Flask-RESTful-DRY Documentation

Release 0.3

Bruce Frederiksen

March 04, 2016

Contents

Contents:

CHAPTER 1

flask.ext.dry.model

This has the `Model` class and validation code that is added to the Flask-SQLAlchemy model.

1.1 flask.ext.dry.model.model

1.2 Other Modules

1.2.1 flask.ext.dry.model.columns

1.2.2 flask.ext.dry.model.validation

1.2.3 flask.ext.dry.model.utils

CHAPTER 2

flask.ext.dry.api

This has the API infrastructure code used to build apis with DRY.

2.1 Intent

This code increases the ability to practice DRY coding (Do not Repeat Yourself). It does this by providing three basic capabilities:

1. Support for declarative programming.
2. Support for dynamically creating a set of classes multiple times (like a cookie cutter) containing different declarations for different specific use cases.
3. Support for implementing HTTP methods from a set of standard re-usable *steps*.

   This turns procedural programming into a declarative form and makes it more re-usable.

2.2 Declarative Programming

One way to raise the bar on DRY is by moving from procedural programming to declarative programming.

Procedural programming specifies how something is to be done indirectly, through a series of executable statements.

Declarative programming specifies how something is to be done by stating it directly. This allows us to write one set of functions that are instructed by these declarations on what is needed for each specific use case. Thus, one set of functions may be re-used for a much wider variety of use cases. The declarations allow extensive parameterization, or configuration, of the re-usable functions so that the code in these functions does not need to be repeated for each use case. This gives us our DRY goal.

2.2.1 Declarations as Class Variables

In Python, these declarations are made by setting class variables on classes. A common library can provide different common patterns of code as different base classes. These base classes have default declarations that can be overridden by their derived classes.

Many of these declarations are tuples of values and the derived class needs to be able to manipulate the default set of values by modifying them (e.g., adding more values, or removing values) rather than completely replacing them. Thus, the derived class needs access to its base class' attributes during the definition of the derived class:

class base: foo = ('a', 'b')

class derived(base): foo += ('c', 'd')

This capability is provided by the declarative metaclass.

2.3 Re-Usable Steps

HTTP methods must do many things: check various request headers, perform database accesses, validate user data, possibly update the database, and create a response with various response headers.

The number of combinations of these things is quite large, making it hard to imagine how to have a one-size-fits-all (DRY) HTTP method.

But if the processing required by the HTTP method is broken down into small steps, it becomes clear that many of these steps can be shared by several HTTP methods:

- Process conditional GET headers (If-None-Match, If-Modified-Since)
- Process conditional update headers (If-Match, If-Unmodified-Since)
- Check CSRF token
- Check Accept header
- Check Content-Type header
- Check authorization
- Get 1 row from database
- Get several rows from database
- Insert 1 row
- Update 1 row
- Delete 1 row
- Unpack JSON request body
- Get current user
- Set ETag response header
- Set Last-Modified response header
- Set caching response headers
- Set Location response header
- Validate User Data
- Check for and report errors
- Filter allowed columns
- Add HATEOAS links
- Convert response to JSON

Looking at it like this, the HTTP method itself could be specified as simply a sequence of these standard steps.

So we have changed HTTP methods from being specified procedurally (as executable statements), to being specified declaratively (as a simple tuple of these steps).

2.3.1 Flask-RESTful Resources

How would we apply this to Flask-RESTful Resources, so that we can specify template resources?

Step 1: Combining HTTP Methods

The first step is to combine the various HTTP methods into a single `_run` method that can be re-used by all to run the steps for that method. This looks like:

```
>>> from flask.ext.dry.api.class_init import declarative
```

```
>>> class item_url(metaclass=declarative):
...     get_steps = ('a', 'c', 'd')
...     put_steps = ('a', 'b', 'c', 'e')
...     delete_steps = ('b', 'd', 'e')
...     def get(self, *keys):
...         return self._run(self.get_steps, *keys)
...     def put(self, *keys):
...         return self._run(self.put_steps, *keys)
...     def delete(self, *keys):
...         return self._run(self.delete_steps, *keys)
...     def _run(self, steps, *keys):                   # DRY HTTP Method
...         self.keys = keys
...         for step in steps:
...             print("doing", step)
...         return 'response'
```

Note: This would also need to be derived from the Resource class in flask.ext.restful.

