Constant Model Documentation

Release 1.1.3

Warren A. Smith

Feb 22, 2022

Contents

Contents

CHAPTER 1

Introduction

StaticModel is a simple framework for modeling collections of multi-valued constants.

1.1 What is a collection of multi-valued constants?

To explain what it is, it first helps to understand what it is not.

The data that our software processes is often represented as collections of multi-valued objects. These collections are often stored in a database and frameworks such as the Django ORM can be used to define and access them. Each item in the collection may have one or more unique identifiers, but the code should not know or care about any specific item in the collection.

Some collections of multi-valued objects *ARE* tightly integrated with the code, in that the code uses specific values within a collection to affect its behavior. These collections should *NOT* be stored in a database. Doing so introduces many problems into the development, maintainence, and deployment of the code. These collections should be defined statically and used throughout the code via those definitions. These are the collections that **StaticModel** is used to define.

Can't we just use built-in collection types for this?

The simple answer is: **Yes**.

However, as our code evolves and the collections become more numerous, gain more members, gain more values, and require custom behavior, using the built-in types leads to code that is harder to write, harder to read, verbose, repetitive, and ugly.

StaticModel was created to solve these problems.

But before we delve into the features of StaticModel, lets explore how we might implement a constant collection using built-in types.

1.2 Using built-in types for constant collections

```
>>> # Prettier collection display
>>> from pprint import pprint as pp
>>>
>>>
>>> ANIMAL_TYPE_DOG = 1
>>> ANIMAL_TYPE_AFRICAN_SWALLOW = 2
>>>
>>>
>>> class Animal:
...     def __init__(self, name, animal_type):
...         self.name = name
...         self.type = animal_type
...
...     def __repr__(self):
...         return "<{}: name={!r}, type={!r}>".format(
...             self.__class__.__name__, self.name, self.type)
...
...     def walk(self):
...         if self.type == ANIMAL_TYPE_AFRICAN_SWALLOW:
...             return '{}: Ok, but I would rather fly'.format(self.name)
...         else:
...             return '{}: Walking...'.format(self.name)
...
...
>>> animals = [Animal('Spot', ANIMAL_TYPE_DOG), Animal('Coco', ANIMAL_TYPE_AFRICAN_
↪SWALLOW)]
>>>
>>> pp(animals)
[<Animal: name='Spot', type=1>, <Animal: name='Coco', type=2>]
>>> pp([animal.walk() for animal in animals])
['Spot: Walking...', 'Coco: Ok, but I would rather fly']
```

Naming and using constants this way is common in many languages. The upper case names signify to the developer that the values cannot or should not be changed once the initial values are assigned. The shared prefix in the name establishes that these constants are associated with each other and our class definition. The code compares variables to the constants to affect its behavior.

However, the shared prefix method of associating these constants with each other is archaic in a relatively modern language like Python. Using some sort of Python construct that provides a shared namespace would be much better.

A simple way to solve the namespace issue is to use the built-in `class` statement.

```
>>> class AnimalType:
...     DOG = 1
...     AFRICAN_SWALLOW = 2
```

This cuts down on the repetition in the constant declarations and puts the constants in a shared namespace. They can be referenced like so:

```
>>> AnimalType.DOG
1
>>> getattr(AnimalType, 'DOG')
1
```

This is better, and can be used in a lot of simple use cases.

However, since Python 3.4, a more sophisticated solution is the built-in `Enum` type.

```
>>> from enum import Enum
>>>
>>> class AnimalType(Enum):
...     DOG = 1
...     AFRICAN_SWALLOW = 2
```

`Enum` provides an api for defining, referencing, and de-referencing a constant collection, as well as supporting iteration.

Reference a member:

```
>>> AnimalType.DOG
<AnimalType.DOG: 1>
>>> getattr(AnimalType, 'DOG')
<AnimalType.DOG: 1>
```

De-reference a member:

```
>>> AnimalType(2)
<AnimalType.AFRICAN_SWALLOW: 2>
>>> try:
...     AnimalType(3)
... except ValueError as e:
...     print(e)
3 is not a valid AnimalType
```

Iterate over the collection:

```
>>> list(AnimalType)
[<AnimalType.DOG: 1>, <AnimalType.AFRICAN_SWALLOW: 2>]
```

The `Animal.walk()` method we defined earlier really belongs on the `AnimalType` class, as it "owns" the data that is being used in the method.

Behavior for `Enum` members can be defined directly on the `Enum` sub-class.

```
>>> class AnimalType(Enum):
...     DOG = 1
...     AFRICAN_SWALLOW = 2
...
...     def walk(self, animal):
...         msg_prefix = '{}: I am a {}'.format(animal.name, self.name.title().
↪replace('_', ' '))
...         if self is AnimalType.AFRICAN_SWALLOW:
...             return '{}: Ok, but I would rather fly'.format(msg_prefix)
...         else:
...             return '{}: Walking...'.format(msg_prefix)
...
>>> class Animal:
...     def __init__(self, name, animal_type):
...         self.name = name
...         self.type = animal_type
...
...     def __repr__(self):
...         return "<{}: name={!r}, type={!r}>".format(
...             self.__class__.__name__, self.name, self.type)
```

(continues on next page)

(continued from previous page)

```
...
...     def walk(self):
...         return self.type.walk(self)
...
...
>>> animals = [Animal('Spot', AnimalType.DOG), Animal('Coco', AnimalType.AFRICAN_
↪SWALLOW)]
>>>
>>> pp(animals)
[<Animal: name='Spot', type=<AnimalType.DOG: 1>>,
 <Animal: name='Coco', type=<AnimalType.AFRICAN_SWALLOW: 2>>]
>>> pp([animal.walk() for animal in animals])
['Spot: I am a Dog: Walking...',
 'Coco: I am a African Swallow: Ok, but I would rather fly']
```

Now, suppose we want to associate an additional value with our constant, beyond the `.name` and `.value` fields that `Enum` provides?

