s on Pythons lower than 3.9. (#209)
- Fix structuring of non-parametrized containers like `list/dict/...` on Pythons lower than 3.9. (#218)

- Fix structuring bare `typing.Tuple` on Pythons lower than 3.9. (#218)
- Fix a wrong `AttributeError` of an missing `__parameters__` attribute. This could happen when inheriting certain generic classes – for example `typing.*` classes are affected. (#217)
- Fix structuring of `enum.Enum` instances in `typing.Literal` types. (#231)
- Fix unstructuring all tuples - unannotated, variable-length, homogenous and heterogenous - to `list`. (#226)
- For `forbid_extra_keys` raise custom `ForbiddenExtraKeyError` instead of generic `Exception`. (#225)
- All preconf converters now support `loads` and `dumps` directly. See an example here.
- Fix mappings with byte keys for the orjson, bson and tomlkit converters. (#241)

15.13 1.10.0 (2022-01-04)

Note: In this release, *cattrs* introduces the *cattrs* package as the main entry point into the library, replacing the `cattr` package.

The `cattr` package is never going away, nor is it technically deprecated. New functionality will be added only to the `cattrs` package, but there is no need to replace your current imports.

This change mirrors a similar change in *attrs*.

- Add PEP 563 (string annotations) support for dataclasses. (#195)
- Fix handling of dictionaries with string Enum keys for bson, orjson, and tomlkit.
- Rename the *cattrs.gen.make_dict_unstructure_fn()* `omit_if_default` parameter to `_cattrs_omit_if_default`, for consistency. The `omit_if_default` parameters to *GenConverter* and *override()* are unchanged.
- Following the changes in *attrs* 21.3.0, add a *cattrs* package mirroring the existing `cattr` package. Both package names may be used as desired, and the `cattr` package isn't going away.

15.14 1.9.0 (2021-12-06)

- Python 3.10 support, including support for the new union syntax (`A | B` vs `Union[A, B]`).
- The `GenConverter` can now properly structure generic classes with generic collection fields. (#149)
- `omit=True` now also affects generated structuring functions. (#166)
- `cattr.gen.{make_dict_structure_fn, make_dict_unstructure_fn}` now resolve type annotations automatically when PEP 563 is used. (#169)
- Protocols are now unstructured as their runtime types. (#177)
- Fix an issue generating structuring functions with renaming and `_cattrs_forbid_extra_keys=True`. (#190)

15.15 1.8.0 (2021-08-13)

- Fix `GenConverter` mapping structuring for unannotated dicts on Python 3.8. (#151)
- The source code for generated un/structuring functions is stored in the `linecache` cache, which enables more informative stack traces when un/structuring errors happen using the `GenConverter`. This behavior can optionally be disabled to save memory.
- Support using the attr converter callback during structure. By default, this is a method of last resort, but it can be elevated to the default by setting `prefer_attrib_converters=True` on `Converter` or `GenConverter`. (#138)
- Fix structuring recursive classes. (#159)
- Converters now support un/structuring hook factories. This is the most powerful and complex venue for customizing un/structuring. This had previously been an internal feature.
- The Common Usage Examples documentation page now has a section on advanced hook factory usage.
- `cattr.override` now supports the `omit` parameter, which makes *cattrs* skip the atribute entirely when unstructuring.
- The `cattr.preconf.bson` module is now tested against the `bson` module bundled with the `pymongo` package, because that package is much more popular than the standalone PyPI `bson` package.

15.16 1.7.1 (2021-05-28)

- `Literal` s are not supported on Python 3.9.0 (supported on 3.9.1 and later), so we skip importing them there. (#150)

15.17 1.7.0 (2021-05-26)

- `cattr.global_converter` (which provides `cattr.unstructure`, `cattr.structure` etc.) is now an instance of `cattr.GenConverter`.
- `Literal` s are now supported and validated when structuring.
- Fix dependency metadata information for *attrs*. (#147)
- Fix `GenConverter` mapping structuring for unannotated dicts. (#148)

15.18 1.6.0 (2021-04-28)

- *cattrs* now uses Poetry.
- `GenConverter` mapping structuring is now ~25% faster, and unstructuring heterogenous tuples is significantly faster.
- Add `cattr.preconf`. This package contains modules for making converters for particular serialization libraries. We currently support the standard library `json`, and third-party `ujson`, `orjson`, `msgpack`, `bson`, `pyyaml` and `tomlkit` libraries.

15.19 1.5.0 (2021-04-15)

- Fix an issue with `GenConverter` unstructuring *attrs* classes and dataclasses with generic fields. (#65)
- `GenConverter` has support for easy overriding of collection unstructuring types (for example, unstructure all sets to lists) through its `unstruct_collection_overrides` argument. (#137)
- Unstructuring mappings with `GenConverter` is significantly faster.
- `GenConverter` supports strict handling of unexpected dictionary keys through its `forbid_extra_keys` argument. (#142)

15.20 1.4.0 (2021-03-21)

- Fix an issue with `GenConverter` un/structuring hooks when a function hook is registered after the converter has already been used.
- Add support for `collections.abc.{Sequence, MutableSequence, Set, MutableSet}`. These should be used on 3.9+ instead of their `typing` alternatives, which are deprecated. (#128)
- The `GenConverter` will unstructure iterables (`list[T]`, `tuple[T, ...]`, `set[T]`) using their type argument instead of the runtime class if its elements, if possible. These unstructuring operations are up to 40% faster. (#129)
- Flesh out `Converter` and `GenConverter` initializer type annotations. (#131)
- Add support for `typing.Annotated` on Python 3.9+. *cattrs* will use the first annotation present. *cattrs* specific annotations may be added in the future. (#127)
- Add support for dataclasses. (#43)

15.21 1.3.0 (2021-02-25)

- *cattrs* now has a benchmark suite to help make and keep cattrs the fastest it can be. The instructions on using it can be found under the Benchmarking section in the docs. (#123)
- Fix an issue unstructuring tuples of non-primitives. (#125)
- *cattrs* now calls `attr.resolve_types` on *attrs* classes when registering un/structuring hooks.
- `GenConverter` structuring and unstructuring of *attrs* classes is significantly faster.

