django-mptt Documentation

Release 0.6.0

Craig de Stigter

April 16, 2014

Contents:

If you're new to django-mptt, you may want to start with these documents to get you up and running:

CHAPTER 1

Overview

Contents

1.1 What is Modified Preorder Tree Traversal?

MPTT is a technique for storing hierarchical data in a database. The aim is to make retrieval operations very efficient.

The trade-off for this efficiency is that performing inserts and moving items around the tree is more involved, as there's some extra work required to keep the tree structure in a good state at all times.

Here's a good article about MPTT to whet your appetite and provide details about how the technique itself works:

- Storing Hierarchical Data in a Database

1.2 What is `django-mptt`?

`django-mptt` is a reusable Django app which aims to make it easy for you to use MPTT with your own Django models.

It takes care of the details of managing a database table as a tree structure and provides tools for working with trees of model instances.

1.2.1 Feature overview

- Simple registration of models - fields required for tree structure will be added automatically.
- The tree structure is automatically updated when you create or delete model instances, or change an instance's parent.
- Each level of the tree is automatically sorted by a field (or fields) of your choice.
- New model methods are added to each registered model for:
  - changing position in the tree

  - retrieving ancestors, siblings, descendants
  - counting descendants
  - other tree-related operations
- A `TreeManager` manager is added to all registered models. This provides methods to:
  - move nodes around a tree, or into a different tree
  - insert a node anywhere in a tree
  - rebuild the MPTT fields for the tree (useful when you do bulk updates outside of django)
- Form fields for tree models.
- Utility functions for tree models.
- Template tags and filters for rendering trees.
- Translations for:
  - Danish
  - French
  - German
  - Polish

CHAPTER 2

Installation

Contents

2.1 Official releases

Official releases are available from PyPI.

Download the .zip distribution file and unpack it. Inside is a script named `setup.py`. Enter this command:

```
python setup.py install
```

...and the package will install automatically.

2.2 Development version

Alternatively, you can get the latest source from our git repository:

```
git clone git://github.com/django-mptt/django-mptt.git django-mptt
```

Add the resulting folder to your PYTHONPATH or symlink the `mptt` directory inside it into a directory which is on your PYTHONPATH, such as your Python installation's `site-packages` directory.

You can verify that the application is available on your PYTHONPATH by opening a Python interpreter and entering the following commands:

```
>>> import mptt
>>> mptt.VERSION
(0, 5, '+dev')
```

When you want to update your copy of the source code, run `git pull` from within the `django-mptt` directory.

Caution: The development version may contain bugs which are not present in the release version and introduce backwards-incompatible changes.
If you're tracking master, keep an eye on the recent Commit History before you update your copy of the source code.

CHAPTER 3

Tutorial

3.1 The Problem

You've created a Django project, and you need to manage some hierarchical data. For instance you've got a bunch of hierarchical pages in a CMS, and sometimes pages are *children* of other pages

Now suppose you want to show a breadcrumb on your site, like this:

```
Home > Products > Food > Meat > Spam > Spammy McDelicious
```

To get all those page titles you might do something like this:

```
titles = []
while page:
    titles.append(page.title)
    page = page.parent
```

That's one database query for each page in the breadcrumb, and database queries are slow. Let's do this a better way.

3.2 The Solution

Modified Preorder Tree Traversal can be a bit daunting at first, but it's one of the best ways to solve this problem.

If you want to go into the details, there's a good explanation here: Storing Hierarchical Data in a Database or Managing Hierarchical Data in Mysql

tl;dr: MPTT makes most tree operations much cheaper in terms of queries. In fact all these operations take at most one query,

- get descendants of a node
- get ancestors of a node
- get all nodes at a given level
- get leaf nodes

And this one takes zero queries:

- count the descendants of a given node

Enough intro. Let's get started.

3.3 Getting started

3.3.1 Add `mptt` To `INSTALLED_APPS`

As with most Django applications, you should add `mptt` to the `INSTALLED_APPS` in your `settings.py` file:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    # ...
    'mptt',
)
```

3.3.2 Set up your model

Start with a basic subclass of MPTTModel, something like this:

```
from django.db import models
from mptt.models import MPTTModel, TreeForeignKey

class Genre(MPTTModel):
    name = models.CharField(max_length=50, unique=True)
    parent = TreeForeignKey('self', null=True, blank=True, related_name='children', db_index=True)

    class MPTTMeta:
        order_insertion_by = ['name']
```

You must define a parent field which is a `TreeForeignKey` to `'self'`. A `TreeForeignKey` is just a regular `ForeignKey` that renders form fields differently in the admin and a few other places.

Because you're inheriting from MPTTModel, your model will also have a number of other fields: `level`, `lft`, `rght`, and `tree_id`. These fields are managed by the MPTT algorithm. Most of the time you won't need to use these fields directly.

That `MPTTMeta` class adds some tweaks to `django-mptt` - in this case, just `order_insertion_by`. This indicates the natural ordering of the data in the tree.

Now create your table in the database:

```
python manage.py syncdb
```

3.3.3 Create some data

Fire up a django shell:

```
python manage.py shell
```

Now create some data to test:

```
from myapp.models import Genre
rock = Genre.objects.create(name="Rock")
blues = Genre.objects.create(name="Blues")
Genre.objects.create(name="Hard Rock", parent=rock)
Genre.objects.create(name="Pop Rock", parent=rock)
```

3.3.4 Make a view

This one's pretty simple for now. Add this lightweight view to your `views.py`:

```
def show_genres(request):
    return render_to_response("genres.html",
                              {'nodes':Genre.objects.all()},
                              context_instance=RequestContext(request))
```

And add a URL for it in `urls.py`:

```
(r'^genres/$', 'myapp.views.show_genres'),
```

3.3.5 Template

`django-mptt` includes some template tags for making this bit easy too. Create a template called `genres.html` in your template directory and put this in it:

