tri.declarative Documentation

Release 5.7.0

Anders Hovmöller

Jul 11, 2022

Contents

tri.declarative contains tools that make it easy to write declarative code. This includes:

- *class decorators* to define classes with subclass semantics in the style of django Model classes
- recursively *evaluating* embedded lambda expressions in complex data structures
- recursively *filtering* of complex data structures
- *keyword argument dispatching*
- *get/set attribute given a path string* (e.g. 'foo__bar__baz')

CHAPTER 1

Class decorators

With just a few lines of code, turn your API from:

```
quux = Foo(things=[Bar(name='a', param=1), Bar(name='b', param=2), Bar(name='c',␣
↪param=2)], baz=3)
```

into:

```
class Quux(Foo):
    a = Bar(param=1)
    b = Bar(param=2)
    c = Bar(param=2)

    class Meta:
        baz = 3
```

And you can still use the first style when it's more convenient!

More detailed usage examples on *@declarative* below.

CHAPTER 2

Evaluating

```
d = dict(
    foo=lambda x: x*2,
    bar=lambda y: y+5,
    baz=[
        foo=lambda x: x*6,
    ],
)

# evaluate only one level
assert evaluate(d, x=2) == dict(
    foo=4,
    bar=lambda y: y+5,  # this function doesn't match the signature so isn't evaluated
    baz=[
        foo=lambda x: x*6,  # one level down so isn't evaluated
    ],
)

# evaluate recursively
assert evaluate_recursive(d, x=2) == dict(
    foo=4,
    bar=lambda y: y+5,  # this function doesn't match the signature so isn't evaluated
    baz=[
        foo=12,
    ],
)
```

CHAPTER 3

Filtering

```
d = dict(
    foo=dict(
        show=False,
        x=1,
    ),
    bar=dict(
        show=True,
        x=2,
    ),
)

assert filter_show_recursive(d) == dict(
    bar=dict(
        show=True,
        x=2,
    ),
)
```

CHAPTER 4

Keyword argument dispatching

@dispatch:

```
@dispatch(
    bar={},
    baz__foo=2)
def foo(bar, baz):
    do_bar(**bar)
    do_baz(**baz)
```

CHAPTER 5

Get/set attribute given a path string

```
class Foo:
    def __init__(a):
        self.a = a

class Bar:
    def __init__(b):
        self.b = b

class Baz:
    def __init__(c):
        self.c = c

x = Foo(Bar(Baz(c=3)))

assert getattr_path(x, 'a__b__c') == 3

assert setattr_path(x, 'a__b__c', 10)
assert getattr_path(x, 'a__b__c') == 10
```

CHAPTER 6

Running tests

You need tox installed then just *make test*.

CHAPTER 7

License

BSD

CHAPTER 8

Documentation

https://trideclarative.readthedocs.org.

CHAPTER 9

Usage

9.1 @declarative

In the example below, the `@declarative(str)` decorator will ensure that all `str` members of class Foo will be collected and sent as `members` constructor keyword argument.

```
from tri_declarative import declarative

@declarative(str)
class Foo:
    bar = 'barbar'
    baz = 'bazbaz'
    boink = 17

    def __init__(self, members):
        assert members['bar'] == 'barbar'
        assert members['baz'] == 'bazbaz'
        assert 'boink' not in members

f = Foo()
```

The value of the `members` argument will also be collected from sub-classes:

```
from tri_declarative import declarative

@declarative(str)
class Foo:

    def __init__(self, members):
        assert members['bar'] == 'barbar'
        assert members['baz'] == 'bazbaz'

class MyFoo(Foo):
    bar = 'barbar'
```

(continues on next page)

(continued from previous page)

```
    baz = 'bazbaz'

    def __init__(self):
        super(MyFoo, self).__init__()

f = MyFoo()
```

The `members` argument can be given another name (`things` in the example below).

```
from tri_declarative.declarative import declarative

@declarative(str, 'things')
class Foo:

    bar = 'barbar'

    def __init__(self, **kwargs):
        assert 'things' in kwargs
        assert kwargs['things']['bar'] == 'barbar'

f = Foo()
```

Note that the collected dict is ordered by class inheritance and by using `sorted` of the values within each class. (In the 'str' example, `sorted` yields in alphabetical order).