This could then be used for different item urls, which could each modify the steps needed for that use case:

```
>>> class venue_item(item_url):
...     get_steps += ('f', 'g')
```

```
>>> class zipcode_item(item_url):
...     put_steps += ('w', 'y')
```

Note the addition of the ‘f’ and ‘g’ steps for venues vs. zipcodes:

```
>>> venue_item().get(24)
doing a
doing c
doing d
doing f
doing g
'response'
```

```
>>> zipcode_item().get('33123')
doing a
doing c
doing d
'response'
```

And the addition of the ‘w’ and ‘y’ steps for zipcodes vs. venues:

```
>>> venue_item().put(24)
doing a
doing b
doing c
doing e
'response'
```

```
>>> zipcode_item().put('33123')
doing a
doing b
doing c
doing e
doing w
doing y
'response'
```

Steps are Just Functions

Each step ends up as a function that takes the above objects as a "context". This same context is passed to each of the steps run for the HTTP method. The context provides two things:

1. A way to pass values from one step to the next. The url parameters are set on the context by the `_run` method before the steps are run, and the final step sets a "response" attribute on the object, which is returned from the `_run` method.
2. Step declarations, to make each step configurable.

Step 2: Ordering of Steps

The steps need to be run in the proper sequence. When derived classes (like `venue_item` and `zipcode_item` above) add steps to their base class' list, they are placed at the end of the list of steps. This isn't necessarily their proper place in the list.

To solve this problem, the step functions have "needs" and "provides" attributes on them to identify what they are expecting, and what they are providing within the context that all of them share. These are used to automatically put the steps into the proper order before they are run.

The "needs" and "provides" attributes are added by a `step` decorator.

```
>>> from flask.ext.dry import step
```

Consider the following steps:

```
>>> @step(needs='keys', provides='row,output_row,etag,last_modified')
... def get_row(context):
...     context.output_row = context.row = "row for " + str(context.keys)
...     print("get_row: got keys:", context.keys, "providing:", context.row)
...     context.etag = "etag for row " + str(context.keys)
...     context.last_modified = "last_modified for row " + str(context.keys)
```

```
>>> @step(needs='etag,last_modified', provides='_checked')
... def conditional_get_check(context):
...     print("conditional_get_check: for etag:", context.etag,
...           "last_modified:", context.last_modified)
```

```
>>> @step(needs='etag,last_modified', provides='_checked')
... def conditional_update_check(context):
...     print("conditional_update_check: for etag:", context.etag,
...           "last_modified:", context.last_modified)
```

```
>>> @step(needs='roles,row', provides='_checked')
... def authorize(context):
...     print("authorize: got roles:", context.roles,
...           "for row:", context.row)
```

```
>>> @step(needs='output_row,columns,_checked', provides='output_row')
... def filter(context):
...     ans = "filtered " + context.output_row
...     print("filter: got row:", context.output_row,
...           "filtered by:", context.columns,
...           "providing:", ans)
...     context.output_row = ans
```

```
>>> @step(needs='row,_checked', provides='modified_row')
... def modify(context):
...     context.modified_row = "modified " + context.row
...     print("modify: got row:", context.row,
...           "providing:", context.modified_row)
```

```
>>> @step(needs='modified_row,keys', provides='_done')
... def update(context):
...     print("update: updating keys:", context.keys,
...           "to row:", context.modified_row)
```

```
>>> @step(needs='keys,_checked', provides='_done')
... def delete(context):
...     print("delete: deleting keys:", context.keys)
```

```
>>> @step(needs='output_row,_checked,_done', provides='output')
... def output_row(context):
...     print("output_row: outputting row:", context.output_row)
...     context.output = context.output_row
```

```
>>> @step(needs='_checked,_done', provides='output')
... def no_output(context):
...     print("no_output: outputting None")
...     context.output = None
```

```
>>> @step(needs='status,output', provides='response')
... def create_response(context):
...     context.response = context.status, context.output
...     print("create_response: returning:", context.response)
```

