cattrs Documentation

Release 1.0.0

Tin Tvrtković

Oct 16, 2020

CONTENTS

Contents:

CHAPTER

ONE

CATTRS

`cattrs` is an open source Python library for structuring and unstructuring data. `cattrs` works best with `attrs` classes and the usual Python collections, but other kinds of classes are supported by manually registering converters.

Python has a rich set of powerful, easy to use, built-in data types like dictionaries, lists and tuples. These data types are also the lingua franca of most data serialization libraries, for formats like json, msgpack, yaml or toml.

Data types like this, and mappings like `dict` s in particular, represent unstructured data. Your data is, in all likelihood, structured: not all combinations of field names are values are valid inputs to your programs. In Python, structured data is better represented with classes and enumerations. `attrs` is an excellent library for declaratively describing the structure of your data, and validating it.

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

Here's a simple taste. The list containing a float, an int and a string gets converted into a tuple of three ints.

```
>>> import cattr
>>> from typing import Tuple
>>>
>>> cattr.structure([1.0, 2, "3"], Tuple[int, int, int])
(1, 2, 3)
```

`cattrs` works well with `attrs` classes out of the box.

```
>>> import attr, cattr
>>>
>>> @attr.s(slots=True, frozen=True)  # It works with normal classes too.
... class C:
...     a = attr.ib()
...     b = attr.ib()
...
>>> instance = C(1, 'a')
>>> cattr.unstructure(instance)
{'a': 1, 'b': 'a'}
```

(continues on next page)

(continued from previous page)

```
>>> cattr.structure({'a': 1, 'b': 'a'}, C)
C(a=1, b='a')
```

Here's a much more complex example, involving `attrs` classes with type metadata.

```
>>> from enum import unique, Enum
>>> from typing import List, Optional, Sequence, Union
>>> from cattr import structure, unstructure
>>> import attr
>>>
>>> @unique
... class CatBreed(Enum):
...     SIAMESE = "siamese"
...     MAINE_COON = "maine_coon"
...     SACRED_BIRMAN = "birman"
...
>>> @attr.s
... class Cat:
...     breed: CatBreed = attr.ib()
...     names: Sequence[str] = attr.ib()
...
>>> @attr.s
... class DogMicrochip:
...     chip_id = attr.ib()
...     time_chipped: float = attr.ib()
...
>>> @attr.s
... class Dog:
...     cuteness: int = attr.ib()
...     chip: Optional[DogMicrochip] = attr.ib()
...
>>> p = unstructure([Dog(cuteness=1, chip=DogMicrochip(chip_id=1, time_chipped=10.0)),
...                  Cat(breed=CatBreed.MAINE_COON, names=('Fluffly', 'Fluffer'))])
...
>>> print(p)
[{'cuteness': 1, 'chip': {'chip_id': 1, 'time_chipped': 10.0}}, {'breed': 'maine_coon
↪', 'names': ('Fluffly', 'Fluffer')}]
>>> print(structure(p, List[Union[Dog, Cat]]))
[Dog(cuteness=1, chip=DogMicrochip(chip_id=1, time_chipped=10.0)), Cat(breed=
↪<CatBreed.MAINE_COON: 'maine_coon'>, names=['Fluffly', 'Fluffer'])]
```

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. Use attrs type metadata to add type metadata to attributes, so `cattrs` will know how to structure and destructure them.

- Free software: MIT license
- Documentation: https://cattrs.readthedocs.io.
- Python versions supported: 3.7 and up. (Older Python versions, like 2.7, 3.5 and 3.6 are supported by older versions; see the changelog.)

