django-extended-choices Documentation

Release 1.0

Stephane "Twidi" Angel

Jan 17, 2019

Contents

CHAPTER 1

What is it?

Little helper application to improve django choices (for fields)

CHAPTER 2

Documentation contents

2.1 django-extended-choices

2.1.1 A little application to improve Django choices

`django-extended-choices` aims to provide a better and more readable way of using choices in Django.

2.1.2 Installation

You can install directly via pip (since version `0.3`):

```
$ pip install django-extended-choices
```

Or from the Github repository (`master` branch by default):

```
$ git clone git://github.com/twidi/django-extended-choices.git
$ cd django-extended-choices
$ sudo python setup.py install
```

2.1.3 Usage

The aim is to replace this:

```
STATE_ONLINE  = 1
STATE_DRAFT   = 2
STATE_OFFLINE = 3
```

(continues on next page)

(continued from previous page)

```
STATE_CHOICES = (
    (STATE_ONLINE,  'Online'),
    (STATE_DRAFT,   'Draft'),
    (STATE_OFFLINE, 'Offline'),
)

STATE_DICT = dict(STATE_CHOICES)

class Content(models.Model):
    title       = models.CharField(max_length=255)
    content     = models.TextField()
    state       = models.PositiveSmallIntegerField(choices=STATE_CHOICES,␣
↪default=STATE_DRAFT)

    def __unicode__(self):
        return u'Content "%s" (state=%s)' % (self.title, STATE_DICT[self.state])

print(Content.objects.filter(state=STATE_ONLINE))
```

by this:

```
from extended_choices import Choices

STATES = Choices(
    ('ONLINE',  1, 'Online'),
    ('DRAFT',   2, 'Draft'),
    ('OFFLINE', 3, 'Offline'),
)

class Content(models.Model):
    title       = models.CharField(max_length=255)
    content     = models.TextField()
    state       = models.PositiveSmallIntegerField(choices=STATES, default=STATES.
↪DRAFT)

    def __unicode__(self):
        return u'Content "%s" (state=%s)' % (self.title, STATES.for_value(self.state).
↪display)

print(Content.objects.filter(state=STATES.ONLINE))
```

As you can see there is only one declaration for all states with, for each state, in order:

- the pseudo-constant name which can be used (`STATES.ONLINE` replaces the previous `STATE_ONLINE`)
- the value to use in the database - which could equally be a string
- the name to be displayed - and you can wrap the text in `ugettext_lazy()` if you need i18n

And then, you can use:

- `STATES`, or `STATES.choices`, to use with `choices=` in fields declarations
- `STATES.for_constant(constant)`, to get the choice entry from the constant name
- `STATES.for_value(constant)`, to get the choice entry from the key used in database
- `STATES.for_display(constant)`, to get the choice entry from the displayable value (can be useful in some case)

Each choice entry obtained by `for_constant`, `for_value` and `for_display` return a tuple as given to the `Choices` constructor, but with additional attributes:

```
>>> entry = STATES.for_constant('ONLINE')
>>> entry == ('ONLINE', 1, 'Online')
True
>>> entry.constant
'ONLINE'
>>> entry.value
1
>>> entry.display
'Online'
```

These attributes are chainable (with a weird example to see chainability):

```
>>> entry.constant.value
1
>>> entry.constant.value.value.display.constant.display
'Online'
```

To allow this, we had to remove support for `None` values. Use empty strings instead.

Note that constants can be accessed via a dict key (`STATES['ONLINE']` for example) if you want to fight your IDE that may warn you about undefined attributes.

You can check whether a value is in a `Choices` object directly:

```
>>> 1 in STATES
True
>>> 42 in STATES
False
```

You can even iterate on a `Choices` objects to get choices as seen by Django:

```
>>> for choice in STATES:
...     print(choice)
(1, 'Online')
(2, 'Draf')
(3, 'Offline')
```

To get all choice entries as given to the `Choices` object, you can use the `entries` attribute:

```
>>> for choice_entry in STATES.entries:
...     print(choice_entry)
('ONLINE',  1, 'Online'),
('DRAFT',   2, 'Draft'),
('OFFLINE', 3, 'Offline'),
```

Or the following dicts, using constants, values or display names, as keys, and the matching choice entry as values:

- `STATES.constants`
- `STATES.values`
- `STATES.displays`

```
>>> STATES.constants['ONLINE'] is STATES.for_constant('ONLINE')
True
>>> STATES.values[2] is STATES.for_value(2)
```

(continues on next page)

(continued from previous page)

```
True
>>> STATES.displays['Offline'] is STATES.for_display('Offline')
True
```

If you want these dicts to be ordered, you can pass the dict class to use to the `Choices` constructor:

```
from collections import OrderedDict
STATES = Choices(
    ('ONLINE',  1, 'Online'),
    ('DRAFT',   2, 'Draft'),
    ('OFFLINE', 3, 'Offline'),
    dict_class = OrderedDict
)
```

Since version `1.1`, the new `OrderedChoices``class is provided, that is exactly that: a ``Choices` using `OrderedDict` by default for `dict_class`. You can directly import it from `extended_choices`.

