Structured Data

Release 0.13.0

Sep 30, 2019

Contents

CHAPTER 1

Overview

Code generators for immutable structured data, including algebraic data types, and functions to destructure them. Structured Data provides three public modules: `structured_data.adt`, `structured_data.match`, and `structured_data.data`.

The `adt` module provides base classes and an annotation type for converting a class into algebraic data types.

The `match` module provides a `Pattern` class that can be used to build match structures, and a `Matchable` class that wraps a value, and attempts to apply match structures to it. If the match succeeds, the bindings can be extracted and used. It includes some special support for `adt` subclasses.

The match architecture allows you tell pull values out of a nested structure:

```
structure = (match.pat.a, match.pat.b[match.pat.c, match.pat.d], 5)
my_value = (('abc', 'xyz'), ('def', 'ghi'), 5)
matchable = match.Matchable(my_value)
if matchable(structure):
    # The format of the matches is not final.
    print(matchable['a'])  # ('abc', 'xyz')
    print(matchable['b'])  # ('def', 'ghi')
    print(matchable['c'])  # 'def'
    print(matchable['d'])  # 'ghi'
```

The subscript operator allows binding both the outside and the inside of a structure. Indexing a `Matchable` is forwarded to a `matches` attribute, which is `None` if the last match was not successful, and otherwise contains an instance of a custom mapping type, which allows building the matched values back up into simple structures.

The `Sum` base class exists to create classes that do not necessarily have a single fixed format, but do have a fixed set of possible formats. This lowers the maintenance burden of writing functions that operate on values of a `Sum` class, because the full list of cases to handle is directly in the class definition.

Here are implementations of common algebraic data types in other languages:

```
class Maybe(adt.Sum, typing.Generic[T]):

    Just: adt.Ctor[T]
```

(continues on next page)

(continued from previous page)

```
    Nothing: adt.Ctor


class Either(adt.Sum, typing.Generic[E, R]):

    Left: adt.Ctor[E]
    Right: adt.Ctor[R]
```

The `data` module provides classes based on these examples.

- Free software: MIT license

1.1 How Can I Help?

Currently, this project has somewhat high quality metrics, though some of them have been higher. I am highly skeptical of this, because I've repeatedly given in to the temptation to code to the metrics. I can't trust the metrics, and I know the code well enough that I can't trust my own judgment to figure out which bits need to be improved and how. I need someone to review the code and identify problem spots based on what doesn't make sense to them. The issues are open.

1.2 Should I Use This?

Until there's a major version out, probably not.

There are several alternatives in the standard library that may be better suited to particular use-cases:

- The `namedtuple` factory creates tuple classes with a single structure; the `typing.NamedTuple` class offers the ability to include type information. The interface is slightly awkward, and the values expose their tuple-nature easily. (NOTE: In Python 3.8, the fast access to namedtuple members means that they bypass user-defined `__getitem__` methods, thereby allowing factory consumers to customize indexing without breaking attribute access. It looks like it does still rely on iteration behavior for various convenience methods.)
- The `enum` module provides base classes to create finite enumerations. Unlike NamedTuple, the ability to convert values into an underlying type must be opted into in the class definition.
- The `dataclasses` module provides a class decorator that converts a class into one with a single structure, similar to a namedtuple, but with more customization: instances are mutable by default, and it's possible to generate implementations of common protocols.
- The Structured Data `adt` decorator is inspired by the design of `dataclasses`. (A previous attempt used metaclasses inspired by the `enum` module, and was a nightmare.) Unlike `enum`, it doesn't require all instances to be defined up front; instead each class defines constructors using a sequence of types, which ultimately determines the number of arguments the constructor takes. Unlike `namedtuple` and `dataclasses`, it allows instances to have multiple shapes with their own type signatures. Unlike using regular classes, the set of shapes is specified up front.
- If you want multiple shapes, and don't want to specify them ahead of time, your best bet is probably a normal tree of classes, where the leaf classes are `dataclasses`.

1.3 Installation

```
pip install structured-data
```

1.4 Documentation

https://python-structured-data.readthedocs.io/

1.5 Development

To run the all tests run:

```
tox
```

CHAPTER 2

Installation

At the command line:

```
pip install structured-data
```

CHAPTER 3

Usage

To use Structured Data in a project:

```
import structured_data
```

Structured Data provides several related facilities.

- To define algebraic data types, see the *structured_data.adt* module.
- To perform destructuring matches of data, see the *structured_data.match* module.

CHAPTER 4

Reference

4.1 structured_data.adt

Base classes for defining abstract data types.

This module provides three public members, which are used together.

Given a structure, possibly a choice of different structures, that you'd like to associate with a type:

- First, create a class, that subclasses the Sum class.
- Then, for each possible structure, add an attribute annotation to the class with the desired name of the constructor, and a type of `Ctor`, with the types within the constructor as arguments.