1.1 Features

- Converts structured data into unstructured data, recursively:
  - `attrs` classes are converted into dictionaries in a way similar to `attr.asdict`, or into tuples in a way similar to `attr.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`.
- Converts unstructured data into structured data, recursively, according to your specification given as a type. The following types are supported:
  - `typing.Optional[T]`.
  - `typing.List[T]`, `typing.MutableSequence[T]`, `typing.Sequence[T]` (converts to a list).
  - `typing.Tuple` (both variants, `Tuple[T, ...]` and `Tuple[X, Y, Z]`).
  - `typing.MutableSet[T]`, `typing.Set[T]` (converts to a set).
  - `typing.FrozenSet[T]` (converts to a frozenset).
  - `typing.Dict[K, V]`, `typing.MutableMapping[K, V]`, `typing.Mapping[K, V]` (converts to a dict).
  - `attrs` classes with simple attributes and the usual `__init__`.
    - Simple attributes are attributes that can be assigned unstructured data, like numbers, strings, and collections of unstructured data.
  - All *attrs* classes with the usual `__init__`, if their complex attributes have type metadata.
  - `typing.Union` s of supported `attrs` classes, given that all of the classes have a unique field.
  - `typing.Union` s of anything, given that you provide a disambiguation function for it.
  - Custom converters for any type can be registered using `register_structure_hook`.

1.2 Credits

Major credits to Hynek Schlawack for creating attrs and its predecessor, characteristic.

`cattrs` is tested with Hypothesis, by David R. MacIver.

`cattrs` is benchmarked using perf, by Victor Stinner.

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install cattrs, run this command in your terminal:

```
$ pip install cattrs
```

This is the preferred method to install cattrs, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for cattrs can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/Tinche/cattrs
```

Or download the tarball:

```
$ curl  -OL https://github.com/Tinche/cattrs/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ pip install -e .
```

CHAPTER

THREE

COMMON USAGE EXAMPLES

This section covers common use examples of cattrs features.

3.1 Using Pendulum for Dates and Time

To use the excellent Pendulum library for datetimes, we need to register structuring and unstructuring hooks for it.

First, we need to decide on the unstructured representation of a datetime instance. Since all our datetimes will use the UTC time zone, we decide to use the UNIX epoch timestamp as our unstructured representation.

Define a class using Pendulum's DateTime:

```
import pendulum
from pendulum import DateTime

@attr.s
class MyRecord:
    a_string: str = attr.ib()
    a_datetime: DateTime = attr.ib()
```

Next, we register hooks for the `DateTime` class on a new `Converter` instance.

```
converter = Converter()

converter.register_unstructure_hook(DateTime, lambda dt: dt.timestamp())

converter.register_structure_hook(DateTime, lambda ts, _: pendulum.from_timestamp(ts))
```

And we can proceed with unstructuring and structuring instances of `MyRecord`.

```
>>> my_record = MyRecord('test', pendulum.datetime(2018, 7, 28, 18, 24))
>>> my_record
MyRecord(a_string='test', a_datetime=DateTime(2018, 7, 28, 18, 24, 0, tzinfo=Timezone(
↪'UTC')))

>>> converter.unstructure(my_record)
{'a_string': 'test', 'a_datetime': 1532802240.0}

>>> converter.structure({'a_string': 'test', 'a_datetime': 1532802240.0}, MyRecord)
MyRecord(a_string='test', a_datetime=DateTime(2018, 7, 28, 18, 24, 0, tzinfo=Timezone(
↪'UTC')))
```

After a while, we realize we *will* need our datetimes to have timezone information. We decide to switch to using the ISO 8601 format for our unstructured datetime instances.

```
>>> converter = cattr.Converter()
>>> converter.register_unstructure_hook(DateTime, lambda dt: dt.to_iso8601_string())
>>> converter.register_structure_hook(DateTime, lambda isostring, _: pendulum.
↪parse(isostring))

>>> my_record = MyRecord('test', pendulum.datetime(2018, 7, 28, 18, 24, tz='Europe/
↪Paris'))
>>> my_record
MyRecord(a_string='test', a_datetime=DateTime(2018, 7, 28, 18, 24, 0, tzinfo=Timezone(
↪'Europe/Paris')))

>>> converter.unstructure(my_record)
{'a_string': 'test', 'a_datetime': '2018-07-28T18:24:00+02:00'}

