# Django utils2 Documentation

***Release 2.8.0***

**Rick van Hattem**

**Mar 22, 2020**

# Contents

Contents:

CHAPTER 1

# Introduction

Travis status:


Coverage:


Django Utils is a collection of small Django helper functions, utilities and classes which make common patterns shorter and easier. It is by no means a complete collection but it has served me quite a bit in the past and I will keep extending it.

Examples are:

- Enum based choicefields
- Models with automatic *__str__*, *__unicode__* and *__repr__* functions based on names and/or slugs using simple mixins.
- Models with automatic *updated_at* and *created_at* fields
- Models with automatic slugs based on the *name* property.
- Iterating through querysets in predefined chunks to prevent out of memory errors

The library depends on the Python Utils library.

Documentation is available at: http://django-utils-2.readthedocs.io/en/latest/

## 1.1 Install

To install:

1. Run *pip install django-utils2* or execute *python setup.py install* in the source directory
2. Add *django_utils* to your *INSTALLED_APPS*

If you want to run the tests, run *py.test* (requirements in *tests/requirements.txt*)

## 1.2 Links

- **Documentation**
  - http://django-utils-2.readthedocs.org/en/latest/
- **Source**
  - https://github.com/WoLpH/django-utils
- **Bug reports**
  - https://github.com/WoLpH/django-utils/issues
- **Package homepage**
  - https://pypi.python.org/pypi/django-utils2
- **My blog**
  - http://w.wol.ph/

CHAPTER 2

# django_utils package

## 2.1 Subpackages

### 2.1.1 django_utils.management package

#### 2.1.1.1 Subpackages

##### 2.1.1.1.1 django_utils.management.commands package

###### 2.1.1.1.1.1 Submodules

###### 2.1.1.1.1.2 django_utils.management.commands.admin_autogen module

**class** django_utils.management.commands.admin_autogen.**Command**
Bases: *django_utils.management.commands.base_command.CustomBaseCommand*

**handle**(**args*, ***kwargs*)

###### 2.1.1.1.1.3 django_utils.management.commands.base_command module

**class** django_utils.management.commands.base_command.**CustomAppCommand**
Bases: *django_utils.management.commands.base_command.CustomBaseCommand*, django.core.management.base.AppCommand

**class** django_utils.management.commands.base_command.**CustomBaseCommand**
Bases: django.core.management.base.BaseCommand, python_utils.logger.Logged

**create_logger**()

**handle**(**args*, ***kwargs*)

**loggers = ()**

#### 2.1.1.1.1.4 django_utils.management.commands.settings module

**class** django_utils.management.commands.settings.**Command**
Bases: *django_utils.management.commands.base_command.CustomBaseCommand*

**add_arguments**(*parser*)

**can_import_settings = True**

**handle**(**args*, ***options*)

**help = 'Get a list of the current settings, any arguments given will be\n used to matc**

**output_types = ['pprint', 'print', 'json', 'csv']**

**render_output**(*data*, *output_type='pprint'*, *show_keys=False*, ***options*)

**requires_model_validation = False**

django_utils.management.commands.settings.**json_default**(*obj*)

#### 2.1.1.1.1.5 Module contents

### 2.1.1.2 Module contents

## 2.1.2 django_utils.templatetags package

### 2.1.2.1 Submodules

### 2.1.2.2 django_utils.templatetags.debug module

**class** django_utils.templatetags.debug.**Formatter**(*max_depth=3*)
Bases: django_utils.templatetags.debug._Formatter

**MAX_LENGTH = 100**

**MAX_LENGTH_DOTS = 3**

**format**(*value*, *depth*, *show_protected*, *show_special*)
Call the formatter with the given value to format and optional depth

```
>>> formatter = Formatter()
>>> class Eggs: pass
>>> formatter(Eggs)
'<Eggs {}>'
```

**format_datetime**(*value*, *depth*, *show_protected*, *show_special*)
Format a date

**Parameters**

- **value** – a date to format
- **depth** – the current depth

**Returns** a formatted string

```
>>> formatter = Formatter()
>>> formatter(datetime.date(2000, 1, 2))
'<date:2000-01-02>'
>>> formatter(datetime.datetime(2000, 1, 2, 3, 4, 5, 6))
'<datetime:2000-01-02 03:04:05.000006>'
```