These are ordered by `generate_steps`:

```
>>> from flask.ext.dry.api.step_utils import generate_steps
```

Which takes a context to verify that it has whatever the steps need to start out.

```
>>> class dummy: pass
>>> context = dummy()
>>> context.roles = ('premium_member', 'management')
>>> context.status = 200
>>> context.keys = (42,)
```

It also wants something to identify the context in errors:

```
>>> context.url_class_name = 'dummy'
>>> context.location = 'doctest'
```

And what attributes are not currently present in the context, but will be provided later:

```
>>> context.provided_attributes = ()
```

The unordered steps are in the `step_fns` attribute of the context:

```
>>> context.step_fns = (authorize, conditional_get_check, get_row,
...                     create_response, output_row)
```

We can now get the steps to execute for this context:

```
>>> for _step in generate_steps(context):
...     print(_step.__name__)
get_row
authorize
conditional_get_check
output_row
create_response
```

Note: This step ordering process actually happens once at program startup. The steps are specified in a `step_fns` attributes by the classes, and these are converted to a `steps` attribute that has all of these steps in the proper order.

Step 3: Step Declarations

The attrs class is used to inject both the steps list, and their declarations into the different HTTP methods.

```
>>> from flask.ext.dry import attrs, extend
```

Thus, rather than `get_steps`, `put_steps`, etc; we have `get_attrs`, `put_attrs`, etc. So now, our item_url class looks like this:

```
>>> class item_url(metaclass=declarative):
...     # boilerplate:
...     url_class_name = 'item_url'
...     location = 'doctest'
...     provided_attributes = ()
...     debug = 0
...
...     roles = ()  # default declaration for authorize step
...     step_fns = (get_row, authorize, create_response)
...     get_attrs = attrs(status=200,     # declaration for create_response
...                       step_fns=extend(conditional_get_check,
...                                       output_row,
...                                      ))
...     put_attrs = attrs(status=204,     # declaration for create_response
...                       step_fns=extend(conditional_update_check,
...                                       modify,
...                                       update,
...                                       no_output,
...                                      ))
...     delete_attrs = attrs(status=204, # declaration for create_response
...                          step_fns=extend(conditional_update_check,
...                                          delete,
...                                          no_output,
...                                         ))
...
...     def get(self, *keys):
...         return self._run(self.get_attrs, *keys)
...     def put(self, *keys):
...         return self._run(self.put_attrs, *keys)
...     def delete(self, *keys):
...         return self._run(self.delete_attrs, *keys)
...     def _run(self, method_attrs, *keys):
...         self.keys = keys
...         method_attrs.copy_into(self)
...         for step in generate_steps(self):
...             step(context=self)
...         return self.response
```

And our derived classes look like this:

```
>>> class venue_item(item_url):
...     put_attrs.roles = \
...     delete_attrs.roles = extend('cma', 'management')
```

```
>>> class zipcode_item(item_url):
...     get_attrs.columns = ('a', 'b', 'c')   # declaration for filter
...     get_attrs.step_fns = extend(filter)
...     put_attrs.roles = \
...     delete_attrs.roles = extend('management')
```

And here's what happens when we run them:

```
>>> venue_item().get(24)
get_row: got keys: (24,) providing: row for (24,)
authorize: got roles: () for row: row for (24,)
conditional_get_check: for etag: etag for row (24,) last_modified: last_modified for row (24,)
output_row: outputting row: row for (24,)
create_response: returning: (200, 'row for (24,)')
(200, 'row for (24,)')
```

Notice the extra filter step on the next one with its `columns` declaration:

```
>>> zipcode_item().get('33123')
get_row: got keys: ('33123',) providing: row for ('33123',)
authorize: got roles: () for row: row for ('33123',)
conditional_get_check: for etag: etag for row ('33123',) last_modified: last_modified for row ('3312
filter: got row: row for ('33123',) filtered by: ('a', 'b', 'c') providing: filtered row for ('33123
output_row: outputting row: filtered row for ('33123',)
create_response: returning: (200, "filtered row for ('33123',)")
(200, "filtered row for ('33123',)")
```