You can check if a constant, value, or display name exists:

```
>>> STATES.has_constant('ONLINE')
True
>>> STATES.has_value(1)
True
>>> STATES.has_display('Online')
True
```

You can create subsets of choices within the same `Choices` instance:

```
>>> STATES.add_subset('NOT_ONLINE', ('DRAFT', 'OFFLINE',))
>>> STATES.NOT_ONLINE
(2, 'Draft')
(3, 'Offline')
```

Now, `STATES.NOT_ONLINE` is a real `Choices` instance, with a subset of the main `STATES` constants.

You can use it to generate choices for when you only want a subset of choices available:

```
offline_state = models.PositiveSmallIntegerField(
    choices=STATES.NOT_ONLINE,
    default=STATES.DRAFT
)
```

As the subset is a real `Choices` instance, you have the same attributes and methods:

```
>>> STATES.NOT_ONLINE.for_constant('OFFLINE').value
3
>>> STATES.NOT_ONLINE.for_value(1).constant
Traceback (most recent call last):
...
KeyError: 3
>>> list(STATES.NOT_ONLINE.constants.keys())
['DRAFT', 'OFFLINE]
>>> STATES.NOT_ONLINE.has_display('Online')
False
```

You can create as many subsets as you want, reusing the same constants if needed:

```
STATES.add_subset('NOT_OFFLINE', ('ONLINE', 'DRAFT'))
```

If you want to check membership in a subset you could do:

```
def is_online(self):
    # it's an example, we could have just tested with STATES.ONLINE
    return self.state not in STATES.NOT_ONLINE_DICT
```

You can add choice entries in many steps using `add_choices`, possibly creating subsets at the same time.

To construct the same `Choices` as before, we could have done:

```
STATES = Choices()
STATES.add_choices(
    ('ONLINE', 1, 'Online)
)
STATES.add_choices(
    ('DRAFT',   2, 'Draft'),
    ('OFFLINE', 3, 'Offline'),
    name='NOT_ONLINE'
)
```

You can also pass the `argument` to the `Choices` constructor to create a subset with all the choices entries added at the same time (it will call `add_choices` with the name and the entries)

The list of existing subset names is in the `subsets` attributes of the parent `Choices` object.

If you want a subset of the choices but not save it in the original `Choices` object, you can use `extract_subset` instead of `add_subset`

```
>>> subset = STATES.extract_subset('DRAFT', 'OFFLINE')
>>> subset
(2, 'Draft')
(3, 'Offline')
```

As for a subset created by `add_subset`, you have a real `Choices` object, but not accessible from the original `Choices` object.

Note that in `extract_subset`, you pass the strings directly, not in a list/tuple as for the second argument of `add_subset`.

2.1.4 Additional attributes

Each tuple must contain three elements. But you can pass a dict as a fourth one and each entry of this dict will be saved as an attribute of the choice entry

```
>>> PLANETS = Choices(
...     ('EARTH', 'earth', 'Earth', {'color': 'blue'}),
...     ('MARS', 'mars', 'Mars', {'color': 'red'}),
... )
>>> PLANETS.EARTH.color
'blue'
```

2.1.5 Auto display/value

We provide two classes to eases the writing of your choices, attended you don't need translation on the display value.

2.1.5.1 AutoChoices

It's the simpler and faster version: you just past constants and:

- the value saved in database will be constant lower cased
- the display value will be the constant with _ replaced by spaces, and the first letter capitalized

```
>>> from extended_choices import AutoChoices
>>> PLANETS = AutoChoices('EARTH', 'MARS')
>>> PLANETS.EARTH.value
'earth'
>>> PLANETS.MARS.display
'Mars'
```

If you want to pass additional attributes, pass a tuple with the dict as a last element:

```
>>> PLANETS = AutoChoices(
...     ('EARTH', {'color': 'blue'}),
...     ('MARS', {'color': 'red'}),
... )
>>> PLANETS.EARTH.value
'earth'
>>> PLANETS.EARTH.color
'blue'
```

You can change the transform function used to convert the constant to the value to be saved and the display value, by passing `value_transform` and `display_transform` functions to the constructor.

```
>>> PLANETS = AutoChoices(
...     'EARTH', 'MARS',
...     value_transform=lambda const: 'planet_' + const.lower().
...     display_transform=lambda const: 'Planet: ' + const.lower().
... )
>>> PLANETS.EARTH.value
'planet_earth'
>>> PLANETS.MARS.display
'Planet: mars'
```

If you find yourself repeting these transform functions you can have a base class that defines these function, as class attributes:

```
>>> class MyAutoChoices(AutoChoices):
...     value_transform=staticmethod(lambda const: const.upper())
...     display_transform=staticmethod(lambda const: const.lower())

>>> PLANETS = MyAutoChoices('EARTH', 'MARS')
>>> PLANETS.EARTH.value
'EARTH'
>>> PLANETS.MARS.dispay
'mars'
```

Of course you can still override the functions by passing them to the constructor.