To look inside an ADT instance, use the functions from the *`structured_data.match`* module.

Putting it together:

```
>>> from structured_data import match
>>> class Example(Sum):
...     FirstConstructor: Ctor[int, str]
...     SecondConstructor: Ctor[bytes]
...     ThirdConstructor: Ctor
...     def __iter__(self):
...         matchable = match.Matchable(self)
...         if matchable(Example.FirstConstructor(match.pat.count, match.pat.string)):
...             count, string = matchable[match.pat.count, match.pat.string]
...             for _ in range(count):
...                 yield string
...         elif matchable(Example.SecondConstructor(match.pat.bytes)):
...             bytes_ = matchable[match.pat.bytes]
...             for byte in bytes_:
...                 yield chr(byte)
...         elif matchable(Example.ThirdConstructor()):
...             yield "Third"
...             yield "Constructor"
```

(continues on next page)

(continued from previous page)

```
>>> list(Example.FirstConstructor(5, "abc"))
['abc', 'abc', 'abc', 'abc', 'abc']
>>> list(Example.SecondConstructor(b"abc"))
['a', 'b', 'c']
>>> list(Example.ThirdConstructor())
['Third', 'Constructor']
```

class structured_data.adt.**Ctor**
Marker class for adt constructors.

To use, index with a sequence of types, and annotate a variable in an adt-decorated class with it.

class structured_data.adt.**Product**
Base class of classes with typed fields.

Examines PEP 526 __annotations__ to determine fields.

If repr is true, a __repr__() method is added to the class. If order is true, rich comparison dunder methods are added.

The Product class examines the class to find annotations. Annotations with a value of "None" are discarded. Fields may have default values, and can be set to inspect.empty to indicate "no default".

The subclass is subclassable. The implementation was designed with a focus on flexibility over ideals of purity, and therefore provides various optional facilities that conflict with, for example, Liskov substitutability. For the purposes of matching, each class is considered distinct.

class structured_data.adt.**Sum**
Base class of classes with disjoint constructors.

Examines PEP 526 __annotations__ to determine subclasses.

If repr is true, a __repr__() method is added to the class. If order is true, rich comparison dunder methods are added.

The Sum class examines the class to find Ctor annotations. A Ctor annotation is the adt.Ctor class itself, or the result of indexing the class, either with a single type hint, or a tuple of type hints. All other annotations are ignored.

The subclass is not subclassable, but has subclasses at each of the names that had Ctor annotations. Each subclass takes a fixed number of arguments, corresponding to the type hints given to its annotation, if any.

4.2 structured_data.data

Example types showing simple usage of adt.Sum.

class structured_data.data.**Either**
An ADT that wraps one type, or the other.

class Left
Auto-generated subclass Left of ADT Either.

Takes 1 argument.

class Right
Auto-generated subclass Right of ADT Either.

Takes 1 argument.

class structured_data.data.**Maybe**
An ADT that wraps a value, or nothing.

class Just
Auto-generated subclass Just of ADT Maybe.

Takes 1 argument.

class Nothing
Auto-generated subclass Nothing of ADT Maybe.

Takes 0 arguments.

4.3 structured_data.match

Utilities for destructuring values using matchables and match targets.

Given a value to destructure, called `value`:

- Construct a matchable: `matchable = Matchable(value)`
- The matchable is initially falsy, but it will become truthy if it is passed a **match target** that matches `value`: `assert matchable(some_pattern_that_matches)` (Matchable returns itself from the call, so you can put the calls in an if-elif block, and only make a given call at most once.)
- When the matchable is truthy, it can be indexed to access bindings created by the target.

class structured_data.match.**AttrPattern**
A matcher that destructures an object using attribute access.

The `AttrPattern` constructor takes keyword arguments. Each name-value pair is the name of an attribute, and a matcher to apply to that attribute.

Attributes are checked in the order they were passed.

destructure(*value*) → Union[Tuple[()], Tuple[Any, Any]]
Return a tuple of sub-values to check.

If self is empty, return no values from self or the target.

Special-case matching against another AttrPattern as follows: Confirm that the target isn't smaller than self, then Extract the first match from the target's match_dict, and Return the smaller value, and the first match's value. (This works as desired when value is self, but all other cases where `isinstance(value, AttrPattern)` are unspecified.)

By default, it takes the first match from the match_dict, and returns the original value, and the result of calling `getattr` with the target and the match's key.

match_dict
Return the dict of matches to check.

class structured_data.match.**Bind**
A wrapper that adds additional bindings to a successful match.

The `Bind` constructor takes a single required argument, and any number of keyword arguments. The required argument is a matcher. When matching, if the match succeeds, the `Bind` instance adds bindings corresponding to its keyword arguments.