```
{% load mptt_tags %}
<ul>
    {% recursetree nodes %}
        <li>
            {{ node.name }}
            {% if not node.is_leaf_node %}
                <ul class="children">
                    {{ children }}
                </ul>
            {% endif %}
        </li>
    {% endrecursetree %}
</ul>
```

That recursetree tag will recursively render that template fragment for all the nodes. Try it out by going to `/genres/`.

There's more; check out the docs for custom admin-site stuff, more template tags, tree rebuild functions etc.

Now you can stop thinking about how to do trees, and start making a great django app!

Topic docs:

CHAPTER 4

Models and Managers

Contents

4.1 Setting up a Django model for MPTT

Start with a basic subclass of MPTTModel, something like this:

```
from django.db import models
from mptt.models import MPTTModel, TreeForeignKey

class Genre(MPTTModel):
    name = models.CharField(max_length=50, unique=True)
    parent = TreeForeignKey('self', null=True, blank=True, related_name='children', db_index=True)
```

You must define a parent field which is a `ForeignKey` to `'self'`. Recommended: use `TreeForeignKey`. You can call it something different if you want - see Model Options below.

Because you're inheriting from MPTTModel, your model will also have a number of other fields: `level`, `lft`, `rght`, and `tree_id`. Most of the time you won't need to use these fields directly, but it's helpful to know they're there.

Please note that if you are using multi-inheritance, MPTTModel should usually be the first class to be inherited from:

```
class Genre(MPTTModel,Foo,Bar):
    name = models.CharField(max_length=50, unique=True)
```

Since MPTTModel inherits from `models.Model`, this is very important when you have "diamond-style" multiple inheritance : you inherit from two Models that both inherit from the same base class (e.g. `models.Model`) . In that case, If MPTTModel is not the first Model, you may get errors at Model validation, like `AttributeError: 'NoneType' object has no attribute 'name'`.

4.2 Model Options

Sometimes you might want to change the names of the above fields, for instance if you've already got a field named `level` and you want to avoid conflicts.

To change the names, create an `MPTTMeta` class inside your class:

```
class Genre(MPTTModel):
    name = models.CharField(max_length=50, unique=True)
    parent = TreeForeignKey('self', null=True, blank=True, related_name='children', db_index=True)

    class MPTTMeta:
        level_attr = 'mptt_level'
        order_insertion_by=['name']
```

The available options for the MPTTMeta class are:

parent_attr The name of a field which relates the model back to itself such that each instance can be a child of another instance. Defaults to `'parent'`.

Users are responsible for setting this field up on the model class, which can be done like so:

```
parent = TreeForeignKey('self', null=True, blank=True, related_name='children', db_index=True)
```

For the following four arguments, if fields with the given names do not exist, they will be added to the model dynamically:

left_attr The name of a field which contains the left tree node edge indicator, which should be a `PositiveIntegerField`. Defaults to `'lft'`.

right_attr The name of a field which contains the right tree node edge indicator, which should be a `PositiveIntegerField`. Defaults to `'rght'`.

tree_id_attr The name of a field which contains the tree id of each node, which should be a `PositiveIntegerField`. Defaults to `'tree_id'`.

Items which do not have a parent are considered to be "root" nodes in the tree and will be allocated a new tree id. All descendants of root nodes will be given the same tree id as their root node.

level_attr The name of a field which contains the (zero-based) level at which an item sits in the tree, which should be a `PositiveIntegerField`. Defaults to `'level'`.

For example, root nodes would have a level of `0` and their immediate children would have have a level of `1`.

order_insertion_by A list of field names which should define ordering when new tree nodes are being inserted or existing nodes are being reparented, with the most significant ordering field name first. Defaults to `[]`.

It is assumed that any field identified as defining ordering will never be `NULL` in the database.

Note that this will require an extra database query to determine where nodes should be positioned when they are being saved. This option is handy if you're maintaining mostly static structures, such as trees of categories, which should always be in alphabetical order.

4.3 Registration of existing models

The preferred way to do model registration in `django-mptt` is by subclassing `MPTTModel`.

However, sometimes that doesn't quite work. For instance, suppose you want to modify Django's Group model to be hierarchical.

You can't subclass MPTTModel without modifying the Group source. Instead, you can do:

```
import mptt
from mptt.fields import TreeForeignKey
from django.contrib.auth.models import Group

# add a parent foreign key
TreeForeignKey(Group, blank=True, null=True, db_index=True).contribute_to_class(Group, 'parent')