There are a few ways to do this, but the most straightforward is to use the built-in `tuple` type for the value.

```
>>> class AnimalType(Enum):
...     # (type_id, description)
...     DOG = (1, "Man's best friend")
...     AFRICAN_SWALLOW = (2, 'Coconut carrier')
...
...     def walk(self, animal):
...         msg_prefix = '{}: I am a {}'.format(animal.name, self.name.title().
↪replace('_', ' '))
...         if self is AnimalType.AFRICAN_SWALLOW:
...             return '{}: Ok, but I would rather fly'.format(msg_prefix)
...         else:
...             return '{}: Walking...'.format(msg_prefix)
```

We can now use the index of the tuple to access the value fields:

```
>>> AnimalType.DOG.value[0]
1
>>> AnimalType.AFRICAN_SWALLOW.value[1]
'Coconut carrier'
```

Iteration still works:

```
>>> pp(list(AnimalType))
[<AnimalType.DOG: (1, "Man's best friend")>,
 <AnimalType.AFRICAN_SWALLOW: (2, 'Coconut carrier')>]
```

Unfortunately, we have lost the ability to de-reference via the number we have designated `type_id`:

```
>>> try:
...     AnimalType(1)
... except ValueError as e:
...     print(e)
1 is not a valid AnimalType
```

In order for the de-reference to work, we need to use the entire tuple:

```
>>> AnimalType((1, "Man's best friend"))
<AnimalType.DOG: (1, "Man's best friend")>
```

This is undesirable. The 'description' should be opaque to our code, and only exist in one place.

To regain proper de-reference capability, we need to add a custom class method. Since we will want to re-use this code, we will go ahead and abstract the custom de-reference method into a base class that `AnimalType` can sub-class.

```
>>> class MultiEnum(Enum):
...     @classmethod
...     def lookup(cls, first_field_value):
...         for member in cls:
...             if member.value[0] == first_field_value:
...                 return member
...
...         raise ValueError('{} is not a valid {}'.format(first_field_value, cls.__
↪name__))
...
>>> class AnimalType(MultiEnum):
...     # (type_id, description)
...     DOG = (1, "Man's best friend")
...     AFRICAN_SWALLOW = (2, 'Coconut carrier')
...
...     def walk(self, animal):
...         msg_prefix = '{}: I am a {}'.format(animal.name, self.name.title().
↪replace('_', ' '))
...         if self is AnimalType.AFRICAN_SWALLOW:
...             return '{}: Ok, but I would rather fly'.format(msg_prefix)
...         else:
...             return '{}: Walking...'.format(msg_prefix)
...
>>> AnimalType.lookup(1)
<AnimalType.DOG: (1, "Man's best friend")>
>>> try:
...     AnimalType.lookup(3)
... except ValueError as e:
...     print(e)
3 is not a valid AnimalType
```

This is better. We can now de-reference the members like before.

However, accessing the member elements by index is not ideal, in that it decreases the readability of the code that uses them.

We can solve that by using the built-in `collections.namedtuple` type:

```
>>> from collections import namedtuple
>>>
>>> AnimalTypeAttrs = namedtuple('AnimalTypeAttrs', ('type_id', 'description'))
>>>
>>> class AnimalType(MultiEnum):
...     DOG = AnimalTypeAttrs(1, "Man's best friend")
...     AFRICAN_SWALLOW = AnimalTypeAttrs(2, 'Coconut carrier')
...
...     def walk(self, animal):
...         msg_prefix = '{}: I am a {}'.format(animal.name, self.name.title().
↪replace('_', ' '))
...         if self is AnimalType.AFRICAN_SWALLOW:
```

(continues on next page)

(continued from previous page)

```
...             return '{}: Ok, but I would rather fly'.format(msg_prefix)
...         else:
...             return '{}: Walking...'.format(msg_prefix)
...
>>> AnimalType.lookup(1)
<AnimalType.DOG: AnimalTypeAttrs(type_id=1, description="Man's best friend")>
>>> AnimalType.DOG.value.type_id
1
>>> AnimalType.DOG.value.description
"Man's best friend"
```

This is better. We have added some complexity and repetition (having to define an additional class and instantiate it to create our member values), but code that references the member field values will be more readable.

Now, lets add some additional fields that will allow us to change behavior in the `.walk()` method to allow for common attributes that multiple members may share without adding additional branching (see the use of `.self.can_fly` in the `.walk()` method).

Suppose we also want to be able to do a lookup by one or more fields and get either a single member back, or a list of results.

Let's change our `MultiEnum` to provide this capability, using the Django ORM api as inspiration:

```
>>> class MultiEnum(Enum):
...     @classmethod
...     def filter(cls, **criteria):
...         results = []
...         for member in cls:
...             for field, value in criteria.items():
...                 if getattr(member.value, field) != value:
...                     break
...             else:
...                 results.append(member)
...
...         return results
...
...     @classmethod
...     def get(cls, **criteria):
...         results = cls.filter(**criteria)
...         if len(results) == 1:
...             return results[0]
...         elif len(results) == 0:
...             raise ValueError('Member does not exist for criteria')
...         else:
...             raise ValueError('Multiple members exist for criteria')
...
>>> AnimalTypeAttrs = namedtuple('AnimalTypeAttrs', (
...     'type_id', 'description', 'can_fly', 'domesticated'))
>>>
>>> class AnimalType(MultiEnum):
...     DOG = AnimalTypeAttrs(1, "Man's best friend", False, True)
...     AFRICAN_SWALLOW = AnimalTypeAttrs(2, 'Coconut carrier', True, False)
...     CAT = AnimalTypeAttrs(3, "Man's overloard", False, True)
...     PARROT = AnimalTypeAttrs(4, "Voice of a mute Pirate", True, True)
...
...     def walk(self, animal):
...         msg_prefix = '{}: I am a {}'.format(animal.name, self.name.title().
↪replace('_', ' '))
```

(continues on next page)