First, the matcher is checked, then the bindings are added in the order they were passed.

bindings
Return the bindings to add to the match.

destructure(*value*)
Return a list of sub-values to check.

If `value is self`, return all of the bindings, and the structure.

Otherwise, return the corresponding bound values, followed by the original value.

structure
Return the structure to match against.

class structured_data.match.**DictPattern**
A matcher that destructures a dictionary by key.

The `DictPattern` constructor takes a required argument, a dictionary where the keys are keys to check, and the values are matchers to apply. It also takes an optional keyword argument, "exhaustive", which defaults to False. If "exhaustive" is True, then the match requires that the matched dictionary has no keys not in the `DictPattern`. Otherwise, "extra" keys are ignored.

Keys are checked in iteration order.

destructure(*value*) → Union[Tuple[()], Tuple[Any, Any]]
Return a tuple of sub-values to check.

If self is exhaustive and the lengths don't match, fail.

If self is empty, return no values from self or the target.

Special-case matching against another DictPattern as follows: Confirm that the target isn't smaller than self, then Extract the first match from the target's match_dict, and Return the smaller value, and the first match's value. Note that the returned DictPattern is never exhaustive; the exhaustiveness check is accomplished by asserting that the lengths start out the same, and that every key in self is present in value. (This works as desired when value is self, but all other cases where `isinstance(value, DictPattern)` are unspecified.)

By default, it takes the first match from the match_dict, and returns the original value, and the result of indexing the target with the match's key.

exhaustive
Return whether the target must of the exact keys as self.

exhaustive_length_must_match(*value: Sized*)
If the match is exhaustive and the lengths differ, fail.

match_dict
Return the dict of matches to check.

class structured_data.match.**MatchDict**
A MutableMapping that allows for retrieval into structures.

The actual keys in the mapping must be string values. Most of the mapping methods will only operate on or yield string keys. The exception is subscription: the "key" in subscription can be a structure made of tuples and dicts. For example, `md["a", "b"] == (md["a"], md["b"])`, and `md[{1: "a"}] == {1: md["a"]}`. The typical use of this will be to extract many match values at once, as in `a, b, c == md["a", "b", "c"]`.

The behavior of most of the pre-defined MutableMapping methods is currently neither tested nor guaranteed.

class structured_data.match.**Matchable**(*value: Any*)
Given a value, attempt to match against a target.

The truthiness of `Matchable` values varies on whether they have bindings associated with them. They are truthy exactly when they have bindings.

`Matchable` values provide two basic forms of syntactic sugar. `m_able(target)` is equivalent to `m_able.match(target)`, and `m_able[k]` will return `m_able.matches[k]` if the `Matchable` is truthy, and raise a `ValueError` otherwise.

match(*target*) → structured_data._match.matchable.Matchable
Match against target, generating a set of bindings.

class structured_data.match.**Pattern**
A matcher that binds a value to a name.

A `Pattern` can be indexed with another matcher to produce an `AsPattern`. When matched with a value, an `AsPattern` both binds the value to the name, and uses the matcher to match the value, thereby constraining it to have a particular shape, and possibly introducing further bindings.

name
Return the name of the matcher.

class structured_data.match.**Property**(*func=None*, *fset=None*, *fdel=None*, *doc=None*, **args*, ***kwargs*)
Decorator with value-based dispatch. Acts as a property.

delete_when(*instance*)
Add a binding to the deleter.

deleter(*deleter*)
Return a copy of self with the deleter replaced.

get_when(*instance*)
Add a binding to the getter.

getter(*getter*)
Return a copy of self with the getter replaced.

set_when(*instance*, *value*)
Add a binding to the setter.

setter(*setter*)
Return a copy of self with the setter replaced.

structured_data.match.**decorate_in_order**(**args*)
Apply decorators in the order they're passed to the function.

structured_data.match.**function**(*_func=None*, ***, *positional_until=0*)
Convert a function to dispatch by value.

The original function is not called when the dispatch function is invoked.

structured_data.match.**names**(*target*) → List[str]
Return every name bound by a target.

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When 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.