>>> converter.structure({'a_string': 'test', 'a_datetime': '2018-07-28T18:24:00+02:00
↪'}, MyRecord)
MyRecord(a_string='test', a_datetime=DateTime(2018, 7, 28, 18, 24, 0, tzinfo=Timezone(
↪'+02:00')))
```

CHAPTER

FOUR

CONVERTERS

All `cattrs` functionality is exposed through a `cattr.Converter` object. Global `cattrs` functions, such as `cattr.unstructure()`, 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.

4.1 Global converter

A global converter is provided for convenience as `cattr.global_converter`. The following functions implicitly use this global converter:

- `cattr.structure`
- `cattr.unstructure`
- `cattr.structure_attr_fromtuple`
- `cattr.structure_attr_fromdict`

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

Larger applications are strongly encouraged to create and customize a different, private instance of `Converter`.

4.2 Converter objects

To create a private converter, simply instantiate a `cattr.Converter`. Currently, a converter contains the following state:

- a registry of unstructure hooks, backed by a `singledispatch` and a `function_dispatch`.
- a registry of structure hooks, backed by a different `singledispatch` and `function_dispatch`.
- a LRU cache of union disambiguation functions.
- a reference to an unstructuring strategy (either AS_DICT or AS_TUPLE).
- a `dict_factory` callable, used for creating `dicts` when dumping `attrs` classes using `AS_DICT`.

4.3 cattr.GenConverter

The `GenConverter` is a converter subclass that automatically generates, compiles and caches specialized structuring and unstructuring hooks for `attrs` classes.

`GenConverter` differs from the old `cattr.Converter` 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

The `GenConverter` will become the default converter type in a later release.

CHAPTER

FIVE

WHAT YOU CAN STRUCTURE AND HOW

The philosophy of `cattrs` structuring is simple: give it an instance of Python built-in types and collections, and a type describing the data you want out. `cattrs` will convert the input data into the type you want, or throw an exception.

All structuring conversions are composable, where applicable. This is demonstrated further in the examples.

5.1 Primitive values

5.1.1 `typing.Any`

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

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

5.1.2 `int`, `float`, `str`, `bytes`

Use any of these primitive types to convert the object to the type.

```
>>> cattr.structure(1, str)
'1'
>>> cattr.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 conversion yourself, directly.

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

5.1.3 Enums

Enums will be structured by their values. This works even for complex values, like tuples.

```
>>> @unique
... class CatBreed(Enum):
...     SIAMESE = "siamese"
...     MAINE_COON = "maine_coon"
...     SACRED_BIRMAN = "birman"
...
>>> cattr.structure("siamese", CatBreed)
<CatBreed.SIAMESE: 'siamese'>
```

Again, in case of errors, the expected exceptions will fly out.

```
>>> cattr.structure("alsatian", CatBreed)
Traceback (most recent call last):
...
ValueError: 'alsatian' is not a valid CatBreed
```

5.2 Collections and other generics

5.2.1 Optionals

`Optional` primitives and collections are supported out of the box.

```
>>> cattr.structure(None, int)
Traceback (most recent call last):
...
TypeError: int() argument must be a string, a bytes-like object or a number, not
↪'NoneType'
>>> cattr.structure(None, Optional[int])
>>> # None was returned.
```

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

This generic type is composable with all other converters.

```
>>> cattr.structure(1, Optional[float])
1.0
```

5.2.2 Lists

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

- `Sequence[T]`
- `MutableSequence[T]`
- `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]`.

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

These generic types are composable with all other converters.

```
>>> cattr.structure((1, None, 3), List[Optional[str]])
['1', None, '3']
```

5.2.3 Sets and frozensets

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

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

Types converting to frozensets are:

- `FrozenSet[T]`
- `frozenset[T]`

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

```
>>> cattr.structure([1, 2, 3, 4], Set)
{1, 2, 3, 4}
```

These generic types are composable with all other converters.

```
>>> cattr.structure([[1, 2], [3, 4]], Set[FrozenSet[str]])
{frozenset({'1', '2'}), frozenset({'4', '3'})}
```

5.2.4 Dictionaries

Dicts can be produced from other mapping objects. To be more precise, the object being converted must expose an `items()` method producing an iterable key-value tuples, and be able to be passed to the `dict` constructor as an argument. Types converting to dictionaries are:

- `Dict[K, V]`
- `MutableMapping[K, V]`
- `Mapping[K, V]`
- `dict[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
>>> cattr.structure(OrderedDict([(1, 2), (3, 4)]), Dict)
{1: 2, 3: 4}
```

These generic types are composable with all other converters. Note both keys and values can be converted.

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

5.2.5 Homogeneous and heterogeneous tuples

Homogeneous and heterogeneous tuples can be produced 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 returned. Any type parameters set to `typing.Any` will be passed through unconverted.

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

The tuple conversion is composable with all other converters.