Also note that the collection of *class* members based on their class does *not* interfere with *instance* constructor argument of the same type.

```
from tri_declarative import declarative

@declarative(str)
class Foo:
    charlie = '3'
    alice = '1'

    def __init__(self, members):
        assert list(members.items()) == [('alice', '1'), ('charlie', '3'),
                                         ('bob', '2'), ('dave', '4'),
                                         ('eric', '5')])
        assert 'animal' not in members


class MyFoo(Foo):
    dave = '4'
    bob = '2'

class MyOtherFoo(MyFoo):
    eric = '5'

    def __init__(self, animal)
        assert animal == 'elephant'

f = MyOtherFoo('elephant')
```

9.2 Real world use-case

Below is a more complete example of using @declarative:

```
from tri_declarative import declarative, creation_ordered


@creation_ordered
class Field:
    pass


class IntField(Field):
    def render(self, value):
        return '%s' % value


class StringField(Field):
    def render(self, value):
        return "'%s'" % value


@declarative(Field, 'table_fields')
class SimpleSQLModel:

    def __init__(self, **kwargs):
        self.table_fields = kwargs.pop('table_fields')

        for name in kwargs:
            assert name in self.table_fields
            setattr(self, name, kwargs[name])

    def insert_statement(self):
        return 'INSERT INTO %s(%s) VALUES (%s)' % (self.__class__.__name__,
                                                   ', '.join(self.table_fields.keys()),
                                                   ', '.join([field.render(getattr(self,
↪ name))
                                                                           for name, field in self.
↪table_fields.items()]))


class User(SimpleSQLModel):
    username = StringField()
    password = StringField()
    age = IntField()


my_user = User(username='Bruce_Wayne', password='Batman', age=42)
assert my_user.username == 'Bruce_Wayne'
assert my_user.password == 'Batman'
assert my_user.insert_statement() == "INSERT INTO User(username, password, age)␣
↪VALUES ('Bruce_Wayne', 'Batman', 42)"

# Fields are ordered by creation time (due to having used the @creation_ordered␣
↪decorator)
assert list(my_user.get_declared('table_fields').keys()) == ['username', 'password',
↪'age']
```

9.3 @with_meta

Class decorator to enable a class (and it's sub-classes) to have a 'Meta' class attribute.

The members of the Meta class will be injected as arguments to constructor calls. e.g.:

```
from tri_declarative import with_meta

@with_meta
class Foo:

    class Meta:
        foo = 'bar'

    def __init__(self, foo, buz):
        assert foo == 'bar'
        assert buz == 'buz'

foo = Foo(buz='buz')

# Members of the 'Meta' class can be accessed thru the get_meta() class method.
assert foo.get_meta() == {'foo': 'bar'}
assert Foo.get_meta() == {'foo': 'bar'}

Foo()  # Crashes, has 'foo' parameter, but no has no 'buz' parameter.
```

The passing of the merged name space to the constructor is optional. It can be disabled by passing `add_init_kwargs=False` to the decorator.

```
from tri_declarative import with_meta

@with_meta(add_init_kwargs=False)
class Foo:
    class Meta:
        foo = 'bar'

Foo()  # No longer crashes
assert Foo().get_meta() == {'foo': 'bar'}
```

Another example:

```
from tri_declarative import with_meta

class Foo:

    class Meta:
        foo = 'bar'
        bar = 'bar'

@with_meta
class Bar(Foo):

    class Meta:
        foo = 'foo'
        buz = 'buz'

    def __init__(self, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        assert kwargs['foo'] == 'foo'  # from Bar (overrides Foo)
        assert kwargs['bar'] == 'bar'  # from Foo
        assert kwargs['buz'] == 'buz'  # from Bar
```

This can be used e.g to enable sub-classes to modify constructor default arguments.

CHAPTER 10

Contents:

10.1 Installation

At the command line:

```
$ pip install tri.declarative
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv tri.declarative
$ pip install tri.declarative
```

10.2 API documentation

tri_declarative.**declarative**(*member_class=None*, *parameter='members'*, *add_init_kwargs=True*, *sort_key=None*, *is_member=None*)

Class decorator to enable classes to be defined in the style of django models. That is, @declarative classes will get an additional argument to constructor, containing a dict with all class members matching the specified type.

Parameters

- **member_class** (*class*) – Class(es) to collect
- **is_member** (*(object) -> bool*) – Function to determine if an object should be collected
- **parameter** (*str*) – Name of constructor parameter to inject
- **add_init_kwargs** (*bool*) – If constructor parameter should be injected (Default: True)
- **sort_key** (*(object) -> object*) – Function to invoke on members to obtain ordering (Default is to use ordering from *creation_ordered*)

tri_declarative.**evaluate_recursive_strict**(*func_or_value*, *__signature=None*, ***kwargs*)
Like *evaluate_recursive* but won't allow un-evaluated callables to slip through.

tri_declarative.**getattr_path**(*obj*, *path*, *default=<object object>*)
Get an attribute path, as defined by a string separated by '__'. getattr_path(foo, 'a__b__c') is roughly equivalent to foo.a.b.c but will short circuit to return None if something on the path is None. If no default value is provided AttributeError is raised if an attribute is missing somewhere along the path. If a default value is provided that value is returned.

tri_declarative.**get_members**(*cls*, *member_class=None*, *is_member=None*, *sort_key=None*, *_parameter=None*)
Collect all class level attributes matching the given criteria.