mptt.register(Group, order_insertion_by=['name'])
```

4.4 MPTTModel instance methods

Subclasses of MPTTModel have the following instance methods:

4.4.1 get_ancestors(ascending=False, include_self=False)

creates a `QuerySet` containing the ancestors of the model instance.

These default to being in descending order (root ancestor first, immediate parent last); passing `True` for the `ascending` argument will reverse the ordering (immediate parent first, root ancestor last).

If `include_self` is `True`, the `QuerySet` will also include the model instance itself.

4.4.2 get_children()

Creates a `QuerySet` containing the immediate children of the model instance, in tree order.

The benefit of using this method over the reverse relation provided by the ORM to the instance's children is that a database query can be avoided in the case where the instance is a leaf node (it has no children).

4.4.3 get_descendants(include_self=False)

Creates a `QuerySet` containing descendants of the model instance, in tree order.

If `include_self` is `True`, the `QuerySet` will also include the model instance itself.

4.4.4 get_descendant_count()

Returns the number of descendants the model instance has, based on its left and right tree node edge indicators. As such, this does not incur any database access.

4.4.5 get_family()

Returns a `QuerySet` containing the ancestors, the model itself and the descendants, in tree order.

4.4.6 get_next_sibling()

Returns the model instance's next sibling in the tree, or `None` if it doesn't have a next sibling.

4.4.7 get_previous_sibling()

Returns the model instance's previous sibling in the tree, or `None` if it doesn't have a previous sibling.

4.4.8 get_root()

Returns the root node of the model instance's tree.

4.4.9 get_siblings(include_self=False)

Creates a `QuerySet` containing siblings of the model instance. Root nodes are considered to be siblings of other root nodes.

If `include_self` is `True`, the `QuerySet` will also include the model instance itself.

4.4.10 insert_at(target, position='first-child', save=False)

Positions the model instance (which must not yet have been inserted into the database) in the tree based on `target` and `position` (when appropriate).

If `save` is True, the model instance's `save()` method will also be called.

4.4.11 is_child_node()

Returns `True` if the model instance is a child node, `False` otherwise.

4.4.12 is_leaf_node()

Returns `True` if the model instance is a leaf node (it has no children), `False` otherwise.

4.4.13 is_root_node()

Returns `True` if the model instance is a root node, `False` otherwise.

4.4.14 move_to(target, position='first-child')

Moves the model instance elsewhere in the tree based on `target` and `position` (when appropriate).

Note: It is assumed that when you call this method, the tree fields in the instance you've called it on, and in any `target` instance passed in, reflect the current state of the database.

Modifying the tree fields manually before calling this method or using tree fields which are out of sync with the database can result in the tree structure being put into an inaccurate state.

If `target` is another model instance, it will be used to determine the type of movement which needs to take place, and will be used as the basis for positioning the model when it is moved, in combination with the `position` argument.

A `target` of `None` indicates that the model instance should be turned into a root node. The `position` argument is disregarded in this case.

Valid values for the `position` argument and their effects on movement are:

`'first-child'` The instance being moved should have `target` set as its new parent and be placed as its *first* child in the tree structure.

`'last-child'` The instance being moved should have `target` set as its new parent and be placed as its *last* child in the tree structure.

`'left'` The instance being moved should have `target`'s parent set as its new parent and should be placed *directly before* `target` in the tree structure.

`'right'` The instance being moved should have `target`'s parent set as its new parent and should be placed *directly after* `target` in the tree structure.

A `ValueError` will be raised if an invalid value is given for the `position` argument.

Note that some of the moves you could attempt to make with this method are invalid - for example, trying to make an instance be its own child or the child of one of its descendants. In these cases, a `mptt.exceptions.InvalidMove` exception will be raised.

The instance itself will be also modified as a result of this call, to reflect the state of its updated tree fields in the database, so it's safe to go on to save it or use its tree fields after you've called this method.

4.5 TreeForeignKey, TreeOneToOneField, TreeManyToManyField

New in version 0.5.

It's recommended you use `mptt.fields.TreeForeignKey` wherever you have a foreign key to an MPTT model. This includes the `parent` link you've just created on your model.

`TreeForeignKey` is just like a regular `ForeignKey` but it makes the default form field display choices in tree form.

There are also `TreeOneToOneField` and `TreeManyToManyField` if you need them.

4.6 The TreeManager custom manager

The default manager for an MPTTModel is a `TreeManager`.

Any `QuerySet` created with this manager will be ordered based on the tree structure, with root nodes appearing in tree id order and and their descendants being ordered in a depth-first fashion.

4.6.1 Methods

The following manager methods are available:

`disable_mptt_updates()` and `delay_mptt_updates()`

These two methods return context managers, and are both for doing efficient bulk updates of large trees. See the autogenerated docs for more information:

- delay_mptt_updates
- disable_mptt_updates

`rebuild()`

Rebuilds the mptt fields for the entire table. This can be handy:

- if your tree gets corrupted somehow.
- After large bulk operations, when you've used `disable_mptt_updates`

`add_related_count(queryset, rel_cls, rel_field, count_attr, cumulative=False)`

Adds a related item count to a given `QuerySet` using its extra method, for a model which has a relation to this manager's model.

`rel_cls` A Django model class which has a relation to this manager's model.

`rel_field` The name of the field in `rel_cls` which holds the relation.

`count_attr` The name of an attribute which should be added to each item in this `QuerySet`, containing a count of how many instances of `rel_cls` are related to it through `rel_field`.

`cumulative` If `True`, the count will be for each item and all of its descendants, otherwise it will be for each item itself.

`root_node(tree_id)`

Returns the root node of tree with the given id.

`insert_node(node, target, position='last-child', save=False)`

Sets up the tree state for `node` (which has not yet been inserted into in the database) so it will be positioned relative to a given `target` node as specified by `position` (when appropriate) when it is inserted, with any neccessary space already having been made for it.

A `target` of `None` indicates that `node` should be the last root node.

If `save` is `True`, `node`'s `save()` method will be called before it is returned.

move_node(node, target, position='last-child')

Moves `node` based on `target`, relative to `position` when appropriate.

A `target` of `None` indicates that `node` should be removed from its current position and turned into a root node. If `node` is a root node in this case, no action will be taken.

The given `node` will be modified to reflect its new tree state in the database.

For more details, see the move_to documentation above.

root_nodes()

Creates a `QuerySet` containing root nodes.

4.6.2 Example usage

In the following examples, we have `Category` and `Question` models. `Question` has a `category` field which is a `TreeForeignKey` to `Category`.

Retrieving a list of root Categories which have a `question_count` attribute containing the number of Questions associated with each root and all of its descendants:

```
roots = Category.objects.add_related_count(Category.tree.root_nodes(), Question,
                                           'category', 'question_counts',
                                           cumulative=True)
```

Retrieving a list of child Categories which have a `question_count` attribute containing the number of Questions associated with each of them:

```
node = Category.objects.get(name='Some Category')
children = Category.objects.add_related_count(node.get_children(), Question,
                                              'category', 'question_counts')
```

CHAPTER 5

Admin classes

5.1 mptt.admin.MPTTModelAdmin

This is a bare-bones tree admin. All it does is enforce ordering, and indent the nodes in the tree to make a pretty tree list view.

Usage:

```
from django.contrib import admin
from mptt.admin import MPTTModelAdmin
from myproject.myapp.models import Node

admin.site.register(Node, MPTTModelAdmin)
```

You can change the indent pixels per level globally by putting this in your settings.py:

```
# default is 10 pixels
MPTT_ADMIN_LEVEL_INDENT = 20
```

If you'd like to specify the pixel amount per Model, define an `mptt_level_indent` attribute in your MPTTModelAdmin:

```
from django.contrib import admin
from mptt.admin import MPTTModelAdmin
from myproject.myapp.models import Node

class CustomMPTTModelAdmin(MPTTModelAdmin)
    # speficfy pixel amount for this ModelAdmin only:
    mptt_level_indent = 20

admin.site.register(Node, CustomMPTTModelAdmin)
```

If you'd like to specify which field should be indented, add an `mptt_indent_field` to your MPTTModelAdmin:

```
# ...
class CustomMPTTModelAdmin(MPTTModelAdmin)
    mptt_indent_field = "some_node_field"
# ...
```

CHAPTER 6

Working with trees in Django forms

Contents

6.1 Fields

The following custom form fields are provided in the `mptt.forms` package.

6.1.1 `TreeNodeChoiceField`

This is the default formfield used by `TreeForeignKey`

A subclass of ModelChoiceField which represents the tree level of each node when generating option labels.

For example, where a form which used a `ModelChoiceField`:

```
category = ModelChoiceField(queryset=Category.tree.all())
```

...would result in a select with the following options:

```
---------
Root 1
Child 1.1
Child 1.1.1
Root 2
Child 2.1
Child 2.1.1
```

Using a `TreeNodeChoiceField` instead:

```
category = TreeNodeChoiceField(queryset=Category.tree.all())
```

...would result in a select with the following options:

```
Root 1
--- Child 1.1
------ Child 1.1.1
Root 2
--- Child 2.1
------ Child 2.1.1
```

The text used to indicate a tree level can by customised by providing a `level_indicator` argument:

```
category = TreeNodeChoiceField(queryset=Category.tree.all(),
                               level_indicator=u'+--')
```

...which for this example would result in a select with the following options:

```
Root 1
+-- Child 1.1
+--+-- Child 1.1.1
Root 2
+-- Child 2.1
+--+-- Child 2.1.1
```

6.1.2 TreeNodeMultipleChoiceField

Just like `TreeNodeChoiceField`, but accepts more than one value.

6.1.3 TreeNodePositionField

A subclass of ChoiceField whose choices default to the valid arguments for the move_to method.

6.2 Forms

The following custom form is provided in the `mptt.forms` package.

6.2.1 MoveNodeForm

A form which allows the user to move a given node from one location in its tree to another, with optional restriction of the nodes which are valid target nodes for the *move_to method*

Fields

The form contains the following fields:

- `target` – a `TreeNodeChoiceField` for selecting the target node for the node movement.

  Target nodes will be displayed as a single `<select>` with its `size` attribute set, so the user can scroll through the target nodes without having to open the dropdown list first.

- `position` – a `TreeNodePositionField` for selecting the position of the node movement, related to the target node.

Construction

Required arguments:

node When constructing the form, the model instance representing the node to be moved must be passed as the first argument.

Optional arguments:

valid_targets If provided, this keyword argument will define the list of nodes which are valid for selection in form. Otherwise, any instance of the same model class as the node being moved will be available for selection, apart from the node itself and any of its descendants.

For example, if you want to restrict the node to moving within its own tree, pass a `QuerySet` containing everything in the node's tree except itself and its descendants (to prevent invalid moves) and the root node (as a user could choose to make the node a sibling of the root node).

target_select_size If provided, this keyword argument will be used to set the size of the select used for the target node. Defaults to `10`.

position_choices A tuple of allowed position choices and their descriptions.

level_indicator A string which will be used to represent a single tree level in the target options.

`save()` method

When the form's `save()` method is called, it will attempt to perform the node movement as specified in the form.

If an invalid move is attempted, an error message will be added to the form's non-field errors (accessible using `{{ form.non_field_errors }}` in templates) and the associated `mptt.exceptions.InvalidMove` will be re-raised.

It's recommended that you attempt to catch this error and, if caught, allow your view to to fall through to rendering the form again again, so the error message is displayed to the user.

Example usage

A sample view which shows basic usage of the form is provided below:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response

from faqs.models import Category
from mptt.exceptions import InvalidMove
from mptt.forms import MoveNodeForm

def move_category(request, category_pk):
    category = get_object_or_404(Category, pk=category_pk)
    if request.method == 'POST':
        form = MoveNodeForm(category, request.POST)
        if form.is_valid():
            try:
                category = form.save()
                return HttpResponseRedirect(category.get_absolute_url())
            except InvalidMove:
                pass
    else:
        form = MoveNodeForm(category)
```

```
    return render_to_response('faqs/move_category.html', {
        'form': form,
        'category': category,
        'category_tree': Category.tree.all(),
    })
```

CHAPTER 7

Working with trees in templates

Contents

7.1 Getting started

Before you can use these tags/filters, you must:

- add "mptt" to your `INSTALLED_APPS` in `settings.py`
- add `{% load mptt_tags %}` in your template.

7.2 Recursive tags

New in version 0.4.

For most setups, recursive tags are the easiest and most efficient way to render trees.

7.2.1 `recursetree`

New in version 0.4.

This tag renders a section of your template recursively for each node in your tree.

For example:

```
<ul class="root">
    {% recursetree nodes %}
        <li>
            {{ node.name }}
            {% if not node.is_leaf_node %}
                <ul class="children">
                    {{ children }}
                </ul>
            {% endif %}
        </li>
    {% endrecursetree %}
</ul>
```

Note the special variables `node` and `children`. These are magically inserted into your context while you're inside the `recursetree` tag.

`node` is an instance of your MPTT model.

`children` : This variable holds the rendered HTML for the children of `node`.

Note: If you already have variables called `node` or `children` in your template, and you need to access them inside the `recursetree` block, you'll need to alias them to some other name first:

```
{% with node as friendly_node %}
    {% recursetree nodes %}
        {{ node.name }} is friends with {{ friendly_node.name }}
        {{ children }}
    {% endrecursetree %}
{% endwith %}
```

7.3 Iterative tags

Why? These tags are better suited to unusually deep trees. If you expect to have trees with depth > 20, you should use these instead of the above.

7.3.1 `full_tree_for_model`

Populates a template variable with a `QuerySet` containing the full tree for a given model.

Usage:

```
{% full_tree_for_model [model] as [varname] %}
```

The model is specified in `[appname].[modelname]` format.

Example:

```
{% full_tree_for_model tests.Genre as genres %}
```

7.3.2 `drilldown_tree_for_node`

Populates a template variable with the drilldown tree for a given node, optionally counting the number of items associated with its children.