And the change in response status and different roles on the next two as an example of step declarations:

```
>>> venue_item().put(24)
get_row: got keys: (24,) providing: row for (24,)
authorize: got roles: ('cma', 'management') for row: row for (24,)
conditional_update_check: for etag: etag for row (24,) last_modified: last_modified for row (24,)
modify: got row: row for (24,) providing: modified row for (24,)
update: updating keys: (24,) to row: modified row for (24,)
no_output: outputting None
create_response: returning: (204, None)
(204, None)
```

```
>>> zipcode_item().put('33123')
get_row: got keys: ('33123',) providing: row for ('33123',)
authorize: got roles: ('management',) for row: row for ('33123',)
conditional_update_check: for etag: etag for row ('33123',) last_modified: last_modified for row ('33
modify: got row: row for ('33123',) providing: modified row for ('33123',)
update: updating keys: ('33123',) to row: modified row for ('33123',)
no_output: outputting None
create_response: returning: (204, None)
(204, None)
```

2.4 Creating Classes Dynamically

Our final challenge concerns the granularity of Flask-RESTful Resources. Each Resource encapsulates all of the HTTP methods to a single URL.

But there is often more than one URL associated with a single conceptual entity. For example, each type of item may have the following URLs:

/api/items To access lists of items (GET) and create new ones (POST).

/api/items/metadata To get metadata information on what columns are allowed on POST and what validation rules to apply (GET).

/api/items/<int:item_id> To access/update/delete a single item (GET, PUT, DELETE).

/api/items/<int:item_id>/metadata To get metadata information on what columns are allowed on PUT and what validation rules to apply (GET).

It would be nice to be able to declare a single conceptual resource class that would take care of all of these URL resources automatically.

To do this, the conceptual resource class must create the url classes dynamically. This is easily done with the new_class function in the standard Python library:

```
>>> from types import new_class
```

Now we can write a `cookie_cutter` class that dynamically creates derived classes and sets attributes on them (see attrs):

```
>>> class Declarative_Resource(metaclass=declarative):
...     url_classes = ()   # default declaration
...
...     @classmethod
...     def resource_init(cls):
...         cls._url_resources = {}   # Keep the url resource classes.
...         for url_class_name in cls.url_classes:
...
...             # Derive the new url resource class from this one, so that
...             # declarations on this class are inherited by all url
...             # resource classes.
...             new_url_class = new_class("{}__{}".format(cls.__name__,
...                                                       url_class_name),
...                                       bases=(cls,))
...
...             # Copy the attributes for this class into the new class.
...             getattr(cls, url_class_name).copy_into(new_url_class)
...
...             cls._url_resources[url_class_name] = new_url_class
```

```
...
...     @classmethod
...     def dump(cls):
...         for name, url_class in sorted(cls._url_resources.items()):
...             print("{}:".format(url_class.__name__))
...             for attr in sorted(getattr(cls, name)._names):
...                 print("    {} = {}"
...                       .format(attr, getattr(url_class, attr)))
```

And write several usage patterns as cookie_cutters:

```
>>> class Item_Resource(Declarative_Resource):
...     url_classes += ('collection', 'self')
...     collection = attrs(foo=10, bar=20)
...     self = attrs(foo=1, bar=2)
```

```
>>> class List_Resource(Declarative_Resource):
...     url_classes += ('list',)
...     list = attrs(foo=100, bar=200)
```

And finally use these patterns to create classes for specific use cases. These may override any of the attrs values in their base classes to cause the derived classes (that will be created later by cookie_cutter) to have different class variables:

```
>>> class venues(Item_Resource):
...     self.foo = 11
```

```
>>> class artists(Item_Resource):
...     collection.bar = 22
```

```
>>> class genres(List_Resource):
...     list.foo = 111
```

Now we initialize these classes:

```
>>> venues.resource_init()
>>> artists.resource_init()
>>> genres.resource_init()
```

And see what we ended up with:

```
>>> venues.dump()
venues__collection:
    bar = 20
    foo = 10
venues__self:
    bar = 2
    foo = 11
```

```
>>> artists.dump()
artists__collection:
    bar = 22
    foo = 10
artists__self:
    bar = 2
    foo = 1
```

```
>>> genres.dump()
genres__list:
    bar = 200
    foo = 111
```

2.5 Conclusion

The two techniques, *Re-Usable Steps* and *Creating Classes Dynamically*, are combined by nesting the method attrs for the *Step 3: Step Declarations* within the url resource `attrs` in the *Declarative_Resource*.