If you want, for an entry, force a specific value, you can do it by simply passing it as a second argument:

```
>>> PLANETS = AutoChoices(
...     'EARTH',
```

(continues on next page)

(continued from previous page)

```
...     ('MARS', 'red-planet'),
... )
>>> PLANETS.MARS.value
'red-planet'
```

And then if you want to set the display, pass a third one:

```
>>> PLANETS = AutoChoices(
...     'EARTH',
...     ('MARS', 'red-planet', 'Red planet'),
... )
>>> PLANETS.MARS.value
'red-planet'
>>> PLANETS.MARS.display
'Red planet'
```

To force a display value but let the db value to be automatically computed, use `None` for the second argument:

```
>>> PLANETS = AutoChoices(
...     'EARTH',
...     ('MARS', None, 'Red planet'),
... )
>>> PLANETS.MARS.value
'mars'
>>> PLANETS.MARS.display
'Red planet'
```

2.1.5.2 AutoDisplayChoices

In this version, you have to define the value to save in database. The display value will be composed like in `AutoChoices`

```
>>> from extended_choices import AutoDisplayChoices
>>> PLANETS = AutoDisplayChoices(
...     ('EARTH', 1),
...     ('MARS', 2),
... )
>>> PLANETS.EARTH.value
1
>>> PLANETS.MARS.display
'Mars'
```

If you want to pass additional attributes, pass a tuple with the dict as a last element:

```
>>> PLANETS = AutoDisplayChoices(
...     ('EARTH', 'earth', {'color': 'blue'}),
...     ('MARS', 'mars', {'color': 'red'}),
... )
>>> PLANETS.EARTH.value
1
>>> PLANETS.EARTH.display
'Earth'
>>> PLANETS.EARTH.color
'blue'
```

As in `AutoChoices`, you can change the transform function for the value to display by passing `display_transform` to the constructor.

If you want, for an entry, force a specific display, you can do it by simply passing it as a third argument:

```
>>> PLANETS = AutoChoices(
...     ('EARTH', 1),
...     ('MARS', 2, 'Red planet'),
... )
>>> PLANETS.MARS.display
'Red planet'
```

2.1.6 Notes

- You also have a very basic field (`NamedExtendedChoiceFormField`) in `extended_choices.fields` which accept constant names instead of values
- Feel free to read the source to learn more about this little Django app.
- You can declare your choices where you want. My usage is in the `models.py` file, just before the class declaration.

2.1.7 Compatibility

The version `1.0` provided a totally new API, and compatibility with the previous one (`0.4.1`) was removed in `1.1`. The last version with the compatibility was `1.0.7`.

If you need this compatibility, you can use a specific version by pinning it in your requirements.

2.1.8 License

Available under the BSD License. See the `LICENSE` file included

2.1.9 Python/Django versions support

Django version	Python versions
1.8	2.7, 3.4, 3.5
1.9, 1.10	2.7, 3.4, 3.5
1.11	2.7, 3.4, 3.5, 3.6
2.0	3.4, 3.5, 3.6
2.1	3.5, 3.6, 3.7

2.1.10 Tests

To run tests from the code source, create a virtualenv or activate one, install Django, then:

```
python -m extended_choices.tests
```

We also provides some quick doctests in the code documentation. To execute them:

```
python -m extended_choices
```

Note: the doctests will work only in python version not display *u* prefix for strings.

2.1.11 Source code

The source code is available on Github.

2.1.12 Developing

If you want to participate in the development of this library, you'll need `Django` installed in your virtualenv. If you don't have it, simply run:

```
pip install -r requirements-dev.txt
```

Don't forget to run the tests ;)

Feel free to propose a pull request on Github!

A few minutes after your pull request, tests will be executed on TravisCi for all the versions of python and Django we support.

2.1.13 Documentation

You can find the documentation on ReadTheDoc

To update the documentation, you'll need some tools:

```
pip install -r requirements-makedoc.txt
```

Then go to the `docs` directory, and run:

```
make html
```

2.1.14 Author

Written by Stephane "Twidi" Angel <s.angel@twidi.com> (http://twidi.com), originally for http://www.liberation.fr

2.2 extended_choices.choices module

Provides a `Choices` class to help using "choices" in Django fields.

The aim is to replace:

```
STATE_ONLINE  = 1
STATE_DRAFT   = 2
STATE_OFFLINE = 3

STATE_CHOICES = (
    (STATE_ONLINE,  'Online'),
    (STATE_DRAFT,   'Draft'),
```

(continues on next page)

(continued from previous page)

```
    (STATE_OFFLINE, 'Offline'),
)

STATE_DICT = dict(STATE_CHOICES)

class Content(models.Model):
    title      = models.CharField(max_length=255)
    content    = models.TextField()
    state      = models.PositiveSmallIntegerField(choices=STATE_CHOICES,␣
↪default=STATE_DRAFT)

    def __unicode__(self):
        return 'Content "%s" (state=%s)' % (self.title, STATE_DICT[self.state])

print(Content.objects.filter(state=STATE_ONLINE))
```