(continued from previous page)

```
...         if self.value.can_fly:
...             return '{}: Ok, but I would rather fly'.format(msg_prefix)
...         else:
...             return '{}: Walking...'.format(msg_prefix)
...
>>> pp(AnimalType.filter(domesticated=True))
[<AnimalType.DOG: AnimalTypeAttrs(type_id=1, description="Man's best friend", can_
↪fly=False, domesticated=True)>,
 <AnimalType.CAT: AnimalTypeAttrs(type_id=3, description="Man's overloard", can_
↪fly=False, domesticated=True)>,
 <AnimalType.PARROT: AnimalTypeAttrs(type_id=4, description='Voice of a mute Pirate',␣
↪can_fly=True, domesticated=True)>]
>>> AnimalType.get(can_fly=True, domesticated=True)
<AnimalType.PARROT: AnimalTypeAttrs(type_id=4, description='Voice of a mute Pirate',␣
↪can_fly=True, domesticated=True)>
>>> AnimalType.filter(description='')
[]
>>> try:
...     AnimalType.get(can_fly=True)
... except ValueError as e:
...     print(e)
Multiple members exist for criteria
>>> try:
...     AnimalType.get(description='')
... except ValueError as e:
...     print(e)
Member does not exist for criteria
>>> animals = [
...     Animal('Spot', AnimalType.DOG),
...     Animal('Coco', AnimalType.AFRICAN_SWALLOW),
...     Animal('Garfield', AnimalType.CAT),
...     Animal('Cotton', AnimalType.PARROT),
... ]
...
>>> pp(animals)
[<Animal: name='Spot', type=<AnimalType.DOG: AnimalTypeAttrs(type_id=1, description=
↪"Man's best friend", can_fly=False, domesticated=True)>>,
 <Animal: name='Coco', type=<AnimalType.AFRICAN_SWALLOW: AnimalTypeAttrs(type_id=2,␣
↪description='Coconut carrier', can_fly=True, domesticated=False)>>,
 <Animal: name='Garfield', type=<AnimalType.CAT: AnimalTypeAttrs(type_id=3,␣
↪description="Man's overloard", can_fly=False, domesticated=True)>>,
 <Animal: name='Cotton', type=<AnimalType.PARROT: AnimalTypeAttrs(type_id=4,␣
↪description='Voice of a mute Pirate', can_fly=True, domesticated=True)>>]
>>> pp([animal.walk() for animal in animals])
['Spot: I am a Dog: Walking...',
 'Coco: I am a African Swallow: Ok, but I would rather fly',
 'Garfield: I am a Cat: Walking...',
 'Cotton: I am a Parrot: Ok, but I would rather fly']
```

The new api in `MultiEnum` works pretty well, and should feel conceptually similar to anyone who has used the Django ORM. Obviously it lacks most of the features of that api, but will suffice for many use cases.

Now comes the tricky part. Suppose we add additional behavior and members. We *can* continue to use our parameterized imperative logic:

```
>>> AnimalTypeAttrs = namedtuple('AnimalTypeAttrs', (
...     'type_id', 'description', 'can_fly', 'domesticated', 'leg_count', 'likes_to_
↪fly',
```

(continues on next page)

(continued from previous page)

```
...     'can_swim', 'likes_to_swim'))
...
>>> class AnimalType(MultiEnum):
...     DOG = AnimalTypeAttrs(1, "Man's best friend", False, True, 4, False, True,␣
↪True)
...     AFRICAN_SWALLOW = AnimalTypeAttrs(2, 'Coconut carrier', True, False, 2, True,␣
↪False, False)
...     CAT = AnimalTypeAttrs(3, "Man's overloard", False, True, 4, False, True,␣
↪False)
...     PARROT = AnimalTypeAttrs(4, 'Voice of a mute Pirate', None, True, 2, True,␣
↪False, False)
...     CHICKEN = AnimalTypeAttrs(5, 'Disaster reporter', True, True, 2, False, False,
↪ False)
...     FISH = AnimalTypeAttrs(6, 'Searcher', False, False, 0, False, True, True)
...
...     def _get_message_prefix(self, animal):
...         return '{}: I am a {}'.format(animal.name, self.name.title().replace('_',
↪' '))
...
...     def walk(self, animal):
...         msg_prefix = self._get_message_prefix(animal)
...         if self.value.can_fly:
...             if self.value.likes_to_fly:
...                 return '{}: Ok, but I would rather fly'.format(msg_prefix)
...             else:
...                 return '{}: Cool! I like to walk, even though I can fly'.
↪format(msg_prefix)
...         elif self.value.can_fly is None and self.value.likes_to_fly:
...             return "{}: I'm walking because my wings have been clipped. I prefer␣
↪to fly".format(
...                 msg_prefix)
...         elif self.value.leg_count == 0:
...             return "{}: I can't walk. No legs!".format(msg_prefix)
...         else:
...             return '{}: Walking...'.format(msg_prefix)
...
...     def fly(self, animal):
...         msg_prefix = self._get_message_prefix(animal)
...         if self.value.can_fly:
...             if self.value.likes_to_fly:
...                 return '{}: Taking off now...'.format(msg_prefix)
...             else:
...                 return '{}: Well, Ok. I can fly, but I would rather walk'.
↪format(msg_prefix)
...         elif self.value.can_fly is None and self.value.likes_to_fly:
...             return "{}: I would love to fly, but my wings have been clipped".
↪format(msg_prefix)
...         else:
...             return "{}: Sorry. I can't fly".format(msg_prefix)
...
...     def swim(self, animal):
...         msg_prefix = self._get_message_prefix(animal)
...         if self.value.can_swim:
...             if self.value.likes_to_swim:
...                 return "{}: Swimming, Swimming...".format(msg_prefix)
...             else:
...                 return "{}: Ok. I'll swim, but I don't like to".format(msg_prefix)
```

(continues on next page)

(continued from previous page)

```
...             else:
...                 return "{}: Sorry. I can't swim".format(msg_prefix)
...
>>> class Animal:
...     def __init__(self, name, animal_type):
...         self.name = name
...         self.type = animal_type
...
...     def __repr__(self):
...         return "<{}: name={!r}, type={!r}>".format(
...             self.__class__.__name__, self.name, self.type)
...
...     def walk(self):
...         return self.type.walk(self)
...
...     def fly(self):
...         return self.type.fly(self)
...
...     def swim(self):
...         return self.type.swim(self)
...
>>> animals = [
...     Animal('Spot', AnimalType.DOG),
...     Animal('Coco', AnimalType.AFRICAN_SWALLOW),
...     Animal('Garfield', AnimalType.CAT),
...     Animal('Cotton', AnimalType.PARROT),
...     Animal('Little', AnimalType.CHICKEN),
...     Animal('Nemo', AnimalType.FISH),
... ]
>>> pp([animal.walk() for animal in animals])
['Spot: I am a Dog: Walking...',
 'Coco: I am a African Swallow: Ok, but I would rather fly',
 'Garfield: I am a Cat: Walking...',
 "Cotton: I am a Parrot: I'm walking because my wings have been clipped. I "
 'prefer to fly',
 'Little: I am a Chicken: Cool! I like to walk, even though I can fly',
 "Nemo: I am a Fish: I can't walk. No legs!"]
>>> pp([animal.fly() for animal in animals])
["Spot: I am a Dog: Sorry. I can't fly",
 'Coco: I am a African Swallow: Taking off now...',
 "Garfield: I am a Cat: Sorry. I can't fly",
 'Cotton: I am a Parrot: I would love to fly, but my wings have been clipped',
 'Little: I am a Chicken: Well, Ok. I can fly, but I would rather walk',
 "Nemo: I am a Fish: Sorry. I can't fly"]
>>> pp([animal.swim() for animal in animals])
['Spot: I am a Dog: Swimming, Swimming...',
 "Coco: I am a African Swallow: Sorry. I can't swim",
 "Garfield: I am a Cat: Ok. I'll swim, but I don't like to",
 "Cotton: I am a Parrot: Sorry. I can't swim",
 "Little: I am a Chicken: Sorry. I can't swim",
 'Nemo: I am a Fish: Swimming, Swimming...']
```