We'll start by adding the HTTP methods to the Declarative_Resource:

```
>>> class Declarative_Resource_2(Declarative_Resource):
...     # Boilerplate
...     debug = 0
...     provided_attributes = ()
...     url_class_name = 'bogus'
...     location = 'doctest'
...
...     # Default all HTTP methods to dis-allowed
...     get_attrs = put_attrs = delete_attrs = post_attrs = None
...
...     def get(self, *keys):
...         import sys
...         print("self", self, file=sys.stderr)
...         return self._run(self.get_attrs, *keys)
...     def put(self, *keys):
...         return self._run(self.put_attrs, *keys)
...     def delete(self, *keys):
...         return self._run(self.delete_attrs, *keys)
...     def post(self, *keys):
...         return self._run(self.post_attrs, *keys)
...
...     def _run(self, method_attrs, *keys):
...         if method_attrs is None:
...             return 405, None
...         method_attrs.copy_into(self)
...         if keys:
...             self.keys = keys
...         for step in generate_steps(self):
...             step(context=self)
...         return self.response
```

This forms the top level in a three tier API class hierarchy. This class should work for all APIs!

Note: Currently, this class is called `DRY_Resource`.

Now we can define different kinds of templates that create different groups of URL Resources. To do the Item_Resource template will require a few more steps for the collection URL:

```
>>> @step(provides='rows')
... def get_rows(context):
...     context.rows = ('row 1', 'row 2')
...     print("get_rows: providing:", context.rows)
```

```
>>> @step(needs='rows', provides='etag,last_modified')
... def hash_etag_for_rows(context):
...     context.etag = 'hashed etag'
...     context.last_modified = None
...     print("hash_etag_for_rows: got rows:", context.rows,
...           "providing etag:", context.etag)
```

```
>>> @step(needs='rows,_checked,_done', provides='output')
... def output_rows(context):
...     print("output_rows: outputting rows:", context.rows)
...     context.output = dict(_response=context.rows)
```

```
>>> @step(provides='_done')
... def insert(context):
...     print("insert: inserting new row")
```

We combine the two techniques into a single Item_Resource class. This is an example of the middle of the three tiers of API classes. There only needs to be a few of these (currently Item_Resource and List_Resource).

```
>>> class Item_Resource_2(Declarative_Resource_2):
...     url_classes = extend('collection', 'self')
...
...     # default declarations for all URLs:
...     roles = ()
...     step_fns = (authorize, create_response)
...     get_attrs = attrs(status = 200,
...                       step_fns = step_fns + (conditional_get_check,))
...     update_step_fns = attrs(status = 204,
...                             step_fns = step_fns + (
...                                          conditional_update_check,
...                                          no_output,))
...
...     # This attrs is for the new collection URL class.
...     collection = attrs(row = None,
...                        get_attrs = get_attrs,
...                        # This attrs is for the post HTTP method.
...                        post_attrs = attrs(status=204,
...                                           step_fns = step_fns))
...     collection.get_attrs.step_fns = extend(get_rows,
...                                            hash_etag_for_rows,
...                                            output_rows)
...     collection.post_attrs.step_fns = extend(insert, no_output)
...
...     self = attrs(get_attrs = get_attrs,
...                  put_attrs = update_step_fns,
...                  delete_attrs = update_step_fns)
...     self.get_attrs.step_fns = extend(get_row, output_row)
...     self.put_attrs.step_fns = extend(modify, update)
...     self.delete_attrs.step_fns = extend(delete)
```

Note: In the interest of brevity, the metadata URLs have been omitted.

