Dirty Models Documentation

Release 0.10.1

alfred82santa, tmarques82, padajuan, xejarque, oarnau

Nov 02, 2017

Contents

CHAPTER 1

Dirty Models

Dirty models for python 3

1.1 Documentation

http://dirty-models.readthedocs.io/

1.2 Features

- Python 3 package.
- Easy to create a model.
- Non destructive modifications.
- Non false positive modifications.
- Able to restore original data for each field or whole model.
- Access to original data.
- Read only fields.
- Alias for fields.
- Custom getters and setters for each fields.
- Automatic cast value.
- Easy import from/export to dict.
- Basic field type implemented.
- Multi type fields.
- Default values for each field or whole model.

- HashMap model. It could be used instead of DynamicModel.
- FastDynamicModel. It could be used instead of DynamicModel. Same behavior, better performance.
- Pickable models.
- Datetime fields can use any datetime format using parser and formatter functions.
- No database dependent.
- Auto documentation using Dirty Models Sphinx extension.
- Json encoder.
- Field access like dictionary but with wildcards.
- Opensource (BSD License)

1.3 Changelog

1.3.1 Version 0.10.1

- *Factory* feature. It allows to define a factory as default value in order to be executed each time model is instanced. (Issue #100)

```
from dirty_models.utils import factory
from datetime import datetime

class Model(BaseModel):

   field_1 = DateTimeField(default=factory(datetime.now))

model = Model()
print(model.field_1)

# 2017-11-02 21:52:46.339040
```

- Makefile fixes.
- Python 3.6 is supported officially. It works since first day, but now tests run on Travis for Python 3.6.

1.3.2 Version 0.10.0

- Pickable lists.
- Improved pickle performance.
- Setting `None` to a field remove content.
- More tests.
- Some code improvements.

1.3.3 Version 0.9.2

- Fix timezone when convert timestamp to datetime.

1.3.4 Version 0.9.1

- Fix installation.

1.3.5 Version 0.9.0

- New EnumField.
- Fixes on setup.py.
- Fixes on requirements.
- Fixes on formatter iters.
- Fixes on code.
- Added `__version__` to main package file.
- Synchronized version between main packege file, `setup.py` and docs.
- Export only modifications.

1.3.6 Version 0.8.1

- Added __contains__ function to models and lists. It allows to use `in` operator.
- Added `default_timezone` parameter to DateTimeFields and TimeFields. If value entered has no a timezone defined, default one will be set.
- Added `force_timezone` parameter to DateTimeFields in order to convert values to a specific timezone.
- More cleanups.

1.3.7 Version 0.8.0

- Renamed internal fields. Now they use double score format `__fieldname__`.
- Raise a RunTimeError exception if two fields use same alias in a model.
- Fixed default docstrings.
- Cleanup default data. Only real name fields are allowed to use as key.
- Added *`get_attrs_by_path()`* in order to get all values using path.
- Added *`get_1st_attr_by_path()`* in order to get first value using path.
- Added option to access fields like in a dictionary, but using wildcards. Only for getters. See: *`get_1st_attr_by_path()`*.
- Added some documentation.

1.3.8 Version 0.7.2

- Fixed inherited structure
- Added `get_default_data` method to models in order to retrieve default data.

1.3.9 Version 0.7.1

- Solved problem formatting dynamic models
- Added date, time and timedelta fields to dynamic models.

1.3.10 Version 0.7.0

- Timedelta field
- Generic formatters
- Json encoder

```
import json
from datetime import datetime
from dirty_models import BaseModel, DatetimeField
from dirty_models.utils import JSONEncoder


class ExampleModel(BaseModel):
    field_datetime = DatetimeField(parse_format="%Y-%m-%dT%H:%M:%S")

model = ExampleModel(field_datetime=datetime.now())

assert json.dumps(model, cls=JSONEncoder) == '{"field_datetime": "2016-05-30T22:22:22
↪"}'
```

- Auto camelCase fields metaclass

1.3.11 Version 0.6.3

- Documentation fixed.
- Allow import main members from root package.