This works, but the code has become much more complex and harder to write and read. The paramertization of imperative logic pattern we had been using has broken down.

Ideally, we would encapsulate some or all of the related custom behavior into classes and call the methods polymorphicaly.

Unfortunately, the `Enum` type does not allow sub-classing if the base class has members. Working around this, while maintaining the functionality we have, is not worth the trouble, as we already have an alternate solution that will be more straightforward.

Instances of `AnimalTypeAttrs` are already being associated with each of the members. We can move the behavior there and continue with our refactor.

Unfortunately, we don't actually have a class definition to add methods to since `namedtuple` builds `AnimalTypeAttrs`, but we can easily sub-class `AnimalTypeAttrs` to get around that.

So here is what we need to do:

- Create a sub-class of `AnimalTypeAttrs`
- Add the base methods to it
- Change the `AnimalType` methods to proxy to `self.value`
- Build the classes that encapsulate the behavior
- Instantiate the appropriate sub-class when we define the members of `AnimalType`

```
>>> AnimalTypeAttrs = namedtuple('AnimalTypeAttrs', (
...     'type_id', 'description', 'domesticated', 'leg_count', 'can_fly', 'can_swim'))
...
>>> class AnimalTypeBehavior(AnimalTypeAttrs):
...
...     def _get_message_prefix(self, animal):
...         return '{}: I am a {}'.format(animal.name, animal.type.name.title().
↪replace('_', ' '))
...
...     def walk(self, animal):
...         msg_prefix = self._get_message_prefix(animal)
...         if self.leg_count > 0:
...             return '{}: Walking...'.format(msg_prefix)
...         else:
...             return "{}: I can't walk. No legs".format(msg_prefix)
...
...     def fly(self, animal):
...         msg_prefix = self._get_message_prefix(animal)
...         if self.can_fly:
...             return '{}: Taking off now...'.format(msg_prefix)
...         else:
...             return "{}: Sorry. I can't fly".format(msg_prefix)
...
...     def swim(self, animal):
...         msg_prefix = self._get_message_prefix(animal)
...         if self.can_swim:
...             return '{}: Swimming, Swimming...'.format(msg_prefix)
...         else:
...             return "{}: Sorry. I can't swim".format(msg_prefix)
...
>>> class ReluctantWalkerAnimalTypeBehavior(AnimalTypeBehavior):
...     def walk(self, animal):
...         return '{}: Ok. but I would rather fly'.format(self._get_message_
↪prefix(animal))
...
>>> class ClippedBirdAnimalTypeBehavior(AnimalTypeBehavior):
...     def walk(self, animal):
...         return "{}: I'm walking because my wings have been clipped. I prefer to␣
↪fly".format(
```

(continues on next page)

(continued from previous page)

```
...             self._get_message_prefix(animal))
...
...     def fly(self, animal):
...         return "{}: I would love to fly, but my wings have been clipped".format(
...             self._get_message_prefix(animal))
...
>>> class ReluctantFlyerAnimalTypeBehavior(AnimalTypeBehavior):
...     def walk(self, animal):
...         return '{}: Cool! I like to walk, even though I can fly'.format(
...             self._get_message_prefix(animal))
...
...     def fly(self, animal):
...         return '{}: Well, Ok. I can fly, but I would rather walk'.format(
...             self._get_message_prefix(animal))
...
>>> class ReluctantSwimmerAnimalTypeBehavior(AnimalTypeBehavior):
...     def swim(self, animal):
...         return "{}: Well, Ok. I'll swim, but I would rather not".format(
...             self._get_message_prefix(animal))
...
>>> class AnimalType(MultiEnum):
...     DOG = AnimalTypeBehavior(1, "Man's best friend", True, 4, False, True)
...     AFRICAN_SWALLOW = ReluctantWalkerAnimalTypeBehavior(
...         2, 'Coconut carrier', False, 2, True, False)
...     CAT = ReluctantSwimmerAnimalTypeBehavior(3, "Man's overloard", True, 4, False,
↪ True)
...     PARROT = ClippedBirdAnimalTypeBehavior(4, 'Voice of a mute Pirate', True, 2,␣
↪True, False)
...     CHICKEN = ReluctantFlyerAnimalTypeBehavior(5, 'Disaster reporter', True, 2,␣
↪True, False)
...     FISH = AnimalTypeBehavior(6, 'Searcher', False, 0, False, True)
...
...     def walk(self, animal):
...         return self.value.walk(animal)
...
...     def fly(self, animal):
...         return self.value.fly(animal)
...
...     def swim(self, animal):
...         return self.value.swim(animal)
...
>>> animals = [
...     Animal('Spot', AnimalType.DOG),
...     Animal('Coco', AnimalType.AFRICAN_SWALLOW),
...     Animal('Garfield', AnimalType.CAT),
...     Animal('Cotton', AnimalType.PARROT),
...     Animal('Little', AnimalType.CHICKEN),
...     Animal('Nemo', AnimalType.FISH),
... ]
>>> pp([animal.walk() for animal in animals])
['Spot: I am a Dog: Walking...',
 'Coco: I am a African Swallow: Ok. but I would rather fly',
 'Garfield: I am a Cat: Walking...',
 "Cotton: I am a Parrot: I'm walking because my wings have been clipped. I "
 'prefer to fly',
 'Little: I am a Chicken: Cool! I like to walk, even though I can fly',
 "Nemo: I am a Fish: I can't walk. No legs"]
```