A drilldown tree consists of a node's ancestors, itself and its immediate children. For example, a drilldown tree for a book category "Personal Finance" might look something like:

```
Books
   Business, Finance & Law
      Personal Finance
         Budgeting (220)
         Financial Planning (670)
```

Usage:

```
{% drilldown_tree_for_node [node] as [varname] %}
```

Extended usage:

```
{% drilldown_tree_for_node [node] as [varname] count [foreign_key] in [count_attr] %}
{% drilldown_tree_for_node [node] as [varname] cumulative count [foreign_key] in [count_attr] %}
```

The foreign key is specified in `[appname].[modelname].[fieldname]` format, where `fieldname` is the name of a field in the specified model which relates it to the given node's model.

When this form is used, a `count_attr` attribute on each child of the given node in the drilldown tree will contain a count of the number of items associated with it through the given foreign key.

If cumulative is also specified, this count will be for items related to the child node and all of its descendants.

Examples:

```
{% drilldown_tree_for_node genre as drilldown %}
{% drilldown_tree_for_node genre as drilldown count tests.Game.genre in game_count %}
{% drilldown_tree_for_node genre as drilldown cumulative count tests.Game.genre in game_count %}
```

See Examples for an example of how to render a drilldown tree as a nested list.

7.4 Filters

7.4.1 `tree_info` filter

Given a list of tree items, iterates over the list, generating two-tuples of the current tree item and a `dict` containing information about the tree structure around the item, with the following keys:

`'new_level'` `True` if the current item is the start of a new level in the tree, `False` otherwise.

`'closed_levels'` A list of levels which end after the current item. This will be an empty list if the next item's level is the same as or greater than the level of the current item.

An optional argument can be provided to specify extra details about the structure which should appear in the `dict`. This should be a comma-separated list of feature names. The valid feature names are:

ancestors Adds a list of unicode representations of the ancestors of the current node, in descending order (root node first, immediate parent last), under the key `'ancestors'`.

For example: given the sample tree below, the contents of the list which would be available under the `'ancestors'` key are given on the right:

```
Books                   ->  []
   Sci-fi               ->  [u'Books']
      Dystopian Futures ->  [u'Books', u'Sci-fi']
```

Using this filter with unpacking in a `{% for %}` tag, you should have enough information about the tree structure to create a hierarchical representation of the tree.

Example:

```
{% for genre,structure in genres|tree_info %}
    {% if structure.new_level %}<ul><li>{% else %}</li><li>{% endif %}
        {{ genre.name }}
    {% for level in structure.closed_levels %}</li></ul>{% endfor %}
{% endfor %}
```

7.4.2 tree_path

Creates a tree path represented by a list of items by joining the items with a separator, which can be provided as an optional argument, defaulting to `' :: '`.

Each path item will be coerced to unicode, so a list of model instances may be given if required.

Example:

```
{{ some_list|tree_path }}
{{ some_node.get_ancestors|tree_path:" > " }}
```

7.5 Examples

Using `drilldown_tree_for_node` and `tree_info` together to render a drilldown menu for a node, with cumulative counts of related items for the node's children:

```
{% drilldown_tree_for_node genre as drilldown cumulative count tests.Game.genre in game_count %}
{% for node,structure in drilldown|tree_info %}
    {% if structure.new_level %}<ul><li>{% else %}</li><li>{% endif %}
    {% ifequal node genre %}
        <strong>{{ node.name }}</strong>
    {% else %}
        <a href="{{ node.get_absolute_url }}">{{ node.name }}</a>
        {% ifequal node.parent_id genre.pk %}({{ node.game_count }}){% endifequal %}
    {% endifequal %}
    {% for level in structure.closed_levels %}</li></ul>{% endfor %}
{% endfor %}
```

Using `tree_info` (with its optional argument) and `tree_path` together to create a multiple-select, which:

- doesn't contain root nodes
- displays the full path to each node

```
<select name="classifiers" multiple="multiple" size="10">
    {% for node,structure in classifiers|tree_info:"ancestors" %}
        {% if node.is_child_node %}
            <option value="{{ node.pk }}">
                {{ structure.ancestors|tree_path }} :: {{ node }}
            </option>
        {% endif %}
    {% endfor %}
</select>
```

CHAPTER 8

Utilities for working with trees

Contents

8.1 List/tree utilities

The mptt.utils module contains the following functions for working with and creating lists of model instances which represent trees.

8.1.1 previous_current_next()

From http://www.wordaligned.org/articles/zippy-triples-served-with-python

Creates an iterator which returns (previous, current, next) triples, with None filling in when there is no previous or next available.

This function is useful if you want to step through a tree one item at a time and you need to refer to the previous or next item in the tree. It is used in the implementation of tree_item_iterator().

Required arguments

items A list or other iterable item.

8.1.2 tree_item_iterator()

This function is used to implement the tree_info template filter, yielding two-tuples of (tree item, tree structure information dict).

See the tree_info documentation for more information.

Required arguments

items A list or iterable of model instances which represent a tree.

Optional arguments

ancestors Boolean. If `True`, a list of unicode representations of the ancestors of the current node, in descending order (root node first, immediate parent last), will be added to the tree structure information `dict` under the key ``'ancestors'`.

8.1.3 drilldown_tree_for_node()

This function is used in the implementation of the `drilldown_tree_for_node` template tag.

It creates an iterable which yields model instances representing a drilldown tree for a given node.

A drilldown tree consists of a node's ancestors, itself and its immediate children, all in tree order.

Optional arguments may be given to specify details of a relationship between the given node's class and another model class, for the purpose of adding related item counts to the node's children.

Required arguments

node A model instance which represents a node in a tree.

Optional arguments

rel_cls A model class which has a relationship to the node's class.

rel_field The name of the field in `rel_cls` which holds the relationship to the node's class.

count_attr The name of an attribute which should be added to each child of the node in the drilldown tree (if any), containing a count of how many instances of `rel_cls` are related to it through `rel_field`.

cumulative If `True`, the count will be for items related to the child node *and* all of its descendants. Defaults to `False`.