1.3.12 Version 0.6.2

- Improved datetime fields parser and formatter definitions. Now there are three ways to define them:
- Format string to use both parse and formatter:

```
class ExampleModel(BaseModel):
    datetime_field = DateTimeField(parse_format='%Y-%m-%dT%H:%M:%SZ')
```

- Define a format string or function for parse and format datetime:

```
class ExampleModel(BaseModel):
    datetime_field = DateTimeField(parse_format={'parser': callable_func,
                                                 'formatter': '%Y-%m-%dT%H:%M:%SZ'})
```

- Use predefined format:

```
DateTimeField.date_parsers = {
    'iso8061': {
        'formatter': '%Y-%m-%dT%H:%M:%SZ',
```

```
        'parser': iso8601.parse_date
    }
}
class ExampleModel(BaseModel):
    datetime_field = DateTimeField(parse_format='iso8061')
```

1.3.13 Version 0.6.1

- Improved model field autoreference.

```
class ExampleModel(BaseModel):
    model_field = ModelField()  # Field with a ExampleModel
    array_of_model = ArrayField(field_type=ModelField())  # Array of ExampleModels
```

1.3.14 Version 0.6.0

- Added default value for fields.

```
class ExampleModel(BaseModel):
    integer_field = IntegerField(default=1)

model = ExampleModel()
assert model.integer_field is 1
```

- Added default values at model level. Inherit default values could be override on new model classes.

```
class InheritExampleModel(ExampleModel):
    __default_data__ = {'integer_field': 2}

model = InheritExampleModel()
assert model.integer_field is 2
```

- Added multi type fields.

```
class ExampleModel(BaseModel):
    multi_field = MultiTypeField(field_types=[IntegerField(), StringField()])

model = ExampleModel()
model.multi_field = 2
assert model.multi_field is 2

model.multi_field = 'foo'
assert model.multi_field is 'foo'
```

1.3.15 Version 0.5.2

- Fixed model structure.
- Makefile helpers.

1.3.16 Version 0.5.1

- Added a easy way to get model structure. It will be used by autodoc libraries as sphinx or json-schema.

1.3.17 Version 0.5.0

- Added autolist parameter to ArrayField. It allows to assign a single item to a list field, so it will be converted to a list with this value.

```
class ExampleModel(BaseModel):
    array_field = ArrayField(field_type=StringField(), autolist=True)

model = ExampleModel()
model.array_field = 'foo'
assert model.array_field[0] is 'foo'
```

1.4 Installation

```
$ pip install dirty-models
```

1.5 Issues

- Getter and setter feature needs refactor to be able to use as decorators.
- DynamicModel is too strange. I don't trust in it. Try to use HashMapModel or FastDynamicModel.

1.6 Basic usage

```
from dirty_models.models import BaseModel
from dirty_models.fields import StringField, IntegerField

class FooBarModel(BaseModel):
    foo = IntegerField()
    bar = StringField(name="real_bar")
    alias_field = IntegerField(alias=['alias1', 'alias2'])



fb = FooBarModel()

fb.foo = 2
assert fb.foo is 2

fb.bar = 'wow'
assert fb.bar is 'wow'
assert fb.real_bar is 'wow'

fb.alias_field = 3
assert fb.alias_field is 3
```

```
assert fb.alias1 is fb.alias_field
assert fb.alias2 is fb.alias_field
assert fb['alias_field'] is 3
```

Note: More examples and documentation on http://dirty-models.readthedocs.io/

CHAPTER 2

Getting started

2.1 About

Dirty Model is a Python library to define transactional models. It means a model itself has no functionality. It just defines a structure in order to store data. It is almost true, but it doesn't. A Dirty Model has some functionality: it could be modified storing changes. This is the main propose of this library.

2.2 How to define a model

To define a model is a simple task. You may create a new model class which inherit from *dirty_models.models.BaseModel* and use our field descriptors to define your fields.

```
class MyModel(BaseModel):

    my_int_field = IntegerField()
    my_string_field = StringField()
```

It is all! You has a new model.

Dirty Models defines an useful set of field descriptors to store any type of data: integer, string, non-empty string, float, date, time, datetime, timedelta, model, list of anything, hashmap, dynamic data, etc. You see all of them in *Fields types*.