(continues on next page)

(continued from previous page)

```
>>> pp([animal.fly() for animal in animals])
["Spot: I am a Dog: Sorry. I can't fly",
 'Coco: I am a African Swallow: Taking off now...',
 "Garfield: I am a Cat: Sorry. I can't fly",
 'Cotton: I am a Parrot: I would love to fly, but my wings have been clipped',
 'Little: I am a Chicken: Well, Ok. I can fly, but I would rather walk',
 "Nemo: I am a Fish: Sorry. I can't fly"]
>>> pp([animal.swim() for animal in animals])
['Spot: I am a Dog: Swimming, Swimming...',
 "Coco: I am a African Swallow: Sorry. I can't swim",
 "Garfield: I am a Cat: Well, Ok. I'll swim, but I would rather not",
 "Cotton: I am a Parrot: Sorry. I can't swim",
 "Little: I am a Chicken: Sorry. I can't swim",
 'Nemo: I am a Fish: Swimming, Swimming...']
```

This design accomplishes our goals. The imperative code has been greatly simplified and custom behavior has been moved to separate classes.

But there are still some aspects of the design that are less than ideal:

- Creation of the `AnimalType` members requires 8 class definitions.
- The classes are in two separate class heirarchies.
- The basic functionality is spread across 3 classes.
- There is lots of repetition of the string 'AnimalType', even though all we are doing is defining the `AnimalType` collection, not actually using it. Granted, the custom behavior class names could be shortened, but their current names describe exactly what they are.
- The `repr()` of each member has become fairly complex, due to the 3 classes involved in its generation.
- We have lost easy access to the sub-set of members that have a behavior that is no longer driven by parameters. A method that encapsulates a loop using `type()` or `isinstance()` to compare a class argument to member values could be added to `MultiEnum` to restore that capability, but a solution based on those operations is less than ideal.
- Because our behavior classes are all decending from `AnimalTypeAttrs`, which is a `namedtuple`, any additional fields that a behavior class needs must be added to the fields required by *all* behavior class instantiations, even if those parameters are not applicable to that class. This could be worked around by abandoing `namedtuple` and writing a custom class that is more flexible, but this would add yet more code to an already complex implementationan.

StaticModel helps simplify a design like this, and comes with additional features that simplify its use with Django and Django Rest Framework.

1.3 The StaticModel solution

So, lets take our existing `AnimalType` implementation and refactor it using **StaticModel**.

```
>>> from staticmodel import StaticModel
>>>
>>>
>>> class AnimalType(StaticModel):
...     _field_names = (
...         'type_id', 'name', 'description', 'domesticated', 'leg_count', 'can_fly',
↪'can_swim')
```

(continues on next page)

(continued from previous page)

```
...     DOG = 1, 'Dog', "Man's best friend", True, 4, False, True
...     FISH = 6, 'Fish', 'Searcher', False, 0, False, True
...
...     def _get_message_prefix(self, animal):
...         return '{}: I am a {}'.format(animal.name, animal.type.name)
...
...     def walk(self, animal):
...         msg_prefix = self._get_message_prefix(animal)
...         if self.leg_count > 0:
...             return '{}: Walking...'.format(msg_prefix)
...         else:
...             return "{}: I can't walk. No legs".format(msg_prefix)
...
...     def fly(self, animal):
...         msg_prefix = self._get_message_prefix(animal)
...         if self.can_fly:
...             return '{}: Taking off now...'.format(msg_prefix)
...         else:
...             return "{}: Sorry. I can't fly".format(msg_prefix)
...
...     def swim(self, animal):
...         msg_prefix = self._get_message_prefix(animal)
...         if self.can_swim:
...             return '{}: Swimming, Swimming...'.format(msg_prefix)
...         else:
...             return "{}: Sorry. I can't swim".format(msg_prefix)
...
>>> class ReluctantWalkerAnimalType(AnimalType):
...     AFRICAN_SWALLOW = 2, 'Afican Swallow', 'Coconut carrier', False, 2, True,␣
↪False
...
...     def walk(self, animal):
...         return '{}: Ok. but I would rather fly'.format(self._get_message_
↪prefix(animal))
...
>>> class ClippedBirdAnimalType(AnimalType):
...     PARROT = 4, 'Parrot', 'Voice of a mute Pirate', True, 2, True, False
...
...     def walk(self, animal):
...         return "{}: I'm walking because my wings have been clipped. I prefer to␣
↪fly".format(
...             self._get_message_prefix(animal))
...
...     def fly(self, animal):
...         return "{}: I would love to fly, but my wings have been clipped".format(
...             self._get_message_prefix(animal))
...
>>> class ReluctantFlyerAnimalType(AnimalType):
...     _field_names = AnimalType._field_names + ('reluctance_reason',)
...     CHICKEN = 5, 'Chicken', 'Disaster reporter', True, 2, True, False, "I can't␣
↪fly very far"
...
...     def walk(self, animal):
...         return '{}: Cool! I like to walk, even though I can fly'.format(
...             self._get_message_prefix(animal))
...
...     def fly(self, animal):
```

(continues on next page)