```
>>> cattr.structure([{1: 1}, {2: 2}], Tuple[Dict[str, float], ...])
({'1': 1.0}, {'2': 2.0})
```

5.2.6 Unions

Unions of `NoneType` and a single other type are supported (also known as `Optional` s). All other unions a require a disambiguation function.

Automatic Disambiguation

In the case of a union consisting exclusively of `attrs` classes, `cattrs` will attempt to generate a disambiguation function automatically; this will succeed only if each class has a unique field. Given the following classes:

```
>>> @attr.s
... class A:
...     a = attr.ib()
...     x = attr.ib()
...
>>> @attr.s
... class B:
...     a = attr.ib()
...     y = attr.ib()
...
>>> @attr.s
... class C:
...     a = attr.ib()
...     z = attr.ib()
...
```

`cattrs` can deduce only instances of `A` will contain *x*, only instances of `B` will contain `y`, etc. A disambiguation function using this information will then be generated and cached. This will happen automatically, the first time an appropriate union is structured.

Manual Disambiguation

To support arbitrary unions, register a custom structuring hook for the union (see *Registering custom structuring hooks*).

5.3 `attrs` classes

5.3.1 Simple `attrs` classes

`attrs` classes using primitives, collections of primitives and their own converters work out of the box. Given a mapping `d` and class `A`, `cattrs` will simply instantiate `A` with `d` unpacked.

```
>>> @attr.s
... class A:
...     a = attr.ib()
...     b = attr.ib(converter=int)
...
>>> cattr.structure({'a': 1, 'b': '2'}, A)
A(a=1, b=2)
```

`attrs` classes deconstructed into tuples can be structured using `cattr.structure_attrs_fromtuple` (`fromtuple` as in the opposite of `attr.astuple` and `converter.unstructure_attrs_astuple`).

```
>>> @attr.s
... class A:
...     a = attr.ib()
...     b = attr.ib(converter=int)
...
>>> cattr.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 = cattr.Converter(unstruct_strat=cattr.UnstructureStrategy.AS_TUPLE)
>>> @attr.s
... class A:
...     a = attr.ib()
...     b = attr.ib(converter=int)
...
>>> converter.structure(['string', '2'], A)
A(a='string', b=2)
```

Structuring from tuples can also be made the default for specific classes only; see registering custom structure hooks below.

5.3.2 Complex `attrs` classes

Complex `attrs` classes are classes with type information available for some or all attributes. These classes support almost arbitrary nesting.

Type information is supported by attrs directly, and can be set using type annotations when using Python 3.6+, or by passing the appropriate type to `attr.ib`.

```
>>> @attr.s
... class A:
...     a: int = attr.ib()
...
>>> attr.fields(A).a
Attribute(name='a', default=NOTHING, validator=None, repr=True, eq=True, order=True,␣
↪hash=None, init=True, metadata=mappingproxy({}), type=<class 'int'>, converter=None,
↪ kw_only=False, inherited=False, on_setattr=None)
```

Type information, when provided, can be used for all attribute types, not only attributes holding `attrs` classes.

```
>>> @attr.s
... class A:
...     a: int = attr.ib(default=0)
...
>>> @attr.s
... class B:
...     b = attr.ib(type=A)  # Legacy syntax.
...
>>> cattr.structure({'b': {'a': '1'}}, B)
B(b=A(a=1))
```

5.4 Registering custom structuring hooks

`cattrs` doesn't know how to structure non-`attrs` classes by default, so it has to be taught. This can be done by registering structuring hooks on a converter instance (including the global converter).

Here's an example involving a simple, classic (i.e. non-`attrs`) Python class.