CHAPTER 9

Upgrade notes

9.1 0.6.0

9.1.1 mptt now requires Python 2.6+, and supports Python 3.2+

mptt 0.6 drops support for both Python 2.4 and 2.5.

This was done to make it easier to support Python 3, as well as support the new context managers (delay_mptt_updates and disable_mptt_updates).

If you absolutely can't upgrade your Python version, you'll need to stick to mptt 0.5.5 until you can.

9.1.2 No more implicit `empty_label=True` on form fields

Until 0.5, `TreeNodeChoiceField` and `TreeNodeMultipleChoiceField` implicitly set `empty_label=True`. This was around since a long time ago, for unknown reasons. It has been removed in 0.6.0 as it caused occasional headaches for users.

If you were relying on this behavior, you'll need to explicitly pass `empty_label=True` to any of those fields you use, otherwise you will start seeing new '———' choices appearing in them.

9.1.3 Deprecated FeinCMSModelAdmin

If you were using `mptt.admin.FeinCMSModelAdmin`, you should switch to using `feincms.admin.tree_editor.TreeEditor` instead, or you'll get a loud deprecation warning.

9.2 0.4.2 to 0.5.5

9.2.1 `TreeManager` is now the default manager, `YourModel.tree` removed

In 0.5, `TreeManager` now behaves just like a normal django manager. If you don't override anything, you'll now get a `TreeManager` by default (`.objects`.)

Before 0.5, `.tree` was the default name for the `TreeManager`. That's been removed, so we recommend updating your code to use `.objects`.

If you don't want to update `.tree` to `.objects` everywhere just yet, you should add an explicit `TreeManager` to your models:

```
objects = tree = TreeManager()
```

9.2.2 `save(raw=True)` keyword argument removed

In earlier versions, MPTTModel.save() had a `raw` keyword argument. If True, the MPTT fields would not be updated during the save. This (undocumented) argument has now been removed.

9.2.3 `_meta` attributes moved to `_mptt_meta`

In 0.4, we deprecated all these attributes on model._meta. These have now been removed:

```
MyModel._meta.left_attr
MyModel._meta.right_attr
MyModel._meta.tree_id_attr
MyModel._meta.level_attr
MyModel._meta.tree_manager_attr
MyModel._meta.parent_attr
MyModel._meta.order_insertion_by
```

If you're still using any of these, you'll need to update by simply renaming `_meta` to `_mptt_meta`.

9.2.4 Running the tests

Tests are now run with:

```
cd tests/
./runtests.sh
```

The previous method (`python setup.py test`) no longer works since we switched to plain distutils.

9.3 0.3 to 0.4.2

9.3.1 Model changes

MPTT attributes on `MyModel._meta` deprecated, moved to `MyModel._mptt_meta`

Most people won't need to worry about this, but if you're using any of the following, note that these are deprecated and will be removed in 0.5:

```
MyModel._meta.left_attr
MyModel._meta.right_attr
MyModel._meta.tree_id_attr
MyModel._meta.level_attr
MyModel._meta.tree_manager_attr
MyModel._meta.parent_attr
MyModel._meta.order_insertion_by
```

They'll continue to work as previously for now, but you should upgrade your code if you can. Simply replace `_meta` with `_mptt_meta`.

Use model inheritance where possible

The preferred way to do model registration in `django-mptt` 0.4 is via model inheritance.

Suppose you start with this:

```
class Node(models.Model):
    ...

mptt.register(Node, order_insertion_by=['name'], parent_attr='padre')
```

First, Make your model a subclass of `MPTTModel`, instead of `models.Model`:

```
from mptt.models import MPTTModel

class Node(MPTTModel):
    ...
```

Then remove your call to `mptt.register()`. If you were passing it keyword arguments, you should add them to an `MPTTMeta` inner class on the model:

```
class Node(MPTTModel):
    ...
    class MPTTMeta:
        order_insertion_by = ['name']
        parent_attr = 'padre'
```

If necessary you can still use `mptt.register`. It was removed in 0.4.0 but restored in 0.4.2, since people reported use cases that didn't work without it.)

For instance, if you need to register models where the code isn't under your control, you'll need to use `mptt.register()`.

Behind the scenes, `mptt.register()` in 0.4 will actually add MPTTModel to `Node.__bases__`, thus achieving the same result as subclassing `MPTTModel`. If you're already inheriting from something other than `Model`, that means multiple inheritance.

You're probably all upgraded at this point :) A couple more notes for more complex scenarios:

9.3.2 More complicated scenarios

What if I'm already inheriting from something?

If your model is already a subclass of an abstract model, you should use multiple inheritance:

```
class Node(MPTTModel, ParentModel):
    ...
```

You should always put MPTTModel as the first model base. This is because there's some complicated metaclass stuff going on behind the scenes, and if Django's model metaclass gets called before the MPTT one, strange things can happen.

Isn't multiple inheritance evil? Well, maybe. However, the Django model docs don't forbid this, and as long as your other model doesn't have conflicting methods, it should be fine.

Note: As always when dealing with multiple inheritance, approach with a bit of caution.

Our brief testing says it works, but if you find that the Django internals are somehow breaking this approach for you, please create an issue with specifics.

Compatibility with 0.3

`MPTTModel` was added in 0.4. If you're writing a library or reusable app that needs to work with 0.3, you should use the `mptt.register()` function instead, as above.

CHAPTER 10

Technical details

Contents

10.1 Tree structure

In addition to the left and right identifiers which form the core of how MPTT works and the parent field, Django MPTT has the following additional fields on each tree node.

10.1.1 Tree id

A unique identifier assigned to each root node and inherited by all its descendants.

This identifier is the means by which Django MPTT implements multiple root nodes.

Since we can use it to identify all nodes which belong to a particular tree, the subtree headed by each root node can have its edge indicators starting at `1` - as long as tree operations are constrained to the appropriate tree id(s), we don't have to worry about overlapping of left and right edge indicator values with those of nodes in other trees.

This approach limits the number of rows affected when making space to insert new nodes and removing gaps left by deleted nodes.

Root node ordering

This field also defines the order in which root nodes are retrieved when you create a `QuerySet` using `TreeManager`, which by default orders the resulting `QuerySet` by tree id, then left edge indicator (for depth-first ordering).