This class can then be used to define specific resource groups. This forms the final, third, tier of the API class hierarchy. There are lots of these! These are what we're trying to make as simple as possible.

```
>>> class Venues(Item_Resource_2):
...     self.put_attrs.roles = \
...     self.delete_attrs.roles = \
...     collection.post_attrs.roles = extend('cma', 'management')
```

```
>>> class Zipcodes(Item_Resource_2):
...     columns = ('a', 'b', 'c')    # declaration for filter
...     self.get_attrs.step_fns = \
...     collection.get_attrs.step_fns = extend(filter)
...     self.put_attrs.roles = \
...     self.delete_attrs.roles = extend('management')
...     collection.post_attrs = None # No posts allowed!
```

Initialize the third tier classes:

```
>>> Venues.resource_init()
>>> Zipcodes.resource_init()
```

Let's give these a try!

These first two are completely defaulted:

```
>>> Venues._url_resources['self']().get(42)
get_row: got keys: (42,) providing: row for (42,)
authorize: got roles: () for row: row for (42,)
conditional_get_check: for etag: etag for row (42,) last_modified: last_modified for row (42,)
output_row: outputting row: row for (42,)
create_response: returning: (200, 'row for (42,)')
(200, 'row for (42,)')
```

```
>>> Venues._url_resources['collection']().get()
get_rows: providing: ('row 1', 'row 2')
authorize: got roles: () for row: None
hash_etag_for_rows: got rows: ('row 1', 'row 2') providing etag: hashed etag
conditional_get_check: for etag: hashed etag last_modified: None
output_rows: outputting rows: ('row 1', 'row 2')
create_response: returning: (200, {'_response': ('row 1', 'row 2')})
(200, {'_response': ('row 1', 'row 2')})
```

This has changed the roles declaration for the authorize step.

```
>>> Venues._url_resources['collection']().post()
authorize: got roles: ('cma', 'management') for row: None
insert: inserting new row
no_output: outputting None
create_response: returning: (204, None)
(204, None)
```

This has an added filter step:

```
>>> Zipcodes._url_resources['self']().get('33123')
get_row: got keys: ('33123',) providing: row for ('33123',)
authorize: got roles: () for row: row for ('33123',)
conditional_get_check: for etag: etag for row ('33123',) last_modified: last_modified for row ('3312
filter: got row: row for ('33123',) filtered by: ('a', 'b', 'c') providing: filtered row for ('33123
output_row: outputting row: filtered row for ('33123',)
create_response: returning: (200, "filtered row for ('33123',)")
(200, "filtered row for ('33123',)")
```

Zipcodes has disabled the post method:

```
>>> Zipcodes._url_resources['collection']().post()
(405, None)
```

2.6 See Also

2.6.1 declarative

Problem

Python does not let derived class have access to their base classes while the derived class is being defined:

```
>>> class base:
...     foo = ('a', 'b')
```

```
>>> class derived(base):
...     foo += ('c', 'd')
Traceback (most recent call last):
    ...
NameError: name 'foo' is not defined
```

Solution

The `declarative` metaclass solves this:

```
>>> from flask.ext.dry.api.class_init import declarative
```

```
>>> class base(metaclass=declarative):
...     foo = ('a', 'b')
>>> class derived(base):
...     foo += ('c', 'd')
```

```
>>> derived.foo
('a', 'b', 'c', 'd')
```

Now methods defined on the base class may access these class variables as the declarations that tell them what to do:

```
>>> class base(metaclass=declarative):
...     foo = ('a', 'b')   # default values
...     def do_foo(self):
...         for i in self.foo:
...             print("doing", i)
>>> class derived1(base):
...     foo += ('c', 'd') # extend default values
>>> class derived2(base):
...     foo = ('x', 'y')   # replace default values
>>> derived1().do_foo()
doing a
doing b
doing c
doing d
>>> derived2().do_foo()
doing x
doing y
```