```
>>> class C(object):
...     def __init__(self, a):
...         self.a = a
...     def __repr__(self):
...         return f'C(a={self.a})'
>>> cattr.structure({'a': 1}, C)
Traceback (most recent call last):
...
ValueError: Unsupported type: <class '__main__.C'>. Register a structure hook for it.
>>>
>>> cattr.register_structure_hook(C, lambda d, t: C(**d))
>>> cattr.structure({'a': 1}, C)
C(a=1)
```

The structuring hooks are callables that take two arguments: the object to convert to the desired class and the type to convert to. The type may seem redundant but is useful when dealing with generic types.

When using `cattr.register_structure_hook`, the hook will be registered on the global converter. If you want to avoid changing the global converter, create an instance of `cattr.Converter` and register the hook on that.

In some situations, it is not possible to decide on the converter using typing mechanisms alone (such as with attrs classes). In these situations, cattrs provides a register_structure_func_hook instead, which accepts a function to determine whether that type can be handled instead.

The function-based hooks are evaluated after the class-based hooks. In the case where both a class-based hook and a function-based hook are present, the class-based hook will be used.

```
>>> class D(object):
...     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"])
>>> cattr.register_structure_hook_func(lambda cls: getattr(cls, "custom", False),␣
↪lambda d, t: t.deserialize(d))
>>> cattr.structure({'a': 2}, D)
D(a=2)
```

CHAPTER

SIX

WHAT YOU CAN UNSTRUCTURE AND HOW

Unstructuring is intended to convert high-level, structured Python data (like instances of complex classes) into simple, unstructured data (like dictionaries).

Unstructuring is simpler than structuring in that no target types are required. Simply provide an argument to `unstructure` and `cattrs` will produce a result based on the registered unstructuring hooks. A number of default unstructuring hooks are documented here.

Unstructuring is primarily done using `Converter.unstructure`.

6.1 Primitive types and collections

Primitive types (integers, floats, strings...) are simply passed through. Collections are copied. There's relatively little value in unstructuring these types directly as they are already unstructured and third-party libraries tend to support them directly.

A useful use case for unstructuring collections is to create a deep copy of a complex or recursive collection.

```
>>> # A dictionary of strings to lists of tuples of floats.
>>> data = {'a': [(1.0, 2.0), (3.0, 4.0)]}
>>>
>>> copy = cattr.unstructure(data)
>>> data == copy
True
>>> data is copy
False
```

6.2 `attrs` classes

`attrs` classes are supported out of the box. `Converter` s support two unstructuring strategies:

- `UnstructureStrategy.AS_DICT` - similar to `attr.asdict`, unstructures `attrs` instances into dictionaries. This is the default.
- `UnstructureStrategy.AS_TUPLE` - similar to `attr.astuple`, unstructures `attrs` instances into tuples.

```
>>> @attr.s
... class C:
...     a = attr.ib()
...     b = attr.ib()
```

(continues on next page)

(continued from previous page)

```
...
>>> inst = C(1, 'a')
>>>
>>> converter = cattr.Converter(unstruct_strat=cattr.UnstructureStrategy.AS_TUPLE)
>>>
>>> converter.unstructure(inst)
(1, 'a')
```

6.2.1 Mixing and matching strategies

Converters publicly expose two helper metods, `Converter.unstructure_attrs_asdict()` and `Converter.unstructure_attrs_astuple()`. These methods can be used with custom unstructuring hooks to selectively apply one strategy to instances of particular classes.

Assume two nested `attrs` classes, `Inner` and `Outer`; instances of `Outer` contain instances of `Inner`. Instances of `Outer` should be unstructured as dictionaries, and instances of `Inner` as tuples. Here's how to do this.

```
>>> @attr.s
... class Inner:
...     a: int = attr.ib()
...
>>> @attr.s
... class Outer:
...     i: Inner = attr.ib()
...
>>> inst = Outer(i=Inner(a=1))
>>>
>>> converter = cattr.Converter()
>>> converter.register_unstructure_hook(Inner, converter.unstructure_attrs_astuple)
>>>
>>> converter.unstructure(inst)
{'i': (1,)}
```

Of course, these methods can be used directly as well, without changing the converter strategy.