Parameters

- **cls** – Class to traverse
- **member_class** (*class*) – Class(es) to collect
- **is_member** (*(object) -> bool*) – Function to determine if an object should be collected
- **sort_key** (*(object) -> object*) – Function to invoke on members to obtain ordering (Default is to use ordering from *creation_ordered*)

class tri_declarative.**Namespace**(**dicts*, ***kwargs*)

__call__(**args*, ***kwargs*)
Call self as a function.

__init__(**dicts*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

tri_declarative.**setattr_path**(*obj*, *path*, *value*)
Set an attribute path, as defined by a string separated by '__'. setattr_path(foo, 'a__b__c', value) is equivalent to "foo.a.b.c = value".

class tri_declarative.**Shortcut**(**dicts*, ***kwargs*)

tri_declarative.**with_meta**(*class_to_decorate=None*, *add_init_kwargs=True*)
Class decorator to enable a class (and it's sub-classes) to have a 'Meta' class attribute.

Parameters **add_init_kwargs** (*bool*) – Pass Meta class members to constructor

Return type class

10.3 History

- Make getattr_path more in line with the standard library getattr semantics

  If a default value is provided, return that on missing attributes

  If no default value is given, give a more detailed error message of what was missing

- Added the special case of the empty path returning the object

10.3.1 5.6.0 (2020-12-02)

- Fix corner case of class Meta failing to merge with None namespace values

10.3.2 5.5.0 (2020-08-21)

- Include tri.struct 4.x as possible requirement

10.3.3 5.4.1 (2020-06-34)

- Optimizations

10.3.4 5.4.0 (2020-04-16)

- Minor bug fix on trailing comma explanation TypeException
- Fix bug when nesting *@class_shortcut* with same name i sub classes
- Refactor code to separate modules to get better stack traces

10.3.5 5.3.0 (2020-04-01)

- Enable *@class_shortcut* to override baseclass shortcuts with the same name.
- Fix *@with_meta* failing on method declarations with *@staticmethod* declaration

10.3.6 5.2.0 (2020-02-28)

- The namespace merge is narrowed to only affect the @with_meta case.
- Handle calling *Namespace* with *call_target__attribute=None*

10.3.7 5.1.1 (2020-02-11)

- Improve namespace merge in @with_meta to not trip up @declarative

10.3.8 5.1.0 (2020-02-11)

- Fix @with_meta argument injector to merge namespaces

10.3.9 5.0.1 (2019-02-03)

- A minor update to the documentation generation to make it play nice with rST

10.3.10 5.0.0 (2019-01-30)

- Added private field to shortcuts: *__tri_declarative_shortcut_stack*. This is useful to be able to figure out a shortcut stack after the fact
- *get_callable_description* thought stuff that contained a lambda in its string representation was a lambda
- **Removed all deprecated APIs/behaviors:**
  - *creation_ordered*
  - The promotion of string values to keys in *Namespace*
- Much improved error messages

10.3.11 4.0.1 (2019-10-23)

- Bugfix to correctly handle Namespace as callable/not callable depending on content

10.3.12 4.0.0 (2019-10-11)

- *get_meta()* now collects extra arguments in a *Namespace* to get consistent override behaviour.
- *should_show* no longer accepts a callable as a valid return value. It will assert on this, because it's always a mistake.
- Added *evaluate_strict* and *evaluate_recursive_strict* that will not accept callables left over after the evaluation. If possible prefer these methods because they will stop the user of your library from making the mistake of not matching the given signature and ending up with an unevaluated callable in the output.

10.3.13 3.1.0 (2019-06-28)

- Fixed issues when Namespace contained a key called any of items, values, keys, or get
- Removed sorting on Namespace kwargs that isn't needed in python 3 anymore. The sorting also destroys the given order which can be surprising
- Removed old obsolete functions collect_namespaces, extract_subkeys, and setdefaults

10.3.14 3.0.0 (2019-06-10)

- Renamed module from *tri.declarative* to *tri_declarative*. This is a breaking change
- Dropped support for python2

10.3.15 2.0.0 (2019-04-12)

- Fixed *get_signature* cache to not pollute struct-like dicts
- New call_target semantics for class method shortcuts, this is a potential breaking change

10.3.16 1.2.1 (2019-13-15)

- Improved documentation output

10.3.17 1.2.0 (2019-13-14)

- Add get_members function to enable reuse of @declarative attribute collection
- Add @class_shortcut decorator to enable @with_meta aware class shortcuts

10.3.18 1.1.0 (2018-11-22)

- Added *generate_rst_docs* function.