By this:

```
from extended_choices import Choices

STATES = Choices(
    ('ONLINE',  1, 'Online'),
    ('DRAFT',   2, 'Draft'),
    ('OFFLINE', 3, 'Offline'),
)

class Content(models.Model):
    title      = models.CharField(max_length=255)
    content    = models.TextField()
    state      = models.PositiveSmallIntegerField(choices=STATES, default=STATES.
↪DRAFT)

    def __unicode__(self):
        return 'Content "%s" (state=%s)' % (self.title, STATES.for_value(self.state).
↪display)

print(Content.objects.filter(state=STATES.ONLINE))
```

Notes

The documentation format in this file is numpydoc.

class extended_choices.choices.**Choices**(**choices*, ***kwargs*)

Bases: `list`

Helper class for choices fields in Django

A choice entry has three representation: constant name, value and display name). So `Choices` takes list of such tuples.

It's easy to get the constant, value or display name given one of these value. See in example.

Parameters

- ***choices** (*list of tuples*) – It's the list of tuples to add to the `Choices` instance, each tuple having three entries: the constant name, the value, the display name.

A dict could be added as a 4th entry in the tuple to allow setting arbitrary arguments to the final `ChoiceEntry` created for this choice tuple.

- **name** (*string, optional*) – If set, a subset will be created containing all the constants. It could be used if you construct your `Choices` instance with many calls to `add_choices`.
- **dict_class** (*type, optional*) – `dict` by default, it's the dict class to use to create dictionaries (`constants`, `values` and `displays`. Could be set for example to `OrderedDict` (you can use `OrderedChoices` that is a simple subclass using `OrderedDict`.

Example

Start by declaring your `Choices`:

```
>>> ALIGNMENTS = Choices(
...     ('BAD', 10, 'bad'),
...     ('NEUTRAL', 20, 'neutral'),
...     ('CHAOTIC_GOOD', 30, 'chaotic good'),
...     ('GOOD', 40, 'good'),
...     dict_class=OrderedDict
... )
```

Then you can use it in a django field, Notice its usage in `choices` and `default`:

```
>>> from django.conf import settings
>>> try:
...     settings.configure(DATABASE_ENGINE='sqlite3')
... except: pass
>>> from django.db.models import IntegerField
>>> field = IntegerField(choices=ALIGNMENTS,  # use ``ALIGNMENTS`` or␣
↪``ALIGNMENTS.choices``.
...                       default=ALIGNMENTS.NEUTRAL)
```

The `Choices` returns a list as expected by django:

```
>>> ALIGNMENTS == ((10, 'bad'), (20, 'neutral'), (30, 'chaotic good'), (40, 'good
↪'))
True
```

But represents it with the constants:

```
>>> repr(ALIGNMENTS)
"[('BAD', 10, 'bad'), ('NEUTRAL', 20, 'neutral'), ('CHAOTIC_GOOD', 30, 'chaotic␣
↪good'), ('GOOD', 40, 'good')]"
```

Use `choices` which is a simple list to represent it as such:

```
>>> ALIGNMENTS.choices
((10, 'bad'), (20, 'neutral'), (30, 'chaotic good'), (40, 'good'))
```

And you can access value by their constant, or as you want:

```
>>> ALIGNMENTS.BAD
10
```

(continues on next page)

(continued from previous page)

```
>>> ALIGNMENTS.BAD.display
'bad'
>>> 40 in ALIGNMENTS
True
>>> ALIGNMENTS.has_constant('BAD')
True
>>> ALIGNMENTS.has_value(20)
True
>>> ALIGNMENTS.has_display('good')
True
>>> ALIGNMENTS.for_value(10)
('BAD', 10, 'bad')
>>> ALIGNMENTS.for_value(10).constant
'BAD'
>>> ALIGNMENTS.for_display('good').value
40
>>> ALIGNMENTS.for_constant('NEUTRAL').display
'neutral'
>>> ALIGNMENTS.constants
OrderedDict([('BAD', ('BAD', 10, 'bad')), ('NEUTRAL', ('NEUTRAL', 20, 'neutral')),
↪ ('CHAOTIC_GOOD', ('CHAOTIC_GOOD', 30, 'chaotic good')), ('GOOD', ('GOOD', 40,
↪'good'))])
>>> ALIGNMENTS.values
OrderedDict([(10, ('BAD', 10, 'bad')), (20, ('NEUTRAL', 20, 'neutral')), (30, (
↪'CHAOTIC_GOOD', 30, 'chaotic good')), (40, ('GOOD', 40, 'good'))])
>>> ALIGNMENTS.displays
OrderedDict([('bad', ('BAD', 10, 'bad')), ('neutral', ('NEUTRAL', 20, 'neutral')),
↪ ('chaotic good', ('CHAOTIC_GOOD', 30, 'chaotic good')), ('good', ('GOOD', 40,
↪'good'))])
```

You can create subsets of choices:

```
>>> ALIGNMENTS.add_subset('WESTERN',('BAD', 'GOOD'))
>>> ALIGNMENTS.WESTERN.choices
((10, 'bad'), (40, 'good'))
>>> ALIGNMENTS.BAD in ALIGNMENTS.WESTERN
True
>>> ALIGNMENTS.NEUTRAL in ALIGNMENTS.WESTERN
False
```

To use it in another field (only the values in the subset will be available), or for checks:

```
>>> def is_western(value):
...     return value in ALIGNMENTS.WESTERN
>>> is_western(40)
True
```

ChoiceEntryClass
alias of *extended_choices.helpers.ChoiceEntry*

add_choices(**choices*, ***kwargs*)
Add some choices to the current `Choices` instance.