**format_dict**(*value*, *depth*, *show_protected*, *show_special*)
Format a string

**Parameters**

- **value** – a str value to format
- **depth** – the current depth

**Returns** a formatted string

```
>>> formatter = Formatter()
>>> formatter({'a': 1, 'b': 2}, 5)
'{a: 1, b: 2}'
```

**format_int**(*value*, *depth*, *show_protected*, *show_special*)
Format an integer/long

**Parameters**

- **value** – an int/long to format
- **depth** – the current depth

**Returns** a formatted string

```
>>> formatter = Formatter()
>>> str(formatter(1, 0))
'1'
>>> formatter(1, 1)
'1'
```

**format_list**(*value*, *depth*, *show_protected*, *show_special*)
Format a string

**Parameters**

- **value** – a list to format
- **depth** – the current depth

**Returns** a formatted string

```
>>> formatter = Formatter()
>>> formatter(list(range(5)))
'[0, 1, 2, 3, 4]'
```

**format_model**(*value*, *depth*, *show_protected*, *show_special*)
Format a string

**Parameters**

- **value** – a str value to format
- **depth** – the current depth

**Returns** a formatted string

```
>>> formatter = Formatter()
>>> from django.contrib.auth.models import User
>>> user = User()
>>> del user.date_joined
>>> str(formatter(user, 5, show_protected=False)[:30])
'<User {email: , first_name: , '
```

**format_object**(*value*, *depth*, *show_protected*, *show_special*)
Format an object

**Parameters**

- **value** – an object to format
- **depth** – the current depth

**Returns** a formatted string

```
>>> formatter = Formatter()
>>> original_max_length = formatter.MAX_LENGTH
>>> formatter.MAX_LENGTH = 50
```

```
>>> class Spam(object):
...     x = 1
...     _y = 2
...     __z = 3
...     __hidden_ = 4
>>> spam = Spam()
```

```
>>> str(formatter(spam, show_protected=True, show_special=True))
'<Spam {x: 1, _Spam__hidden_: 4, _Spam__z: 3, __dict__:...}>'
>>> str(formatter(spam, show_protected=False, show_special=False))
'<Spam {x: 1}>'
```

```
>>> formatter.MAX_LENGTH = original_max_length
```

**format_str**(*value*, *depth*, *show_protected*, *show_special*)
Format a string

**Parameters**

- **value** – a str value to format
- **depth** – the current depth

**Returns** a formatted string

```
>>> formatter = Formatter()
>>> str(formatter('test'))
'test'
>>> str(formatter(six.b('test')))
'test'
```

**format_unicode**(*value*, *depth*, *show_protected*, *show_special*)
Format a string

**Parameters**

- **value** – a unicode value to format
- **depth** – the current depth

**Returns** a formatted string

```
>>> formatter = Formatter()
>>> original_max_length = formatter.MAX_LENGTH
>>> formatter.MAX_LENGTH = 10
>>> str(formatter('x' * 11))
'xxxxxxx...'
>>> formatter.MAX_LENGTH = original_max_length
```

django_utils.templatetags.debug.**debug**(*value*, *max_depth=3*)

Debug template filter to print variables in a pretty way

```
>>> str(debug(123).strip())
'<pre style="border: 1px solid #fcc; background-color: #ccc;">123</pre>'
```

#### 2.1.2.3 Module contents

## 2.2 Submodules

## 2.3 django_utils.base_models module

**class** django_utils.base_models.**CreatedAtModelBase**(**args*, ***kwargs*)

Bases: *django_utils.base_models.ModelBase*

**class Meta**

**abstract = False**

**db_table = 'django_utils_created_at_model_base'**

**created_at**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**get_next_by_created_at**(***morekwargs*)

**get_next_by_updated_at**(***morekwargs*)

**get_previous_by_created_at**(***morekwargs*)

**get_previous_by_updated_at**(***morekwargs*)

**updated_at**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**class** django_utils.base_models.**ModelBase**(**args*, ***kwargs*)

Bases: django.db.models.base.Model

**class Meta**

**abstract = False**

**db_table = 'django_utils_model_base'**

**class** django_utils.base_models.**ModelBaseMeta**
Bases: django.db.models.base.ModelBase

Model base with more readable naming convention

Example: Assuming the model is called *app.FooBarObject*

Default Django table name: *app_foobarobject* Table name with this base: *app_foo_bar_object*