The declarative metaclass makes a deepcopy of each base class attribute referenced by the derived class. This prevents updates made in the derived class from corrupting the base class:

```
>>> class base(metaclass=declarative):
...     foo = ['a', 'b']
>>> class derived(base):
...     foo.extend(('c', 'd'))
>>> derived.foo
['a', 'b', 'c', 'd']
>>> base.foo
['a', 'b']
```

2.6.2 attrs

Attrs, allow you to define a set of attributes that will be applied later to a class or object.

```
>>> from flask.ext.dry import attrs
```

Basics

Attrs are created with keyword arguments:

```
>>> a = attrs(foo=1, bar=2)
```

They may be copied into a class:

```
>>> class c: pass
>>> a.copy_into(c)
>>> c.foo
1
>>> c.bar
2
```

Or an object:

```
>>> class d:
...     # for lookup modifier, later
...     def get_expanded_attr(self, attr): return getattr(self, attr)
>>> obj = d()
>>> a.copy_into(obj)
>>> obj.foo
1
>>> obj.bar
2
```

In the latter case, the class isn't changed:

```
>>> d.foo
Traceback (most recent call last):
    ...
AttributeError: type object 'd' has no attribute 'foo'
```

They may also be changed, like any other object:

```
>>> a.foo = 100
>>> obj2 = d()
>>> a.copy_into(obj2)
>>> obj2.foo
```

```
100
>>> obj2.bar
2
```

Finally, attrs make a deepcopy of all values assigned to them, as well as the values copied into the receiving class or object so that one object can not corrupt the value of another.

```
>>> l = [1, 2, 3]
>>> a = attrs(foo=l)     # foo is a deepcopy of l
>>> obj4 = d()
>>> a.copy_into(obj4)    # obj4.foo is a deepcopy of a.foo
```

```
>>> a.foo.append(4)
>>> obj5 = d()
>>> a.copy_into(obj5)    # obj5.foo is a second deepcopy of a.foo
```

```
>>> obj4.foo.append(5)
>>> obj5.foo.append(6)
>>> obj4.foo
[1, 2, 3, 5]
>>> obj5.foo
[1, 2, 3, 4, 6]
>>> a.foo
[1, 2, 3, 4]
>>> l
[1, 2, 3]
```

Modifiers

Normally, the attrs attributes replace any attributes of the same name already in the class or object.

```
>>> a = attrs(foo=1, bar=2)
>>> obj6 = d()
>>> obj6.bar = 400
>>> obj6.baz = 500
>>> a.copy_into(obj6)
>>> obj6.foo         # set by a.copy_into
1
>>> obj6.bar         # overridden by a.copy_into
2
>>> obj6.baz         # untouched by a.copy_into
500
```

But attrs may also have modifiers as attribute values. When these are copied into a class or object, they modify any pre-existing value.

Two of these operate on tuples: `extend` and `remove`.

```
>>> from flask.ext.dry import extend, remove
```

```
>>> a = attrs(foo=extend(4, 5), bar=remove(4, 6))
>>> obj5 = d()
>>> obj5.foo = (1, 2, 3)
>>> obj5.bar = (4, 5, 6)
>>> a.copy_into(obj5)
>>> obj5.foo
(1, 2, 3, 4, 5)
```

```
>>> obj5.bar
(5,)
```

Finally, `lookup` can be used to lookup a value at the time that the copy_into is done:

```
>>> from flask.ext.dry import lookup
```

```
>>> a = attrs(foo=lookup('x'), bar=lookup('y.z'))
>>> obj5 = d()
>>> obj5.x = 'first x value'
>>> obj5.y = d()
>>> obj5.y.z = 'first x.z value'
>>> a.copy_into(obj5)
>>> obj5.foo
'first x value'
>>> obj5.bar
'first x.z value'
```

```
>>> obj5.x = 'second x value'
>>> obj5.y.z = 'second x.z value'
>>> a.copy_into(obj5)
>>> obj5.foo
'second x value'
>>> obj5.bar
'second x.z value'
```

You may create your own modifiers by deriving from `modifier`. Read the source code to see how to do this.

CHAPTER 3

Source Repository

Mercurial is used for the source code repository. The repository is hosted on BitBucket.org here.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search