The given choices will be added to the existing choices. If a `name` attribute is passed, a new subset will be created with all the given choices.

Note that it's not possible to add new choices to a subset.

Parameters

- ***choices** (*list of tuples*) – It's the list of tuples to add to the `Choices` instance, each tuple having three entries: the constant name, the value, the display name.

  A dict could be added as a 4th entry in the tuple to allow setting arbitrary arguments to the final `ChoiceEntry` created for this choice tuple.

  If the first entry of `*choices` is a string, then it will be used as a name for a new subset that will contain all the given choices.

- ****kwargs** (*dict*) –

  name [string] Instead of using the first entry of the `*choices` to pass a name of a subset to create, you can pass it via the `name` named argument.

Example

```
>>> MY_CHOICES = Choices()
>>> MY_CHOICES.add_choices(('ZERO', 0, 'zero'))
>>> MY_CHOICES
[('ZERO', 0, 'zero')]
>>> MY_CHOICES.add_choices('SMALL', ('ONE', 1, 'one'), ('TWO', 2, 'two'))
>>> MY_CHOICES
[('ZERO', 0, 'zero'), ('ONE', 1, 'one'), ('TWO', 2, 'two')]
>>> MY_CHOICES.SMALL
[('ONE', 1, 'one'), ('TWO', 2, 'two')]
>>> MY_CHOICES.add_choices(('THREE', 3, 'three'), ('FOUR', 4, 'four'), name=
↪'BIG')
>>> MY_CHOICES
[('ZERO', 0, 'zero'), ('ONE', 1, 'one'), ('TWO', 2, 'two'), ('THREE', 3,
↪'three'), ('FOUR', 4, 'four')]
>>> MY_CHOICES.BIG
[('THREE', 3, 'three'), ('FOUR', 4, 'four')]
```

Raises

- **RuntimeError** – When the `Choices` instance is marked as not mutable, which is the case for subsets.
- **ValueError** –
  - if the subset name is defined as first argument and as named argument. * if some constants have the same name or the same value. * if at least one constant or value already exists in the instance.

add_subset (*name*, *constants*)

Add a subset of entries under a defined name.

This allow to defined a "sub choice" if a django field need to not have the whole choice available.

The sub-choice is a new `Choices` instance, with only the wanted the constant from the main `Choices` (each "choice entry" in the subset is shared from the main `Choices`) The sub-choice is accessible from the main `Choices` by an attribute having the given name.

Parameters

- **name** (*string*) – Name of the attribute that will old the new `Choices` instance.

- **constants** (*list or tuple*) – List of the constants name of this `Choices` object to make available in the subset.

Returns The newly created subset, which is a `Choices` object

Return type *Choices*

Example

```
>>> STATES = Choices(
...     ('ONLINE',  1, 'Online'),
...     ('DRAFT',   2, 'Draft'),
...     ('OFFLINE', 3, 'Offline'),
... )
>>> STATES
[('ONLINE', 1, 'Online'), ('DRAFT', 2, 'Draft'), ('OFFLINE', 3, 'Offline')]
>>> STATES.add_subset('NOT_ONLINE', ('DRAFT', 'OFFLINE',))
>>> STATES.NOT_ONLINE
[('DRAFT', 2, 'Draft'), ('OFFLINE', 3, 'Offline')]
>>> STATES.NOT_ONLINE.DRAFT
2
>>> STATES.NOT_ONLINE.for_constant('DRAFT') is STATES.for_constant('DRAFT')
True
>>> STATES.NOT_ONLINE.ONLINE
Traceback (most recent call last):
...
AttributeError: 'Choices' object has no attribute 'ONLINE'
```

Raises **ValueError** –

- If `name` is already an attribute of the `Choices` instance. * If a constant is not defined as a constant in the `Choices` instance.

choices

Property that returns a tuple formatted as expected by Django.

Example

```
>>> MY_CHOICES = Choices(('FOO', 1, 'foo'), ('BAR', 2, 'bar'))
>>> MY_CHOICES.choices
((1, 'foo'), (2, 'bar'))
```

extract_subset (**constants*)

Create a subset of entries

This subset is a new `Choices` instance, with only the wanted constants from the main `Choices` (each "choice entry" in the subset is shared from the main `Choices`)

Parameters ***constants** (*list*) – The constants names of this `Choices` object to make available in the subset.