**class** django_utils.base_models.**NameCreatedAtModelBase**(**args*, ***kwargs*)
Bases: *django_utils.base_models.NameModelBase*, *django_utils.base_models.CreatedAtModelBase*

**class Meta**

**abstract = False**

**db_table = 'django_utils_name_created_at_model_base'**

**get_next_by_created_at**(***morekwargs*)

**get_next_by_updated_at**(***morekwargs*)

**get_previous_by_created_at**(***morekwargs*)

**get_previous_by_updated_at**(***morekwargs*)

**class** django_utils.base_models.**NameMixin**
Bases: object

Mixin to automatically get a unicode and repr string base on the name

```
>>> x = NameMixin()
>>> x.pk = 123
>>> x.name = 'test'
>>> repr(x)
'<NameMixin[123]: test>'
>>> str(x)
'test'
>>> str(six.text_type(x))
'test'
```

**class** django_utils.base_models.**NameModelBase**(**args*, ***kwargs*)
Bases: *django_utils.base_models.NameMixin*, *django_utils.base_models.ModelBase*

**class Meta**

**abstract = False**

**db_table = 'django_utils_name_model_base'**

**name**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**class** django_utils.base_models.**SlugCreatedAtModelBase**(**args*, ***kwargs*)
Bases: *django_utils.base_models.SlugModelBase*, *django_utils.base_models.CreatedAtModelBase*

**class Meta**

**abstract = False**

```
        db_table = 'django_utils_slug_created_at_model_base'
```

**get_next_by_created_at**(***morekwargs*)

**get_next_by_updated_at**(***morekwargs*)

**get_previous_by_created_at**(***morekwargs*)

**get_previous_by_updated_at**(***morekwargs*)

**class** django_utils.base_models.**SlugMixin**

Bases: *django_utils.base_models.NameMixin*

Mixin to automatically slugify the name and add both a name and slug to the model

```
>>> x = NameMixin()
>>> x.pk = 123
>>> x.name = 'test'
>>> repr(x)
'<NameMixin[123]: test>'
>>> str(x)
'test'
>>> str(six.text_type(x))
'test'
```

**class Meta**

Bases: object

```
        unique_together = ('slug',)
```

**save**(**args*, ***kwargs*)

**class** django_utils.base_models.**SlugModelBase**(**args*, ***kwargs*)

Bases: *django_utils.base_models.SlugMixin*, *django_utils.base_models.NameModelBase*

**class Meta**

```
        abstract = False

        db_table = 'django_utils_slug_model_base'
```

**slug**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

## 2.4 django_utils.choices module

### 2.4.1 Usage

Create a *Choices* class and add *Choice* objects to the class to define your choices.

#### 2.4.1.1 Example with explicit values:

The normal Django version:

```
class Human(models.Model):
    GENDER = (
        ('m', 'Male'),
        ('f', 'Female'),
        ('o', 'Other'),
    )
    gender = models.CharField(max_length=1, choices=GENDER)
```

The Django Utils Choices version:

```
from django_utils import choices

class Human(models.Model):
    class Gender(choices.Choices):
        Male = choices.Choice('m')
        Female = choices.Choice('f')
        Other = choices.Choice('o')

    gender = models.CharField(max_length=1, choices=Gender)
```

To reference these properties:

```
Human.create(gender=Human.Gender.Male)
```

#### 2.4.1.2 Example with implicit values:

The normal Django version:

```
class SomeModel(models.Model):
    SOME_ENUM = (
        (1, 'foo'),
        (2, 'bar'),
        (3, 'spam'),
        (4, 'eggs'),
    )
    enum = models.IntegerField(choices=SOME_ENUM, default=1)
```

The Django Utils Choices version:

```
from django_utils import choices

class SomeModel(models.Model):
    class Enum(choices.Choices):
        Foo = choices.Choice()
        Bar = choices.Choice()
        Spam = choices.Choice()
        Eggs = choices.Choice()

    enum = models.IntegerField(
        choices=Enum, default=Enum.Foo)
```

To reference these properties:

```
SomeModel.create(enum=SomeModel.Enum.Spam)
```