When a new root node is created, it is assigned the next-largest tree id available, so by default root nodes (and thus their subtrees) are displayed in the order they were created.

Movement and volatility

Since root node ordering is defined by tree id, it can also be used to implement movement of other nodes as siblings of a target root node.

When you use the node movement API to move a node to be a sibling of a root node, tree ids are shuffled around to achieve the new ordering required. Given this, you should consider the tree id to be **volatile**, so it's recommended that you don't store tree ids in your applications to identify particular trees.

Since *every* node has a tree id, movement of a node to be a sibling of a root node can potentially result in a large number of rows being modified, as the further away the node you're moving is from its target location, the larger the number of rows affected - every node with a tree id between that of the node being moved and the target root node will require its tree id to be modified.

10.1.2 Level

The level (or "depth") at which a node sits in the tree.

Root nodes are level `0`, their immediate children are level `1`, *their* immediate children are level `2` and so on...

This field is purely denormalisation for convenience. It avoids the need to examine the tree structure to determine the level of a particular node and makes queries which need to take depth into account easier to implement using Django's ORM. For example, limiting the number of levels of nodes which are retrieved for the whole tree or any subtree:

```
# Retrieve root nodes and their immediate children only
SomeModel.tree.filter(level__lte=1)

# Retrieve descendants of a node up to two levels below it
node.get_descendants().filter(level__lte=node.level + 2)
```

10.2 Concurrency

Most CRUD methods involve the execution of multiple queries. These methods need to be made mutually exclusive, otherwise corrupt hierarchies might get created. Mutual exclusivity is usually achieved by placing the queries between **LOCK TABLE** and **UNLOCK TABLE** statements. However, mptt can't do the locking itself because the LOCK/UNLOCK statements are not transaction safe and would therefore mess up the client code's transactions. This means that it's the client code's responsibility to ensure that calls to mptt CRUD methods are mutually exclusive.

Note: In the above paragraph 'client code' means any django app that uses mptt base models.

10.3 Running the test suite

The `mptt.tests` package is set up as a project which holds a test settings module and defines models for use in testing MPTT. You can run the tests from the command-line using the `manage.py` script:

```
python manage.py test
```

CHAPTER 11

Autogenerated documentation

These docs are generated by Sphinx from the docstrings in `django-mptt`.

They're not necessarily very helpful. You might be just as well off reading the source code.

11.1 mptt

11.1.1 mptt.admin

class mptt.admin.**MPTTChangeList**(*request*, *model*, *list_display*, *list_display_links*, *list_filter*, *date_hierarchy*, *search_fields*, *list_select_related*, *list_per_page*, *list_max_show_all*, *list_editable*, *model_admin*)

get_query_set(**args*, ***kwargs*)

get_queryset(*request*)

class mptt.admin.**MPTTModelAdmin**(*model*, *admin_site*)

A basic admin class that displays tree items according to their position in the tree. No extra editing functionality beyond what Django admin normally offers.

change_list_template = u'admin/mptt_change_list.html'

form

alias of `MPTTAdminForm`

formfield_for_foreignkey(*db_field*, *request*, ***kwargs*)

get_changelist(*request*, ***kwargs*)

Returns the ChangeList class for use on the changelist page.

media

class mptt.admin.**MPTTAdminForm**(**args*, ***kwargs*)

A form which validates that the chosen parent for a node isn't one of its descendants.

base_fields = {}

clean()

declared_fields = {}

media

11.1.2 mptt.exceptions

MPTT exceptions.

exception mptt.exceptions.**CantDisableUpdates**
User tried to disable updates on a model that doesn't support it (abstract, proxy or a multiple-inheritance subclass of an MPTTModel)

exception mptt.exceptions.**InvalidMove**
An invalid node move was attempted.

For example, attempting to make a node a child of itself.

11.1.3 mptt.fields

Model fields for working with trees.

class mptt.fields.**TreeForeignKey**(*to*, *to_field=None*, *rel_class=<class 'django.db.models.fields.related.ManyToOneRel'>*, *db_constraint=True*, ***kwargs*)
Extends the foreign key, but uses mptt's TreeNodeChoiceField as the default form field.

This is useful if you are creating models that need automatically generated ModelForms to use the correct widgets.

formfield(***kwargs*)
Use MPTT's TreeNodeChoiceField

class mptt.fields.**TreeOneToOneField**(*to*, *to_field=None*, ***kwargs*)

formfield(***kwargs*)

class mptt.fields.**TreeManyToManyField**(*to*, *db_constraint=True*, ***kwargs*)

formfield(***kwargs*)

11.1.4 mptt.forms

Form components for working with trees.

class mptt.forms.**TreeNodeChoiceField**(*queryset*, **args*, ***kwargs*)
A ModelChoiceField for tree nodes.

class mptt.forms.**TreeNodeMultipleChoiceField**(*queryset*, **args*, ***kwargs*)
A ModelMultipleChoiceField for tree nodes.

class mptt.forms.**TreeNodePositionField**(**args*, ***kwargs*)
A ChoiceField for specifying position relative to another node.

DEFAULT_CHOICES = ((u'first-child', <django.utils.functional.__proxy__ object at 0x4026a10>), (u'last-child', <django.u

FIRST_CHILD = u'first-child'

LAST_CHILD = u'last-child'

LEFT = u'left'

RIGHT = u'right'

class mptt.forms.**MoveNodeForm**(*node*, **args*, ***kwargs*)

A form which allows the user to move a given node from one location in its tree to another, with optional restriction of the nodes which are valid target nodes for the move.

base_fields = {'target': <mptt.forms.TreeNodeChoiceField object at 0x4026c10>, 'position': <mptt.forms.TreeNodePo

media

save()

Attempts to move the node using the selected target and position.

If an invalid move is attempted, the related error message will be added to the form's non-field errors and the error will be re-raised. Callers should attempt to catch `InvalidNode` to redisplay the form with the error, should it occur.