All of them defines some common parameters on constructor:

- `default` defines default value when a new model is instanced.

```
class MyModel(BaseModel):

    my_int_field = IntegerField(default=3)
    my_string_field = StringField()

model = MyModel()
```

```
assert model.my_int_field == 3 # True
```

- `alias` defines a list of alias for field. Alias could be used like regular field there is no differences. It is not a good practice to define alias for same data, but it useful in some scenarios.

```
class MyModel(BaseModel):

    my_int_field = IntegerField(default=3, alias=['integer_field'])
    my_string_field = StringField()

model = MyModel()

assert model.my_int_field == 3 # True
assert model.integer_field == 3 # True
```

- `name` defines real field name. It will be used on export data for example. Some time you need to define a field with weird characters to fit to a third party API. So, you could define a real name with this parameter. If it is not defined, name used on model definition is assumed as real name. Otherwise if it is defined, name defined on model become an alias for field.

```
class MyModel(BaseModel):

    my_int_field = IntegerField(default=3, name='real_integer_field')
    my_string_field = StringField()

model = MyModel()

assert model.my_int_field == 3 # True
assert model.real_integer_field == 3 # True

print(model.export_data())
# Prints
# {'real_integer_field': 3}
```

- `read_only` defines whether field could be modified (easily). Of course, there are ways to modify it, but they must be used explicitly. See *Unlocker*.

```
class MyModel(BaseModel):

    my_int_field = IntegerField(default=3, read_only=True)
    my_string_field = StringField()

model = MyModel()

assert model.my_int_field == 3 # True

# Non read only field
model.my_string_field = 'string'
assert model.my_string_field == 'string' # True

# Read only field
model.my_int_field = 4
assert model.my_int_field == 4 # False
assert model.my_int_field == 3 # True
```

- `doc` allows to define field docstring programmatically. But, don't worry, you could use docstrings on regular way.

- `getter` allows to define a function to get value.
- `setter` allows to define a function to set value.

2.3 How to set data

There are some ways to set data in models.

2.3.1 Assign value to a field

Probably the most easy is just assigning value to field:

```
class MyModel(BaseModel):

    my_int_field = IntegerField(default=3, read_only=True)
    my_string_field = StringField()

model = MyModel()

model.my_int_field = 3
assert model.my_int_field == 3 # True
```

Be aware, Dirty Model will try to cast value to field type. It means that you could assign string value `'3'` to a integer field and it will be cast to `3`. If value could not be cast it will be ignored. `None` is a particular value, it removes data from field.

```
class MyModel(BaseModel):

    my_int_field = IntegerField()
    my_string_field = StringField()

model = MyModel()

# Automatic cast
model.my_int_field = '3'
assert model.my_int_field == 3 # True
assert model.my_int_field == '3' # False

# Using None to remove data
model.my_int_field = None
assert model.my_int_field is None # True
```

2.3.2 Set data for whole model on contructor

Dictionary could be cast to model on contructor:

```
class MyModel(BaseModel):

    my_int_field = IntegerField()
    my_string_field = StringField()

model = MyModel(data={'my_int_field': 3, 'my_string_field': 'string'})
```

```
assert model.my_int_field == 3 # True
assert model.my_string_field == 'string' # True
```

On the other hand you could use keyword arguments to set some fields:

```
class MyModel(BaseModel):

    my_int_field = IntegerField()
    my_string_field = StringField()

model = MyModel(my_int_field=3, my_string_field='string')

assert model.my_int_field == 3 # True
assert model.my_string_field == 'string' # True
```

2.3.3 Import data

Some time you want to set data to whole model, but model already exists, so you could import data:

```
class MyModel(BaseModel):

    my_int_field = IntegerField()
    my_string_field = StringField()

model = MyModel()

model.import_data({'my_int_field': 3, 'my_string_field': 'string'})

assert model.my_int_field == 3 # True
assert model.my_string_field == 'string' # True
```

2.4 How to get data

In the same way, there are several methods to get data from model.

2.4.1 Use data from field

It is the simplest way to get data. Just use field.

```
class MyModel(BaseModel):

    my_int_field = IntegerField()
    my_string_field = StringField()

model = MyModel()

model.my_int_field = 3

assert model.my_int_field == 3 # True
```

2.4.2 Export data

It is possible to export data to a dict.

```
class MyModel(BaseModel):

    my_int_field = IntegerField()
    my_string_field = StringField()

model = MyModel()

model.my_int_field = 3

print(model.export_data())
# {'my_int_field': 3}
```

2.5 How to remove data

Once more, there are two way to remove data.