Returns The newly created subset, which is a `Choices` object

Return type *Choices*

Example

```
>>> STATES = Choices(
...     ('ONLINE',  1, 'Online'),
...     ('DRAFT',   2, 'Draft'),
...     ('OFFLINE', 3, 'Offline'),
... )
>>> STATES
[('ONLINE', 1, 'Online'), ('DRAFT', 2, 'Draft'), ('OFFLINE', 3, 'Offline')]
>>> subset = STATES.extract_subset('DRAFT', 'OFFLINE')
>>> subset
[('DRAFT', 2, 'Draft'), ('OFFLINE', 3, 'Offline')]
>>> subset.DRAFT
2
>>> subset.for_constant('DRAFT') is STATES.for_constant('DRAFT')
True
>>> subset.ONLINE
Traceback (most recent call last):
...
AttributeError: 'Choices' object has no attribute 'ONLINE'
```

Raises **ValueError** – If a constant is not defined as a constant in the `Choices` instance.

for_constant(*constant*)

Returns the `ChoiceEntry` for the given constant.

Parameters **constant** (*string*) – Name of the constant for which we want the choice entry.

Returns The instance of `ChoiceEntry` for the given constant.

Return type *ChoiceEntry*

Raises **KeyError** – If the constant is not an existing one.

Example

```
>>> MY_CHOICES = Choices(('FOO', 1, 'foo'), ('BAR', 2, 'bar'))
>>> MY_CHOICES.for_constant('FOO')
('FOO', 1, 'foo')
>>> MY_CHOICES.for_constant('FOO').value
1
>>> MY_CHOICES.for_constant('QUX')
Traceback (most recent call last):
...
KeyError: 'QUX'
```

for_display(*display*)

Returns the `ChoiceEntry` for the given display name.

Parameters **display** (*string*) – Display name for which we want the choice entry.

Returns The instance of `ChoiceEntry` for the given display name.

Return type *ChoiceEntry*

Raises **KeyError** – If the display name is not an existing one.

Example

```
>>> MY_CHOICES = Choices(('FOO', 1, 'foo'), ('BAR', 2, 'bar'))
>>> MY_CHOICES.for_display('foo')
('FOO', 1, 'foo')
>>> MY_CHOICES.for_display('foo').constant
'FOO'
>>> MY_CHOICES.for_display('qux')
Traceback (most recent call last):
...
KeyError: 'qux'
```

for_value(*value*)

Returns the `ChoiceEntry` for the given value.

Parameters **value** – Value for which we want the choice entry.

Returns The instance of `ChoiceEntry` for the given value.

Return type *ChoiceEntry*

Raises **KeyError** – If the value is not an existing one.

Example

```
>>> MY_CHOICES = Choices(('FOO', 1, 'foo'), ('BAR', 2, 'bar'))
>>> MY_CHOICES.for_value(1)
('FOO', 1, 'foo')
>>> MY_CHOICES.for_value(1).display
'foo'
>>> MY_CHOICES.for_value(3)
Traceback (most recent call last):
...
KeyError: 3
```

has_constant(*constant*)

Check if the current `Choices` object has the given constant.

Parameters **constant** (*string*) – Name of the constant we want to check..

Returns `True` if the constant is present, `False` otherwise.

Return type boolean

Example

```
>>> MY_CHOICES = Choices(('FOO', 1, 'foo'), ('BAR', 2, 'bar'))
>>> MY_CHOICES.has_constant('FOO')
True
>>> MY_CHOICES.has_constant('QUX')
False
```

has_display(*display*)

Check if the current `Choices` object has the given display name.

Parameters **display** (*string*) – Display name we want to check..

Returns `True` if the display name is present, `False` otherwise.

Return type boolean

Example

```
>>> MY_CHOICES = Choices(('FOO', 1, 'foo'), ('BAR', 2, 'bar'))
>>> MY_CHOICES.has_display('foo')
True
>>> MY_CHOICES.has_display('qux')
False
```

has_value(*value*)

Check if the current `Choices` object has the given value.

Parameters **value** – Value we want to check.

Returns `True` if the value is present, `False` otherwise.

Return type boolean

Example

```
>>> MY_CHOICES = Choices(('FOO', 1, 'foo'), ('BAR', 2, 'bar'))
>>> MY_CHOICES.has_value(1)
True
>>> MY_CHOICES.has_value(3)
False
```

class extended_choices.choices.**OrderedChoices**(**choices*, ***kwargs*)

Bases: *extended_choices.choices.Choices*

Simple subclass of `Choices` using `OrderedDict` as `dict_class`

Example

Start by declaring your `Choices`:

```
>>> ALIGNMENTS = OrderedChoices(
...     ('BAD', 10, 'bad'),
...     ('NEUTRAL', 20, 'neutral'),
...     ('CHAOTIC_GOOD', 30, 'chaotic good'),
...     ('GOOD', 40, 'good'),
... )
```

```
>>> ALIGNMENTS.dict_class
<class 'collections.OrderedDict'>
```

```
>>> ALIGNMENTS.constants
OrderedDict([('BAD', ('BAD', 10, 'bad')), ('NEUTRAL', ('NEUTRAL', 20, 'neutral')),
 ↪ ('CHAOTIC_GOOD', ('CHAOTIC_GOOD', 30, 'chaotic good')), ('GOOD', ('GOOD', 40,
 ↪'good'))])
>>> ALIGNMENTS.values
OrderedDict([(10, ('BAD', 10, 'bad')), (20, ('NEUTRAL', 20, 'neutral')), (30, (
 ↪'CHAOTIC_GOOD', 30, 'chaotic good')), (40, ('GOOD', 40, 'good'))])
```