(continued from previous page)

```
...         return '{}: Well, Ok. I can fly, but I would rather walk because {}'.
↪format(
...             self._get_message_prefix(animal), self.reluctance_reason)
...
>>> class ReluctantSwimmerAnimalType(AnimalType):
...     CAT = 3, 'Cat', "Man's overloard", True, 4, False, True
...
...     def swim(self, animal):
...         return "{}: Well, Ok. I'll swim, but I would rather not".format(
...             self._get_message_prefix(animal))
...
>>> pp(AnimalType.members.all())
[<AnimalType.DOG: type_id=1, name='Dog', description="Man's best friend",␣
↪domesticated=True, leg_count=4, can_fly=False, can_swim=True>,
 <AnimalType.FISH: type_id=6, name='Fish', description='Searcher', domesticated=False,
↪ leg_count=0, can_fly=False, can_swim=True>,
 <ReluctantWalkerAnimalType.AFRICAN_SWALLOW: type_id=2, name='Afican Swallow',␣
↪description='Coconut carrier', domesticated=False, leg_count=2, can_fly=True, can_
↪swim=False>,
 <ClippedBirdAnimalType.PARROT: type_id=4, name='Parrot', description='Voice of a␣
↪mute Pirate', domesticated=True, leg_count=2, can_fly=True, can_swim=False>,
 <ReluctantFlyerAnimalType.CHICKEN: type_id=5, name='Chicken', description='Disaster␣
↪reporter', domesticated=True, leg_count=2, can_fly=True, can_swim=False, reluctance_
↪reason="I can't fly very far">,
 <ReluctantSwimmerAnimalType.CAT: type_id=3, name='Cat', description="Man's overloard
↪", domesticated=True, leg_count=4, can_fly=False, can_swim=True>]
>>> pp(AnimalType.members.filter(domesticated=False))
[<AnimalType.FISH: type_id=6, name='Fish', description='Searcher', domesticated=False,
↪ leg_count=0, can_fly=False, can_swim=True>,
 <ReluctantWalkerAnimalType.AFRICAN_SWALLOW: type_id=2, name='Afican Swallow',␣
↪description='Coconut carrier', domesticated=False, leg_count=2, can_fly=True, can_
↪swim=False>]
>>> AnimalType.members.get(can_fly=True, domesticated=False)
<ReluctantWalkerAnimalType.AFRICAN_SWALLOW: type_id=2, name='Afican Swallow',␣
↪description='Coconut carrier', domesticated=False, leg_count=2, can_fly=True, can_
↪swim=False>
>>> AnimalType.members.filter(description='')
[]
>>> try:
...     AnimalType.members.get(can_fly=True)
... except AnimalType.MultipleObjectsReturned as e:
...     print(e)
AnimalType.members.get(can_fly=True) yielded multiple objects.
>>> try:
...     AnimalType.members.get(description='')
... except AnimalType.DoesNotExist as e:
...     print(e)
AnimalType.members.get(description='') yielded no objects.
>>> ReluctantFlyerAnimalType.members.all()
[<ReluctantFlyerAnimalType.CHICKEN: type_id=5, name='Chicken', description='Disaster␣
↪reporter', domesticated=True, leg_count=2, can_fly=True, can_swim=False, reluctance_
↪reason="I can't fly very far">]
```

Notice:

- Only 5 classes have been created, 4 of which provide custom behavior, as before.
- None of this implementation is a workaround.

- Classes are in a single class heirarchy.
- Basic functionality is contained in the single base class.
- Members are defined on their class, much like `Enum`, puting the instance data and the behavior that uses it close together.
- The base `AnimalType` class gains the members defined on its sub-classes, enabling polymorphic access to the entire collection.
- The concise member declarations.
- The ability to query `AnimalType` members with an api similar to the Django ORM is provided, much like our `MultiEnum` class.
- The `repr()` value of each member has been simplified.
- Easy access to only members with a particular behavior.
- Additional fields only have to be defined on and used for the classes they apply to (see `ReluctantFlyerAnimalType`).

CHAPTER 2

StaticModel User Guide

2.1 Static Models

A Static Model is defined using a class definition to create a sub-class of `staticmodel.StaticModel`.

Member field names are declared with the `_field_names` class attribute. The value should be a sequence of strings.

Members are declared with an uppercase class attribute. Member values should be sequences with the same number of items as the value of `_field_names`.

The subclass of `StaticModel` that is created can have other attributes and methods, just like a regular class. The only restriction is that **identifier names must be lower case or begin with an underscore**.

Once the class has been defined, **the members are transformed into instances of the model**.

```
>>> from staticmodel import StaticModel
>>>
>>>
>>> class AnimalType(StaticModel):
...     _field_names = 'name', 'description', 'domesticated', 'has_legs'
...     _WALKING_TEXT = "{} walking..."
...
...     DOG = 'Dog', "Man's best friend", True, True
...     CAT = 'Cat', "Man's gracious overlord", True, True
...     SNAKE = 'Snake', "Man's slithering companion", True, False
...
...     def walk(self):
...         if self.has_legs:
...             return self._WALKING_TEXT.format(self.name)
...         else:
...             return "{} can't walk.".format(self.name)
```

2.2 Member access methods

If the member name (the attribute name of the member defined on the model) is known, it can be accessed just like any other attribute.

```
>>> AnimalType.DOG
<AnimalType.DOG: name='Dog', description="Man's best friend", domesticated=True, has_
↪legs=True>
```

The entire collection of members can be retrieved with the `members.all()` method.

```
>>> pp(AnimalType.members.all())
[<AnimalType.DOG: name='Dog', description="Man's best friend", domesticated=True, has_
↪legs=True>,
 <AnimalType.CAT: name='Cat', description="Man's gracious overlord",␣
↪domesticated=True, has_legs=True>,
 <AnimalType.SNAKE: name='Snake', description="Man's slithering companion",␣
↪domesticated=True, has_legs=False>]
>>> pp([item.walk() for item in AnimalType.members.all()])
['Dog walking...', 'Cat walking...', "Snake can't walk."]
```

Model members may be filtered with the model's `members.filter()` method.

```
>>> pp(AnimalType.members.filter(has_legs=True))
[<AnimalType.DOG: name='Dog', description="Man's best friend", domesticated=True, has_
↪legs=True>,
 <AnimalType.CAT: name='Cat', description="Man's gracious overlord",␣
↪domesticated=True, has_legs=True>]
```

Providing no criteria to `members.filter()` is the same as calling `members.all()`.

```
>>> pp(AnimalType.members.filter())
[<AnimalType.DOG: name='Dog', description="Man's best friend", domesticated=True, has_
↪legs=True>,
 <AnimalType.CAT: name='Cat', description="Man's gracious overlord",␣
↪domesticated=True, has_legs=True>,
 <AnimalType.SNAKE: name='Snake', description="Man's slithering companion",␣
↪domesticated=True, has_legs=False>]
```

The `members.all()` and `members.filter()` methods return an empty list if no members were found.

```
>>> class NoMembers(StaticModel):
...     _field_names = ('something',)
...
>>> pp(NoMembers.members.all())
[]
>>> pp(NoMembers.members.filter(something='nothing'))
[]
>>>
```

A single model member may be retrieved directly using the model's `members.get()` method.

```
>>> AnimalType.members.get(name='Dog')
<AnimalType.DOG: name='Dog', description="Man's best friend", domesticated=True, has_
↪legs=True>
>>>
```