2.5.1 Using `del` keyword

Simplest way to remove data from field is to use `del` python keyword.

```
class MyModel(BaseModel):

    my_int_field = IntegerField()
    my_string_field = StringField()

model = MyModel()

model.my_int_field = 3
del model.my_int_field

assert model.my_int_field is None # True
```

2.5.2 Use `None` as value

Other way is to set `None` to field.

```
class MyModel(BaseModel):

    my_int_field = IntegerField()
    my_string_field = StringField()

model = MyModel()

model.my_int_field = 3
model.my_int_field = None

assert model.my_int_field is None # True
```

CHAPTER 3

Dirty Models API

3.1 Models

Base models for dirty_models.

class dirty_models.models.**BaseModel**(*data=None*, *flat=False*, **args*, ***kwargs*)
Bases: *dirty_models.base.BaseData*

Base model with dirty feature. It stores original data and saves modifications in other side.

clear()
Clears all the data in the object, keeping original data

clear_all()
Clears all the data in the object

clear_modified_data()
Clears only the modified data

copy()
Creates a copy of model

delete_attr_by_path(*field_path*)
It deletes fields looked up by field path. Field path is dot-formatted string path: `parent_field.child_field`.

Parameters **field_path** (*str*) – field path. It allows * as wildcard.

delete_field_value(*name*)
Mark this field to be deleted

export_data()
Get the results with the modified_data

export_deleted_fields()
Resturns a list with any deleted fields form original data. In tree models, deleted fields on children will be appended.

export_modifications()
Returns model modifications.

export_modified_data()
Get the modified data

export_original_data()
Get the original data

flat_data()
Pass all the data from modified_data to original_data

get_1st_attr_by_path(*field_path*, ***kwargs*)
It returns first value looked up by field path. Field path is dot-formatted string path: `parent_field.child_field`.

Parameters

- **field_path** (*str*) – field path. It allows `*` as wildcard.
- **default** – Default value if field does not exist. If it is not defined `AttributeError` exception will be raised.

Returns value

get_attrs_by_path(*field_path*, *stop_first=False*)
It returns list of values looked up by field path. Field path is dot-formatted string path: `parent_field.child_field`.

Parameters

- **field_path** (*list or None.*) – field path. It allows `*` as wildcard.
- **stop_first** (*bool*) – Stop iteration on first value looked up. Default: False.

Returns A list of values or None it was a invalid path.

Return type `list` or `None`

classmethod get_default_data()
Returns a dictionary with default data.

Returns dict

get_field_value(*name*)
Get the field value from the modified data or the original one

get_fields()
Returns used fields of model

get_original_field_value(*name*)
Returns original field value or None

classmethod get_structure()
Returns a dictionary with model field objects.

Returns dict

import_data(*data*)
Set the fields established in data to the instance

import_deleted_fields(*data*)
Set data fields to deleted

is_modified()
Returns whether model is modified or not

is_modified_field(*name*)
Returns whether a field is modified or not

reset_attr_by_path(*field_path*)
It restores original values for fields looked up by field path. Field path is dot-formatted string path: `parent_field.child_field`.

Parameters **field_path** (*str*) – field path. It allows * as wildcard.

reset_field_value(*name*)
Resets value of a field

set_field_value(*name*, *value*)
Set the value to the field modified_data

class dirty_models.models.**DynamicModel**(**args*, ***kwargs*)
Bases: dirty_models.models.BaseDynamicModel

DynamicModel allow to create model with no structure. Each instance has its own derivated class from DynamicModels.

class dirty_models.models.**FastDynamicModel**(**args*, ***kwargs*)
Bases: dirty_models.models.BaseDynamicModel

FastDynamicModel allow to create model with no structure.

get_current_structure()
Returns a dictionary with model field objects.

Returns dict

get_validated_object(*field_type*, *value*)
Returns the value validated by the field_type

class dirty_models.models.**HashMapModel**(**args*, ***kwargs*)
Bases: dirty_models.base.InnerFieldTypeMixin, *dirty_models.models.BaseModel*

Hash map model with dirty feature. It stores original data and saves modifications in other side.

copy()
Creates a copy of model

get_validated_object(*value*)
Returns the value validated by the field_type

3.2 Fields types

Fields to be used with dirty models.

class dirty_models.fields.**IntegerField**(*name=None*, *alias=None*, *getter=None*, *setter=None*, *read_only=False*, *default=None*, *doc=None*)
Bases: dirty_models.fields.BaseField

It allows to use an integer as value in a field.

Automatic cast from:

- float
- str if all characters are digits
- Enum if value of enum can be cast.

class dirty_models.fields.**FloatField**(*name=None*, *alias=None*, *getter=None*, *setter=None*, *read_only=False*, *default=None*, *doc=None*)

Bases: dirty_models.fields.BaseField

It allows to use a float as value in a field.