```
>>> @attr.s
... class C:
...     a: int = attr.ib()
...     b: str = attr.ib()
...
>>> inst = C(1, 'a')
>>>
>>> converter = cattr.Converter()
>>>
>>> converter.unstructure_attrs_astuple(inst)  # Default is AS_DICT.
(1, 'a')
```

CHAPTER

SEVEN

CUSTOMIZING CLASS UN/STRUCTURING

This section deals with customizing the unstructuring and structuring processes in `cattrs`.

7.1 Manual 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.

7.2 Using `cattr.gen` generators

`cattrs` includes a module, `cattr.gen`, which allows for generating and compiling specialized functions for unstructuring `attrs` classes.

One reason for generating these functions in advance is that they can bypass a lot of `cattrs` machinery and be significantly faster than normal `cattrs`.

Another reason is that it's possible to override behavior on a per-attribute basis.

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

7.2.1 `omit_if_default`

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

```
>>> from cattr.gen import make_dict_unstructure_fn, override
>>>
>>> @attr.s
... class WithDefault:
...     a = attr.ib()
...     b = attr.ib(factory=dict)
>>>
>>> c = cattr.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.

```
>>> from pendulum import DateTime
>>> from cattr.gen import make_dict_unstructure_fn, override
>>>
>>> @attr.s
... class TestClass:
...     a: Optional[int] = attr.ib(default=None)
...     b: DateTime = attr.ib(factory=DateTime.utcnow)
>>>
>>> c = cattr.Converter()
>>> hook = make_dict_unstructure_fn(TestClass, c, 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.

7.2.2 `rename`

Using the rename override makes `cattrs` simply 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 cattr.gen import make_dict_unstructure_fn, make_dict_structure_fn, override
>>>
>>> @attr.s
... class ExampleClass:
...     klass: Optional[int] = attr.ib()
>>>
>>> c = cattr.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)
```

7.3 Using `cattr.gen.GenConverter`

The `cattr.gen` module also contains a `Converter` subclass, the `GenConverter`. The `GenConverter`, upon first encountering an `attrs` class, will use the mentioned generation functions to generate the specialized hooks for it, register the hooks and use them.

CHAPTER

EIGHT

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:

8.1 Types of Contributions

8.1.1 Report Bugs

Report bugs at https://github.com/Tinche/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.

8.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

8.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

8.1.4 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.

8.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/Tinche/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 :)

8.2 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 virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv cattrs
$ cd cattrs/
$ pip install -e .\[dev\]
```

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:

```
$ flake8 cattrs tests
$ pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

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.

8.3 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://travis-ci.org/Tinche/cattrs/pull_requests and make sure that the tests pass for all supported Python versions.

8.4 Tips

To run a subset of tests:

```
$ pytest tests.test_unstructure
```

CHAPTER

NINE

HISTORY

9.1 1.1.0 (UNRELEASED)

- 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.

9.2 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.

9.3 0.9.1 (2019-10-26)

- Python 3.8 support.

9.4 0.9.0 (2018-07-22)

- Python 3.7 support.

9.5 0.8.1 (2018-06-19)

- The disambiguation function generator now supports unions of `attrs` classes and NoneType.

9.6 0.8.0 (2018-04-14)

- Distribution fix.

9.7 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.
  As an alternative, create a new `Converter`::

  .. code-block:: python

      >>> converter = cattr.Converter(unstruct_strat=cattr.UnstructureStrategy.AS_TUPLE)
- Some micro-optimizations were applied; a `structure(unstructure(obj))` roundtrip is now up to 2 times faster.

9.8 0.6.0 (2017-12-25)

- Packaging fixes. (#17)

9.9 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.
  In other words, this no longer works:

.. code-block:: python

```
@attr.s
class A:
    a: Optional[int] = attr.ib()

>>> cattr.structure({}, A)
```

- `cattr.typed` removed since the functionality is now present in `attrs` itself. Replace instances of `cattr.typed(type)` with `attr.ib(type=type)`.

9.10 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`.

9.11 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.

9.12 0.2.0 (2016-10-02)

- Tests and documentation.

9.13 0.1.0 (2016-08-13)

- First release on PyPI.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search