5.2 Documentation improvements

Structured Data could always use more documentation, whether as part of the official Structured Data docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/mwchase/python-structured-data/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 code contributions are welcome :)

5.4 Development

To set up *python-structured-data* for local development:

1. Fork python-structured-data (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/python-structured-data.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. 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
```

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 6

Authors

- Max Woerner Chase - https://mwchase.neocities.org

CHAPTER 7

Changelog

7.1 Unreleased

7.2 0.13.0 (2019-09-29)

7.2.1 Added

- `match.function` and `match.Property` decorators for Haskell-style function definitions.

7.2.2 Fixed

- Accessing data descriptors on `Sum` and `Product` instances.

7.3 0.12.1 (2019-09-04)

7.3.1 Added

- Product classes can make use of custom `__new__`.

7.4 0.12.0 (2019-09-03)

7.4.1 Added

- Product base class

7.4.2 Changed

- Improved documentation of some match constructors.
- Exposed `MatchDict` type, so it gets documented.
- Converted the `adt` decorator to a `Sum` base class.

7.4.3 Removed

- `Guard` type removed in favor of user-defined validation functions.

7.5 0.11.1 (2019-03-23)

7.5.1 Changed

- Restore proper behavior of `__new__` overrides.

7.6 0.11.0 (2019-03-23)

7.6.1 Changed

- Consider all overrides of checked dunder methods, not just those in the decorated class.

7.7 0.10.1 (2019-03-22)

7.7.1 Added

- A non-ergonomic but simple wrapper class for use by the typing plugin. It's not available to runtime code.

7.8 0.10.0 (2019-03-21)

7.8.1 Changed

- Actually, the facade was working, I was just confused. Restored the facade.

7.9 0.9.0 (2019-03-20)

7.9.1 Changed

- Removed the facade.
- Added stability guarantee to Ctor.

7.10 0.8.0 (2019-03-19)

7.10.1 Changed

- Rewrote the facade.

7.11 0.7.0 (2019-03-19)

7.11.1 Changed

- Tried to put up a facade for type analysis. It didn't work.

7.12 0.6.1 (2019-03-18)

7.12.1 Added

- `Bind` class for attaching extra data to a match structure.
- PEP 561 support.

7.12.2 Changed

- As-patterns are now formed with indexing instead of the `@` operator.
- `AttrPattern` and `DictPattern` now take keyword arguments instead of a `dict` argument, and form new versions of themselves with an `alter` method.
- Actually. Change `DictPattern` back, stop trying to keep these things in synch.

7.13 0.6.0 (2018-07-27)

7.13.1 Added

- `AttrPattern` and `DictPattern` classes that take a `dict` argument and perform destructuring match against arbitrary objects, and mappings, respectively.

7.13.2 Changed

- Added special handling for matching AsPatterns against different AsPatterns. This is subject to change, as it's definitely an edge case.

7.14 0.5.0 (2018-07-22)

7.14.1 Added

- `Matchable` class is now callable and indexable. Calling is forwarded to the `match` method, and indexing forwards to the `matches` attribute, if it exists, and raises an error otherwise.
- `Matchable` class now has custom coercion to bool: `False` if the last match attempt failed, `True` otherwise.

7.14.2 Changed

- Renamed `enum` to `adt` to avoid confusion.
- Renamed `ValueMatcher` to `Matchable`.
- `Matchable.match` now returns the `Matchable` instance, which can then be coerced to `bool`, or indexed directly.

7.15 0.4.0 (2018-07-21)

7.15.1 Added

- Mapping class especially for match values. It's capable of quickly and concisely pulling out groups of variables, but it also properly supports extracting just a single value.
- Mapping class can now index from a `dict` to a `dict`, in order to support `**kwargs` unpacking.

7.15.2 Fixed

- A bug (not present in any released version) that caused the empty tuple target to accept any tuple value. This is included partly because this was just such a weird bug.

7.15.3 Removed

- Unpublished the `MatchFailure` exception type, and the `desugar` function.

7.16 0.3.0 (2018-07-15)

7.16.1 Added

- Simpler way to create match bindings.
- Dependency on the `astor` library.
- First attempt at populating the annotations and signature of the generated constructors.
- `data` module containing some generic algebraic data types.
- Attempts at monad implementations for `data` classes.

7.16.2 Changed

- Broke the package into many smaller modules.
- Switched many attributes to use a `WeakKeyDictionary` instead.
- Moved prewritten methods into a class to avoid defining reserved methods at the module level.
- When assigning equality methods is disabled for a decorated class, the default behavior is now `object` semantics, rather than failing comparison and hashing with a `TypeError`.
- The prewritten comparison methods no longer return `NotImplemented`.

7.16.3 Removed

- Ctor metaclass.

7.17 0.2.1 (2018-07-13)

7.17.1 Fixed

- Removed an incorrect classifier. This code cannot run on pypy.

7.18 0.2.0 (2018-07-13)

7.18.1 Added

- Explicit `__bool__` implementation, to consider all constructor instances as truthy, unless defined otherwise.
- Python 3.7 support.

7.18.2 Changed

- Marked the enum constructor base class as private. (`EnumConstructor` -> `_EnumConstructor`)
- Switched scope of test coverage to supported versions. (Python 3.7)

7.18.3 Removed

- Support for Python 3.6 and earlier.
- Incidental functionality required by supported Python 3.6 versions. (Hooks to enable restricted subclassing.)

7.19 0.1.0 (2018-06-10)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index