15.22 1.2.0 (2021-01-31)

- `converter.unstructure` now supports an optional parameter, `unstructure_as`, which can be used to unstructure something as a different type. Useful for unions.
- Improve support for union un/structuring hooks. Flesh out docs for advanced union handling. (#115)
- Fix `GenConverter` behavior with inheritance hierarchies of *attrs* classes. ([#117](https://github.com/python-attrs/cattrs/pull/117 #116)
- Refactor `GenConverter.un/structure_attrs_fromdict` into `GenConverter.gen_un/structure_attrs_fromdict` to allow calling back to `Converter.un/structure_attrs_fromdict` without sideeffects. (#118)

15.23 1.1.2 (2020-11-29)

- The default disambiguator will not consider non-required fields any more. (#108)
- Fix a couple type annotations. (#107 #105)
- Fix a `GenConverter` unstructuring issue and tests.

15.24 1.1.1 (2020-10-30)

- Add metadata for supported Python versions. (#103)

15.25 1.1.0 (2020-10-29)

- Python 2, 3.5 and 3.6 support removal. If you need it, use a version below 1.1.0.
- Python 3.9 support, including support for built-in generic types (`list[int]` vs `typing.List[int]`).
- *cattrs* now includes functions to generate specialized structuring and unstructuring hooks. Specialized hooks are faster and support overrides (`omit_if_default` and `rename`). See the `cattr.gen` module.
- *cattrs* now includes a converter variant, `cattr.GenConverter`, that automatically generates specialized hooks for attrs classes. This converter will become the default in the future.
- Generating specialized structuring hooks now invokes attr.resolve_types on a class if the class makes use of the new PEP 563 annotations.
- *cattrs* now depends on *attrs* >= 20.1.0, because of `attr.resolve_types`.
- Specialized hooks now support generic classes. The default converter will generate and use a specialized hook upon encountering a generic class.

15.26 1.0.0 (2019-12-27)

- *attrs* classes with private attributes can now be structured by default.
- Structuring from dictionaries is now more lenient: extra keys are ignored.
- *cattrs* has improved type annotations for use with Mypy.
- Unstructuring sets and frozensets now works properly.

15.27 0.9.1 (2019-10-26)

- Python 3.8 support.

15.28 0.9.0 (2018-07-22)

- Python 3.7 support.

15.29 0.8.1 (2018-06-19)

- The disambiguation function generator now supports unions of *attrs* classes and NoneType.

15.30 0.8.0 (2018-04-14)

- Distribution fix.

15.31 0.7.0 (2018-04-12)

- Removed the undocumented `Converter.unstruct_strat` property setter.
- Removed the ability to set the `Converter.structure_attrs` instance field.
- Some micro-optimizations were applied; a `structure(unstructure(obj))` roundtrip is now up to 2 times faster.

15.32 0.6.0 (2017-12-25)

- Packaging fixes. (#17)

15.33 0.5.0 (2017-12-11)

- structure/unstructure now supports using functions as well as classes for deciding the appropriate function.
- added `Converter.register_structure_hook_func`, to register a function instead of a class for determining handler func.
- added `Converter.register_unstructure_hook_func`, to register a function instead of a class for determining handler func.
- vendored typing is no longer needed, nor provided.
- Attributes with default values can now be structured if they are missing in the input. (#15)
- `Optional` attributes can no longer be structured if they are missing in the input.
- `cattr.typed` removed since the functionality is now present in *attrs* itself. Replace instances of `cattr.typed(type)` with `attr.ib(type=type)`.

15.34 0.4.0 (2017-07-17)

- `Converter.loads` is now `Converter.structure`, and `Converter.dumps` is now `Converter.unstructure`.
- Python 2.7 is supported.
- Moved `cattr.typing` to `cattr.vendor.typing` to support different vendored versions of typing.py for Python 2 and Python 3.
- Type metadata can be added to *attrs* classes using `cattr.typed`.

15.35 0.3.0 (2017-03-18)

- Python 3.4 is no longer supported.
- Introduced `cattr.typing` for use with Python versions 3.5.2 and 3.6.0.
- Minor changes to work with newer versions of `typing`.
- Bare Optionals are not supported any more (use `Optional[Any]`).
- Attempting to load unrecognized classes will result in a ValueError, and a helpful message to register a loads hook.
- Loading *attrs* classes is now documented.
- The global converter is now documented.
- `cattr.loads_attrs_fromtuple` and `cattr.loads_attrs_fromdict` are now exposed.

15.36 0.2.0 (2016-10-02)

- Tests and documentation.

15.37 0.1.0 (2016-08-13)

- First release on PyPI.

PYTHON MODULE INDEX

C

INDEX

D

F

G

I

J

L

M

N

O

P

R

S

T

U

V

W