The `members.get()` method raises `<model>.DoesNotExist` if the query is unsuccessful and `<model>.MultipleObjectsReturned` if more than one is returned.

```
>>> try:
...     AnimalType.members.get(name='Eagle')
... except AnimalType.DoesNotExist as e:
...     print(e)
AnimalType.members.get(name='Eagle') yielded no objects.
>>> try:
...     AnimalType.members.get(domesticated=True)
... except AnimalType.MultipleObjectsReturned as e:
...     print(e)
AnimalType.members.get(domesticated=True) yielded multiple objects.
>>>
```

The `members.choices()` method is a shortcut for generating lists of 2-item tuples for use in things like Django field definitions. By default, it returns all members and uses the first two fields defined on the model.

```
>>> pp(AnimalType.members.choices())
[('Dog', "Man's best friend"),
 ('Cat', "Man's gracious overlord"),
 ('Snake', "Man's slithering companion")]
>>>
```

If field names are specified, there must be no more than 2.

```
>>> try:
...     AnimalType.members.choices('name', 'description', 'domesticated')
... except ValueError as e:
...     print(e)
Maximum number of specified fields for AnimalType.members.choices() is 2
```

If only a singe field name is provided, or if the model only has one field, then the same field is used for both items of the tuple.

```
>>> pp(AnimalType.members.choices('name'))
[('Dog', 'Dog'), ('Cat', 'Cat'), ('Snake', 'Snake')]
```

The `members.choices()` method may also be provided with criteria to limit the members included in the results, much like `members.filter()`.

```
>>> pp(AnimalType.members.choices(has_legs=True))
[('Dog', "Man's best friend"), ('Cat', "Man's gracious overlord")]
>>> pp(AnimalType.members.choices('name', has_legs=True))
[('Dog', 'Dog'), ('Cat', 'Cat')]
```

2.2.1 The _member_name field

The name of each member's class attribute on the model and parent models is available as the `_member_name` field on the member.

```
>>> AnimalType.DOG._member_name
'DOG'
```

The `_member_name` field can be used in member queries if needed.

```
>>> AnimalType.members.get(_member_name='CAT')
<AnimalType.CAT: name='Cat', description="Man's gracious overlord", domesticated=True,
↪ has_legs=True>
>>> AnimalType.members.filter(_member_name='DOG')
[<AnimalType.DOG: name='Dog', description="Man's best friend", domesticated=True, has_
↪legs=True>]
```

However, if only a single member is needed, using the built-in `getattr()` is more concise.

```
>>> getattr(AnimalType, 'SNAKE')
<AnimalType.SNAKE: name='Snake', description="Man's slithering companion",␣
↪domesticated=True, has_legs=False>
```

2.3 Sub-models

Models can have sub-models. Sub-models are created using normal sub-class syntax.

```
>>> class WildAnimalType(AnimalType):
...     DEER = 'Deer', 'Likes to hide', False, True
...     ANTELOPE = 'Antelope', 'Likes to run', False, True
...
...     def walk(self):
...         return '{}warily'.format(super().walk())
```

Sub-models inherit the `_field_names` attribute of their parent model.

```
>>> WildAnimalType._field_names
('name', 'description', 'domesticated', 'has_legs')
>>> WildAnimalType.DEER
<WildAnimalType.DEER: name='Deer', description='Likes to hide', domesticated=False,␣
↪has_legs=True>
```

However, sub-models DO NOT inherit the members of their parent model.

```
>>> WildAnimalType.DOG
Traceback (most recent call last):
    ...
AttributeError: 'WildAnimalType' model does not contain member 'DOG'
>>> pp(WildAnimalType.members.all())
[<WildAnimalType.DEER: name='Deer', description='Likes to hide', domesticated=False,␣
↪has_legs=True>,
 <WildAnimalType.ANTELOPE: name='Antelope', description='Likes to run',␣
↪domesticated=False, has_legs=True>]
```

Parent models **gain the members** of their sub-models. Notice that the `AnimalType` model now contains the members just defined in the `WildAnimalType` sub-model.

```
>>> pp(AnimalType.members.all())
[<AnimalType.DOG: name='Dog', description="Man's best friend", domesticated=True, has_
↪legs=True>,
 <AnimalType.CAT: name='Cat', description="Man's gracious overlord",␣
↪domesticated=True, has_legs=True>,
 <AnimalType.SNAKE: name='Snake', description="Man's slithering companion",␣
↪domesticated=True, has_legs=False>,
```

(continues on next page)

(continued from previous page)

```
 <WildAnimalType.DEER: name='Deer', description='Likes to hide', domesticated=False,␣
↪has_legs=True>,
 <WildAnimalType.ANTELOPE: name='Antelope', description='Likes to run',␣
↪domesticated=False, has_legs=True>]
```

The members that the parent has gained are accessed exactly the same way as the other members, and behave as expected.

```
>>> pp([item.walk() for item in AnimalType.members.all()])
['Dog walking...',
 'Cat walking...',
 "Snake can't walk.",
 'Deer walking...warily',
 'Antelope walking...warily']
```

2.3.1 Additional fields

Additional field names can be provided by overriding `_field_names` in sub-models. A good practice is to reference the parent model's values as demonstrated in the `SmallHousePet` model below.

```
>>> class SmallHousePet(AnimalType):
...     _field_names = AnimalType._field_names + ('facility',)
...
...     FISH = 'Fish', 'Likes to swim', True, True, 'tank'
...     RODENT = 'Rodent', 'Likes to eat', True, True, 'cage'
```

Member queries on the sub-model can use the additional field names.

```
>>> pp(SmallHousePet.members.filter(facility='tank'))
[<SmallHousePet.FISH: name='Fish', description='Likes to swim', domesticated=True,␣
↪has_legs=True, facility='tank'>]
```

Parent models are not aware of additional fields that have been added by sub-models, so those additional fields cannot be used in member queries.

```
>>> try:
...     AnimalType.members.filter(facility='tank')
... except AnimalType.InvalidField as e:
...     print(e)
...
Invalid field 'facility'
```

2.4 Primitive Collections

Model members may be rendered as primitive collections.

The methods `members.all()` and `members.filter()` return a list with the methods `values()` and `values_list()` defined on it.