(continues on next page)

(continued from previous page)

```
>>> ALIGNMENTS.displays
OrderedDict([('bad', ('BAD', 10, 'bad')), ('neutral', ('NEUTRAL', 20, 'neutral')),
↪ ('chaotic good', ('CHAOTIC_GOOD', 30, 'chaotic good')), ('good', ('GOOD', 40,
↪'good'))])
```

class extended_choices.choices.**AutoDisplayChoices**(**choices*, ***kwargs*)

Bases: *extended_choices.choices.OrderedChoices*

Subclass of `OrderedChoices` that will compose the display value based on the constant.

To compose the display value, it will call a `display_transform` function, that is defined as a class attribute but can be overridden by passing it to the constructor.

Example

```
>>> ALIGNMENTS = AutoDisplayChoices(
...     ('BAD', 10),
...     ('NEUTRAL', 20),
...     ('CHAOTIC_GOOD', 30, 'THE CHAOS'),
...     ('GOOD', 40, {'additional': 'attributes'}),
... )
```

```
>>> ALIGNMENTS.BAD.display
'Bad'
>>> ALIGNMENTS.NEUTRAL.choice_entry
('NEUTRAL', 20, 'Neutral')
>>> ALIGNMENTS.CHAOTIC_GOOD.display
'THE CHAOS'
>>> ALIGNMENTS.GOOD.choice_entry.additional
'attributes'
```

static display_transform(*const*)

class extended_choices.choices.**AutoChoices**(**choices*, ***kwargs*)

Bases: *extended_choices.choices.AutoDisplayChoices*

Subclass of `AutoDisplayChoices` that will also compose the value to be saved based on the constant.

To compose the display value, it will call a `display_transform` function, that is defined as a class attribute but can be overridden by passing it to the constructor.

In this class, the `*choices` argument can simply be strings, or tuples with one element (or two to add additional attributes)

Example

```
>>> ALIGNMENTS = AutoChoices(
...     'BAD',
...     ('NEUTRAL', ),
...     ('CHAOTIC_GOOD', 'chaos', 'THE CHAOS'),
...     ('GOOD', None, 'Yeah', {'additional': 'attributes'}),
... )
```

```
>>> ALIGNMENTS.BAD.value
'bad'
>>> ALIGNMENTS.BAD.display
'Bad'
>>> ALIGNMENTS.NEUTRAL.choice_entry
('NEUTRAL', 'neutral', 'Neutral')
>>> ALIGNMENTS.CHAOTIC_GOOD.value
'chaos'
>>> ALIGNMENTS.CHAOTIC_GOOD.display
'THE CHAOS'
>>> ALIGNMENTS.GOOD.value
'good'
>>> ALIGNMENTS.GOOD.display
'Yeah'
>>> ALIGNMENTS.GOOD.choice_entry.additional
'attributes'
```

add_choices(**choices*, ***kwargs*)
Disallow super method to thing the first argument is a subset name

static value_transform(*const*)

2.3 extended_choices.fields module

Provides a form field for django to use constants instead of values as available values.

Notes

The documentation format in this file is numpydoc.

class extended_choices.fields.**NamedExtendedChoiceFormField**(*choices*, **args*, ***kwargs*)

Bases: django.forms.fields.Field

Field to use with choices where values are constant names instead of choice values.

Should not be very useful in normal HTML form, but if API validation is done via a form, it will to have more readable constants in the API that values

to_python(*value*)
Convert the constant to the real choice value.

2.4 extended_choices.helpers module

Provides classes used to construct a full Choices instance.

Notes

The documentation format in this file is numpydoc.

class extended_choices.helpers.**ChoiceAttributeMixin**(*value*, *choice_entry*)
Bases: future.types.newobject.newobject

Base class to represent an attribute of a `ChoiceEntry`.

Used for `constant`, `name`, and `display`.

It must be used as a mixin with another type, and the final class will be a type with added attributes to access the `ChoiceEntry` instance and its attributes.

choice_entry
The `ChoiceEntry` instance that hold the current value, used to access its constant, value and display name.

Type instance of `ChoiceEntry`

constant
Returns the choice field holding the constant of the attached `ChoiceEntry`.

Type property

value
Returns the choice field holding the value of the attached `ChoiceEntry`.

Type property

display
Returns the choice field holding the display name of the attached `ChoiceEntry`.

Type property

original_value
The value used to create the current instance.

Type ?

creator_type
The class that created a new class. Will be `ChoiceAttributeMixin` except if it was overridden by the author.