Automatic cast from:

- int
- str if all characters are digits and there is only one dot (.).
- Enum if value of enum can be cast.

class dirty_models.fields.**BooleanField**(*name=None*, *alias=None*, *getter=None*, *setter=None*, *read_only=False*, *default=None*, *doc=None*)

Bases: dirty_models.fields.BaseField

It allows to use a boolean as value in a field.

Automatic cast from:

- int 0 become False, anything else True
- str true and yes become True, anything else False. It is case-insensitive.
- Enum if value of enum can be cast.

class dirty_models.fields.**StringField**(*name=None*, *alias=None*, *getter=None*, *setter=None*, *read_only=False*, *default=None*, *doc=None*)

Bases: dirty_models.fields.BaseField

It allows to use a string as value in a field.

Automatic cast from:

- int
- float
- Enum if value of enum can be cast.

class dirty_models.fields.**StringIdField**(*name=None*, *alias=None*, *getter=None*, *setter=None*, *read_only=False*, *default=None*, *doc=None*)

Bases: *dirty_models.fields.StringField*

It allows to use a string as value in a field, but not allows empty strings. Empty string are like None and they will remove data of field.

Automatic cast from:

- int
- float
- Enum if value of enum can be cast.

set_value(*obj*, *value*)

Sets value to model if not empty

class dirty_models.fields.**TimeField**(*parse_format=None*, *default_timezone=None*, ***kwargs*)

Bases: dirty_models.fields.DateTimeBaseField

It allows to use a time as value in a field.

Automatic cast from:

- list items will be used to construct time object as arguments.

- `dict` items will be used to construct `time` object as keyword arguments.
- `str` will be parsed using a function or format in `parser` constructor parameter.
- `int` will be used as timestamp.
- `datetime` will get time part.
- `Enum` if value of enum can be cast.

class `dirty_models.fields.`**`DateField`**(*parse_format=None*, ***kwargs*)
Bases: `dirty_models.fields.DateTimeBaseField`

It allows to use a date as value in a field.

Automatic cast from:

- `list` items will be used to construct `date` object as arguments.
- `dict` items will be used to construct `date` object as keyword arguments.
- `str` will be parsed using a function or format in `parser` constructor parameter.
- `int` will be used as timestamp.
- `datetime` will get date part.
- `Enum` if value of enum can be cast.

class `dirty_models.fields.`**`DateTimeField`**(*parse_format=None*, *default_timezone=None*, *force_timezone=False*, ***kwargs*)
Bases: `dirty_models.fields.DateTimeBaseField`

It allows to use a datetime as value in a field.

Automatic cast from:

- `list` items will be used to construct `datetime` object as arguments.
- `dict` items will be used to construct `datetime` object as keyword arguments.
- `str` will be parsed using a function or format in `parser` constructor parameter.
- `int` will be used as timestamp.
- `date` will set date part.
- `Enum` if value of enum can be cast.

class `dirty_models.fields.`**`TimedeltaField`**(*name=None*, *alias=None*, *getter=None*, *setter=None*, *read_only=False*, *default=None*, *doc=None*)
Bases: `dirty_models.fields.BaseField`

It allows to use a timedelta as value in a field.

Automatic cast from:

- `float` as seconds.
- `int` as seconds.
- `Enum` if value of enum can be cast.

class `dirty_models.fields.`**`ModelField`**(*model_class=None*, ***kwargs*)
Bases: `dirty_models.fields.BaseField`

It allows to use a model as value in a field. Model type must be defined on constructor using param model_class. If it is not defined self model will be used. It means model inside field will be the same class than model who define field.

Automatic cast from:

- dict.
- collections.Mapping.

model_class
Model_class getter: model class used on field

class dirty_models.fields.**ArrayField**(*autolist=False*, ***kwargs*)
Bases: dirty_models.fields.InnerFieldTypeMixin, dirty_models.fields.BaseField

It allows to create a ListModel (iterable in dirty_models.types) of different elements according to the specified field_type. So it is possible to have a list of Integers, Strings, Models, etc. When using a model with no specified model_class the model inside field.

Automatic cast from:

- set.
- tuple.

autolist
autolist getter: autolist flag allows to convert a simple item on a list with one item.

class dirty_models.fields.**HashMapField**(*model_class=None*, ***kwargs*)
Bases: dirty_models.fields.InnerFieldTypeMixin, *dirty_models.fields.ModelField*

It allows to create a field which contains a hash map.