The `values()` method returns a list of dictionaries.

```
>>> # Custom function that returns the same results in python 2 and 3
>>> # for lists containing dictionaries.
>>> from staticmodel.util import jsonify
>>>
>>>
>>> jsonify(AnimalType.members.all().values())
[
  {
    "name": "Dog",
    "description": "Man's best friend",
    "domesticated": true,
    "has_legs": true
  },
  {
    "name": "Cat",
    "description": "Man's gracious overlord",
    "domesticated": true,
    "has_legs": true
  },
  {
    "name": "Snake",
    "description": "Man's slithering companion",
    "domesticated": true,
    "has_legs": false
  },
  {
    "name": "Deer",
    "description": "Likes to hide",
    "domesticated": false,
    "has_legs": true
  },
  {
    "name": "Antelope",
    "description": "Likes to run",
    "domesticated": false,
    "has_legs": true
  },
  {
    "name": "Fish",
    "description": "Likes to swim",
    "domesticated": true,
    "has_legs": true
  },
  {
    "name": "Rodent",
    "description": "Likes to eat",
    "domesticated": true,
    "has_legs": true
  }
]
>>> jsonify(AnimalType.members.filter(name='Rodent').values())
[
  {
    "name": "Rodent",
    "description": "Likes to eat",
    "domesticated": true,
    "has_legs": true
```

(continues on next page)

(continued from previous page)

```
  }
]
```

The `values_list()` method returns a list of tuples.

```
>>> pp(AnimalType.members.all().values_list())
[('Dog', "Man's best friend", True, True),
 ('Cat', "Man's gracious overlord", True, True),
 ('Snake', "Man's slithering companion", True, False),
 ('Deer', 'Likes to hide', False, True),
 ('Antelope', 'Likes to run', False, True),
 ('Fish', 'Likes to swim', True, True),
 ('Rodent', 'Likes to eat', True, True)]
>>> pp(AnimalType.members.filter(domesticated=False).values_list())
[('Deer', 'Likes to hide', False, True),
 ('Antelope', 'Likes to run', False, True)]
```

Notice that when the `AnimalType` model was used to execute `.values()` or `.values_list()`, the `facility` field was not included in the results. This is because the default fields for these methods is the value of `AnimalType._field_names`, which does not include `facility`.

Specific fields for `.values()` and `.values_list()` may be provided by passing them as positional parameters to those methods.

```
>>> jsonify(AnimalType.members.all().values('name', 'domesticated', 'facility'))
[
  {
    "name": "Dog",
    "domesticated": true,
    "facility": null
  },
  {
    "name": "Cat",
    "domesticated": true,
    "facility": null
  },
  {
    "name": "Snake",
    "domesticated": true,
    "facility": null
  },
  {
    "name": "Deer",
    "domesticated": false,
    "facility": null
  },
  {
    "name": "Antelope",
    "domesticated": false,
    "facility": null
  },
  {
    "name": "Fish",
    "domesticated": true,
    "facility": "tank"
  },
  {
```

(continues on next page)

(continued from previous page)

```
    "name": "Rodent",
    "domesticated": true,
    "facility": "cage"
  }
]
>>> pp(AnimalType.members.all().values_list('name', 'description', 'facility'))
[('Dog', "Man's best friend", None),
 ('Cat', "Man's gracious overlord", None),
 ('Snake', "Man's slithering companion", None),
 ('Deer', 'Likes to hide', None),
 ('Antelope', 'Likes to run', None),
 ('Fish', 'Likes to swim', 'tank'),
 ('Rodent', 'Likes to eat', 'cage')]
```

Notice that some members have the `facility` field set to None (or null when converted to JSON). These are placeholders for fields that were requested, but do not exist on that member.

The `values_list()` method can be passed the `flat=True` parameter to collapse the values in the result.

```
>>> jsonify(AnimalType.members.all().values_list('facility', flat=True))
[
  "tank",
  "cage"
]
```

Using `flat=True` usually only makes sense when limiting the results to a single field name.

```
>>> jsonify(AnimalType.members.all().values_list('name', 'description', flat=True))
[
  "Dog",
  "Man's best friend",
  "Cat",
  "Man's gracious overlord",
  "Snake",
  "Man's slithering companion",
  "Deer",
  "Likes to hide",
  "Antelope",
  "Likes to run",
  "Fish",
  "Likes to swim",
  "Rodent",
  "Likes to eat"
]
```

CHAPTER 3

Django model fields

Static Model provides custom Django model fields in the `staticmodel.django.models` package:

- `StaticModelCharField` (sub-class of `django.db.models.CharField`)
- `StaticModelTextField` (sub-class of `django.db.models.TextField`)
- `StaticModelIntegerField` (sub-class of `django.db.models.IntegerField`)

Static model members are returned, and can be set, as the value of the fields on a django model object.

All fields take the following keyword arguments in addition to the arguments taken by their respective parent classes:

- `static_model`: The static model class associated with this field.
- `value_field_name`: The static model field name whose value will be stored in the database. Defaults to the first field name in `static_model._field_names`.
- `display_field_name`: The static model field name whose value will be used as the display value in the `choices` passed to the parent field. Defaults to the value of `value_field_name`.

When the model field is instantiated, it validates the values of `value_field_name` and `display_field_name` against **every member** of the static model to insure the fields exist and contain a value appropriate for the value of the field. This ensures that error-causing inconsistencies are detected early during development.

CHAPTER 4

Django Rest Framework serializer fields

Static Model provides custom serializer fields in the `staticmodel.django.rest_framework.serializers` module:

- `StaticModelCharField` (sub-class of `rest_framework.serializers.CharField`)
- `StaticModelIntegerField` (sub-class of `rest_framework.serializers.IntegerField`)

All fields take the following keyword arguments in addition to the arguments taken by their respective parent classes:

- `static_model`: The static model class associated with this field.
- `lookup_field_name`: The static model field name that will be used to lookup the static model member when deserializing, and the field name to retrieve the value from when serializing (unless `static_model_expand=True`. See below.). Defaults to the first field name in `static_model._field_names`.
- `static_model_expand`: When set to `True`, return the entire static model member as a mapping. Defaults to `False`.

Regardless of the value of `static_model_expand`, if the value passed during deserialization is a mapping, it will be used to retrieve the lookup value using `lookup_field_name`.

Python Module Index

S

Index

S