Type type

Example

Classes can be created manually:

```
>>> class IntChoiceAttribute(ChoiceAttributeMixin, int): pass
>>> field = IntChoiceAttribute(1, ChoiceEntry(('FOO', 1, 'foo')))
>>> field
1
>>> field.constant, field.value, field.display
('FOO', 1, 'foo')
>>> field.choice_entry
('FOO', 1, 'foo')
```

Or via the `get_class_for_value` class method:

```
>>> klass = ChoiceAttributeMixin.get_class_for_value(1.5)
>>> klass.__name__
'FloatChoiceAttribute'
>>> float in klass.mro()
True
```

constant
Property that returns the `constant` attribute of the attached `ChoiceEntry`.

display
Property that returns the `display` attribute of the attached `ChoiceEntry`.

classmethod get_class_for_value(*value*)
Class method to construct a class based on this mixin and the type of the given value.

Parameters **value** – The value from which to extract the type to create the new class.

Notes

The create classes are cached (in `cls.__classes_by_type`) to avoid recreating already created classes.

value
Property that returns the `value` attribute of the attached `ChoiceEntry`.

class extended_choices.helpers.**ChoiceEntry**
Bases: `tuple`

Represents a choice in a `Choices` object, with easy access to its attribute.

Expecting a tuple with three entries. (constant, value, display name), it will add three attributes to access then: `constant`, `value` and `display`.

By passing a dict after these three first entries, in the tuple, it's also possible to add some other attributes to the `ChoiceEntry` instance.

Parameters **tuple** (*tuple*) – A tuple with three entries, the name of the constant, the value, and the display name. A dict could be added as a fourth entry to add additional attributes.

Example

```
>>> entry = ChoiceEntry(('FOO', 1, 'foo'))
>>> entry
('FOO', 1, 'foo')
>>> (entry.constant, entry.value, entry.display)
('FOO', 1, 'foo')
>>> entry.choice
(1, 'foo')
```

You can also pass attributes to add to the instance to create:

```
>>> entry = ChoiceEntry(('FOO', 1, 'foo', {'bar': 1, 'baz': 2}))
>>> entry
('FOO', 1, 'foo')
>>> entry.bar
1
>>> entry.baz
2
```

Raises **AssertionError** – If the number of entries in the tuple is not expected. Must be 3 or 4.

class ChoiceAttributeMixin(*value*, *choice_entry*)

Bases: `future.types.newobject.newobject`

Base class to represent an attribute of a `ChoiceEntry`.

Used for `constant`, `name`, and `display`.

It must be used as a mixin with another type, and the final class will be a type with added attributes to access the `ChoiceEntry` instance and its attributes.

choice_entry

The `ChoiceEntry` instance that hold the current value, used to access its constant, value and display name.

Type instance of `ChoiceEntry`

constant

Returns the choice field holding the constant of the attached `ChoiceEntry`.

Type property

value

Returns the choice field holding the value of the attached `ChoiceEntry`.

Type property

display

Returns the choice field holding the display name of the attached `ChoiceEntry`.

Type property

original_value

The value used to create the current instance.

Type ?

creator_type

The class that created a new class. Will be `ChoiceAttributeMixin` except if it was overridden by the author.

Type type

Example

Classes can be created manually:

```
>>> class IntChoiceAttribute(ChoiceAttributeMixin, int): pass
>>> field = IntChoiceAttribute(1, ChoiceEntry(('FOO', 1, 'foo')))
>>> field
1
>>> field.constant, field.value, field.display
('FOO', 1, 'foo')
>>> field.choice_entry
('FOO', 1, 'foo')
```

Or via the `get_class_for_value` class method:

```
>>> klass = ChoiceAttributeMixin.get_class_for_value(1.5)
>>> klass.__name__
'FloatChoiceAttribute'
>>> float in klass.mro()
True
```

constant

Property that returns the `constant` attribute of the attached `ChoiceEntry`.

display
Property that returns the `display` attribute of the attached `ChoiceEntry`.

classmethod get_class_for_value(*value*)
Class method to construct a class based on this mixin and the type of the given value.

Parameters **value** – The value from which to extract the type to create the new class.

Notes

The create classes are cached (in `cls.__classes_by_type`) to avoid recreating already created classes.

value
Property that returns the `value` attribute of the attached `ChoiceEntry`.

extended_choices.helpers.**create_choice_attribute**(*creator_type*, *value*, *choice_entry*)
Create an instance of a subclass of ChoiceAttributeMixin for the given value.

Parameters

- **creator_type** (*type*) – `ChoiceAttributeMixin` or a subclass, from which we'll call the `get_class_for_value` class-method.
- **value** – The value for which we want to create an instance of a new subclass of `creator_type`.
- **choice_entry** (`ChoiceEntry`) – The `ChoiceEntry` instance that hold the current value, used to access its constant, value and display name.

Returns An instance of a subclass of `creator_type` for the given value

Return type *ChoiceAttributeMixin*

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

e

Index

N

O

T

V