10.3.19 1.0.6 (2018-09-28)

- *Shortcut* is now a special case when merging *Namespace* objects. When already in a Namespace, a Shortcut now get overwritten by *setitem_path()*, not merged into the written value.

10.3.20 1.0.5 (2018-09-21)

- Fix broken handling of empty key

10.3.21 1.0.4 (2018-09-21)

- Cleanup Namespace path logic and make sure it is symmetrical and tested.
- Added deprecation warning on string to dict promotion on namespace merge.

1.0.3 (2018-06-26)

- Fixed release functionality

1.0.2 (2018-06-18)

- Don't support *RefinableObject* in *evaluate_recursive*. This was a mistake.

1.0.1 (2018-06-15)

- Support *RefinableObject* in *evaluate_recursive*.

1.0.0 (2018-05-23)

- Cleanup deprecation warnings from inspect.getargspec

0.34.0 (2017-08-21)

- Fix bug in 0.33.0 when promoting callable to *Namespace*.

0.33.0 (2017-08-21)

- Fix bug when promoting callable to *Namespace*.
- Fix handling of *EMPTY* marker.

0.32.0 (2017-07-04)

- Added promoting callable namespace members to *Namespace* with *call_target* in *setdefaults_path*.

0.31.0 (2017-06-15)

- Improve *sort_after* to allow more combinations of *after*=... specifications. e.g. by name of an entry also moved by spec.
- Changed name of first parameter of *setdefaults_path* to *__target__* to avoid collitions with namespace parameters.
- Added *RefinableObject* base for reuse by classes wanting to be able to be configured via constructor kwarg parameters in a declarative fashion. (The namespace of possible constructor overrides are declared with *Refinable()* for values and the decorator *@refinable* for methods.
- Added first incarnation of crawling the definitions to recursively find available parameters on objects and their aggregates.
- Added *Shortcut* abstraction to be able to find pre-defined set of overrides of *RefinableObject* classes.

0.30.0 (2017-02-10)

- *evaluate* and *evaluate_recursive* also works for methods as well as for functions.

0.29.0 (2016-09-12)

- Fixed loop detection in flatten for *Namespace*s. This resulted in data corruption.

0.28.0 (2016-07-15)

- Added *Namespace* subclass of *tri.struct.Struct* to explicit capture the path splitting semantics. (And added method for flattening a *Namespace* back to path notation.)

0.27.0 (2016-07-13)

- Fix bug in *evaluate* signature detection with optional arguments. (*lambda a, b=17: a+b* was correctly matched but *lambda b, a=17: a+b* was not)

0.26.0 (2016-05-06)

- Added *EMPTY* marker to *setdefaults_path* to avoid mixup when empty dict is provided in function defaults.

0.25.0 (2016-04-28)

- Added @dispatch decorator

0.24.0 (2016-04-20)

- Fix bug in *setdefault_path* tripping up on key ordering.
- Dropped *namespace_factory* keyword argument to *setdefaults_path* not likely ever beeing used.

0.23.0 (2016-04-15)

- *setdefaults_path* now accepts multiple default dicts. (To simplify the pattern of shortcuts in tri.form, tri.query and tri.table where we now will end up with: *new_kwargs = setdefaults_path(Struct(), kwargs, dict(...))*

0.22.0 (2016-03-24)

- *sort_after()* should produce an error when attempting to sort after non-existant keys
- Tweaked namespace merge in *setdefaults_path*

0.21.0 (2016-03-01)

- Fix corner case in collect_namespaces where one parameter imply a value and others imply a namespace.
- Added *setdefaults_path* helper with __ namespace traversal.

0.20.0 (2016-02-29)

- Added *assert_kwargs_not_empty* convenience function.
- Improved documentation.

0.19.0 (2016-01-12)

- When making instances of a class decorated with *@declarative* the declared values are copied (shallow) before being passed to *__init__*.
- Instances will get an own copy of the declared attributes written to their *__dict__*

10.4 Credits

- Johan Lübcke <johan.lubcke@trioptima.com>
- Anders Hovmöller <anders.hovmoller@trioptima.com>

10.5 Contributing

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

Issues, feature requests, etc are handled on github.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Index

Symbols

D

E

G

N

S

T

W