**class** django_utils.choices.**Choice**(*value=None*, *label=None*)

Bases: object

The choice object has an optional label and value. If the value is not given an autoincrementing id (starting from 1) will be used

```
>>> choice = Choice('value', 'label')
>>> choice
<Choice[1]:label>
>>> str(choice)
'label'
```

```
>>> choice = Choice()
>>> choice
<Choice[2]:None>
>>> str(choice)
'None'
```

**deconstruct**()

**order = 0**

**class** django_utils.choices.**Choices**

Bases: object

The choices class is what you should inherit in your Django models

```
>>> choices = Choices()
>>> choices.choices[0]
Traceback (most recent call last):
...
KeyError: 'Key 0 does not exist'
>>> choices.choices
OrderedDict()
>>> str(choices.choices)
'OrderedDict()'
>>> choices.choices.items()
[]
>>> choices.choices.keys()
[]
>>> choices.choices.values()
[]
>>> list(choices)
[]
```

```
>>> class ChoiceTest(Choices):
...     a = Choice()
>>> choices = ChoiceTest()
>>> choices.choices.items()
[(0, <Choice[...]:a>)]
>>> choices.a
0
>>> choices.choices['a']
<Choice[...]:a>
>>> choices.choices[0]
<Choice[...]:a>
>>> choices.choices.keys()
[0]
>>> choices.choices.values()
['a']
>>> list(choices)
```

(continues on next page)

(continued from previous page)

```
[(0, <Choice[...]:a>)]
>>> list(ChoiceTest)
[(0, <Choice[...]:a>)]
```

**choices = OrderedDict()**

**class** django_utils.choices.**ChoicesDict**

Bases: object

The choices dict is an object that stores a sorted representation of the values by key and database value

**items**()

**keys**()

**values**()

**class** django_utils.choices.**ChoicesMeta**

Bases: type

The choices metaclass is where all the magic happens, this automatically creates a ChoicesDict to get a sorted list of keys and values

**class** django_utils.choices.**LiteralChoices**

Bases: *django_utils.choices.Choices*

Special version of the Choices class that uses the label as the value

```
>>> class Role(LiteralChoices):
...     admin = Choice()
...     user = Choice()
...     guest = Choice()
```

```
>>> Role.choices.values()
['admin', 'user', 'guest']
>>> Role.choices.keys()
['admin', 'user', 'guest']
```

**choices = OrderedDict()**

## 2.5 django_utils.fields module

**class** django_utils.fields.**RecursiveField**(*field_name=None*, *parent_field='parent'*, *default=None*)

Bases: object

**PREFIX = 'get_'**

**contribute_to_class**(*cls*, *name*)

**get**(*instance*)

## 2.6 django_utils.queryset module

django_utils.queryset.**queryset_iterator**(*queryset*, *chunksize=1000*, *getfunc=<built-in function getattr>*)

'' Iterate over a Django Queryset ordered by the primary key

This method loads a maximum of chunksize (default: 1000) rows in it's memory at the same time while django normally would load all rows in it's memory. Using the iterator() method only causes it to not preload all the classes.

Note that the implementation of the iterator does not support ordered query sets.

## 2.7 django_utils.utils module

django_utils.utils.**to_json**(*request*, *data*)

## 2.8 django_utils.view_decorators module

**exception** django_utils.view_decorators.**UnknownViewResponseError**
Bases: *django_utils.view_decorators.ViewError*

**exception** django_utils.view_decorators.**ViewError**
Bases: exceptions.Exception

django_utils.view_decorators.**env**(*function=None*, *login_required=False*, *response_class=<class 'django.http.response.HttpResponse'>*)
View decorator that automatically adds context and renders response

Keyword arguments: login_required – is everyone allowed or only authenticated users

Adds a RequestContext (request.context) with the following context items: name – current function name

Stores the template in request.template and assumes it to be in <app>/<view>.html

django_utils.view_decorators.**json_default_handler**(*obj*)

django_utils.view_decorators.**permanent_redirect**(*url*, **args*, ***kwargs*)

django_utils.view_decorators.**redirect**(*url='./'*, **args*, ***kwargs*)

## 2.9 django_utils.views module

django_utils.views.**error_403**(*request*, **args*, ***kwargs*)

django_utils.views.**error_404**(*request*, **args*, ***kwargs*)

django_utils.views.**error_500**(*request*, **args*, ***kwargs*)

## 2.10 Module contents

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## d

# Index