Automatic cast from:

- dict.
- BaseModel.

class dirty_models.fields.**BlobField**(*name=None*, *alias=None*, *getter=None*, *setter=None*, *read_only=False*, *default=None*, *doc=None*)
Bases: dirty_models.fields.BaseField

It allows any type of data.

class dirty_models.fields.**MultiTypeField**(*field_types=None*, ***kwargs*)
Bases: dirty_models.fields.BaseField

It allows to define multiple type for a field. So, it is possible to define a field as a integer and as a model field, for example.

class dirty_models.fields.**EnumField**(*enum_class*, **args*, ***kwargs*)
Bases: dirty_models.fields.BaseField

It allows to create a field which contains a member of an enumeration.

Automatic cast from:

- Any value of enumeration.
- Any member name of enumeration.

3.3 Inner model types

Internal types for dirty models

class dirty_models.model_types.**ListModel**(*seq=None*, **args*, ***kwargs*)
Bases: dirty_models.base.InnerFieldTypeMixin, *dirty_models.base.BaseData*

Dirty model for a list. It has the behavior to work as a list implementing its methods and has also the methods export_data, export_modified_data, import_data and flat_data to work also as a model, storing original and modified values.

append(*item*)
Appending elements to our list

clear()
Resets our list, keeping original data

clear_all()
Resets our list

clear_modified_data()
Clears only the modified data

count(*value*)
Gives the number of occurrencies of a value in the list

delete_attr_by_path(*field*)
Function for deleting a field specifying the path in the whole model as described in dirty:models.models.BaseModel.perform_function_by_path()

export_data()
Retrieves the data in a jsoned form

export_deleted_fields()
Returns a list with any deleted fields form original data. In tree models, deleted fields on children will be appended.

export_modifications()
Returns list modifications.

export_modified_data()
Retrieves the modified data in a jsoned form

export_original_data()
Retrieves the original_data

extend(*iterable*)
Given an iterable, it adds the elements to our list

flat_data()
Function to pass our modified values to the original ones

get_1st_attr_by_path(*field_path*, ***kwargs*)
It returns first value looked up by field path. Field path is dot-formatted string path: parent_field.child_field.

Parameters

- **field_path** (*str*) – field path. It allows * as wildcard.
- **default** – Default value if field does not exist. If it is not defined AttributeError exception will be raised.

Returns value

get_attrs_by_path(*field_path*, *stop_first=False*)
It returns list of values looked up by field path. Field path is dot-formatted string path: `parent_field.child_field`.

Parameters

- **field_path** (*list or None.*) – field path. It allows * as wildcard.
- **stop_first** (*bool*) – Stop iteration on first value looked up. Default: False.

Returns value

get_validated_object(*value*)
Returns the value validated by the field_type

import_data(*data*)
Uses data to add it to the list

import_deleted_fields(*data*)
Set data fields to deleted

index(*value*)
Gets the index in the list for a value

initialise_modified_data()
Initialise the modified_data if necessary

insert(*index*, *p_object*)
Insert an element to a list

is_modified()
Returns whether list is modified or not

pop(**args*)
Obtains and delete the element from the list

remove(*value*)
Deleting an element from the list

reset_attr_by_path(*field*)
Function for restoring a field specifying the path in the whole model as described in `dirty:models.models.BaseModel.perform_function_by_path()`

reverse()
Reverses the list order

sort()
Sorts the list

dirty_models.model_types.**modified_data_decorator**(*function*)
Decorator to initialise the modified_data if necessary. To be used in list functions to modify the list

3.4 Base classes

Base classes for Dirty Models

class dirty_models.base.**BaseData**(**args*, ***kwargs*)
Bases: object

Base class for data inside dirty model.

get_parent()
Returns parent model

get_read_only()
Returns whether model could be modified or not

is_locked()
Returns whether model is locked

lock()
Lock model to avoid modification on read only fields

set_parent(*value*)
Sets parent model

set_read_only(*value*)
Sets whether model could be modified or not

unlock()
Unlock model to be able to write even it's read only

class dirty_models.base.**Unlocker**(*item*)
Bases: object

Unlocker instances helps to lock and unlock models easily

3.5 Utilities

dirty_models.utils.**factory**
alias of *Factory*

class dirty_models.utils.**JSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)
Bases: json.encoder.JSONEncoder

Json encoder for Dirty Models

default_model_iter
alias of ModelFormatterIter

class dirty_models.utils.**Factory**(*func*)
Bases: object

Factory decorator could be used to define result of a function as default value. It could be useful to define a *DateTimeField* with datetime.datetime.now() in order to set the current datetime.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

T

U