11.1.5 mptt.managers

A custom manager for working with trees of objects.

class mptt.managers.**TreeManager**

A manager for working with trees of objects.

add_related_count(*queryset*, *rel_model*, *rel_field*, *count_attr*, *cumulative=False*)

Adds a related item count to a given `QuerySet` using its `extra` method, for a `Model` class which has a relation to this `Manager`'s `Model` class.

Arguments:

rel_model A `Model` class which has a relation to this *Manager*''s `Model` class.

rel_field The name of the field in `rel_model` which holds the relation.

count_attr The name of an attribute which should be added to each item in this `QuerySet`, containing a count of how many instances of `rel_model` are related to it through `rel_field`.

cumulative If `True`, the count will be for each item and all of its descendants, otherwise it will be for each item itself.

delay_mptt_updates(**args*, ***kwds*)

Context manager. Delays mptt updates until the end of a block of bulk processing.

NOTE that this context manager causes inconsistencies! MPTT model methods are not guaranteed to return the correct results until the end of the context block.

When to use this method: If used correctly, this method can be used to speed up bulk updates. This is best for updates in a localised area of the db table, especially if all the updates happen in a single tree and the rest of the forest is left untouched. No subsequent rebuild is necessary.

delay_mptt_updates does a partial rebuild of the modified trees (not the whole table). If used indiscriminately, this can actually be much slower than just letting the updates occur when they're required.

The worst case occurs when every tree in the table is modified just once. That results in a full rebuild of the table, which can be *very* slow.

If your updates will modify most of the trees in the table (not a small number of trees), you should consider using TreeManager.disable_mptt_updates, as it does much fewer queries.

Transactions: This doesn't enforce any transactional behavior. You should wrap this in a transaction to ensure database consistency.

Exceptions: If an exception occurs before the processing of the block, delayed updates will not be applied.

Usage:

```
with transaction.atomic():
    with MyNode.objects.delay_mptt_updates():
        ## bulk updates.
```

disable_mptt_updates(**args*, ***kwds*)
Context manager. Disables mptt updates.

NOTE that this context manager causes inconsistencies! MPTT model methods are not guaranteed to return the correct results.

When to use this method: If used correctly, this method can be used to speed up bulk updates.

This doesn't do anything clever. It *will* mess up your tree. You should follow this method with a call to TreeManager.rebuild() to ensure your tree stays sane, and you should wrap both calls in a transaction.

This is best for updates that span a large part of the table. If you are doing localised changes (1 tree, or a few trees) consider using delay_mptt_updates. If you are making only minor changes to your tree, just let the updates happen.

Transactions: This doesn't enforce any transactional behavior. You should wrap this in a transaction to ensure database consistency.

If updates are already disabled on the model, this is a noop.

Usage:

```
with transaction.atomic():
    with MyNode.objects.disable_mptt_updates():
        ## bulk updates.
    MyNode.objects.rebuild()
```

get_query_set()
Returns a `QuerySet` which contains all tree items, ordered in such a way that that root nodes appear in tree id order and their subtrees appear in depth-first order.

get_queryset()
Returns a `QuerySet` which contains all tree items, ordered in such a way that that root nodes appear in tree id order and their subtrees appear in depth-first order.

get_queryset_ancestors(*queryset*, *include_self=False*)
Returns a queryset containing the ancestors of all nodes in the given queryset.

If `include_self=True`, nodes in `queryset` will also be included in the result.

get_queryset_descendants(*queryset*, *include_self=False*)
Returns a queryset containing the descendants of all nodes in the given queryset.

If `include_self=True`, nodes in `queryset` will also be included in the result.

init_from_model(*model*)
Sets things up. This would normally be done in contribute_to_class(), but Django calls that before we've created our extra tree fields on the model (which we need). So it's done here instead, after field setup.

insert_node(*node*, *target*, *position=u'last-child'*, *save=False*, *allow_existing_pk=False*)
Sets up the tree state for `node` (which has not yet been inserted into in the database) so it will be positioned relative to a given `target` node as specified by `position` (when appropriate) it is inserted, with any neccessary space already having been made for it.

A `target` of `None` indicates that `node` should be the last root node.

If `save` is `True`, `node`'s `save()` method will be called before it is returned.

NOTE: This is a low-level method; it does NOT respect `MPTTMeta.order_insertion_by`. In most cases you should just set the node's parent and let mptt call this during save.

left_attr

level_attr

move_node(*node*, *target*, *position=u'last-child'*)
Moves `node` relative to a given `target` node as specified by `position` (when appropriate), by examining both nodes and calling the appropriate method to perform the move.

A `target` of `None` indicates that `node` should be turned into a root node.

Valid values for `position` are `'first-child'`, `'last-child'`, `'left'` or `'right'`.

`node` will be modified to reflect its new tree state in the database.

This method explicitly checks for `node` being made a sibling of a root node, as this is a special case due to our use of tree ids to order root nodes.

NOTE: This is a low-level method; it does NOT respect `MPTTMeta.order_insertion_by`. In most cases you should just move the node yourself by setting node.parent.

parent_attr

partial_rebuild(*tree_id*)
Partially rebuilds a tree i.e. It rebuilds only the tree with given tree_id in database table using *parent* link.

rebuild()
Rebuilds all trees in the database table using *parent* link.

right_attr

root_node(*tree_id*)
Returns the root node of the tree with the given id.

root_nodes()
Creates a `QuerySet` containing root nodes.

tree_id_attr

11.1.6 mptt.models

class mptt.models.**MPTTModel**(**args*, ***kwargs*)
Base class for tree models.

class Meta

abstract = False

MPTTModel.**delete**(**args*, ***kwargs*)
Calling `delete` on a node will delete it as well as its full subtree, as opposed to reattaching all the subnodes to its parent node.

There are no argument specific to a MPTT model, all the arguments will be passed directly to the django's `Model.delete`.

`delete` will not return anything.

MPTTModel.**get_ancestors**(*ascending=False*, *include_self=False*)
Creates a `QuerySet` containing the ancestors of this model instance.

This defaults to being in descending order (root ancestor first, immediate parent last); passing `True` for the `ascending` argument will reverse the ordering (immediate parent first, root ancestor last).

If `include_self` is `True`, the `QuerySet` will also include this model instance.

`MPTTModel.`**`get_children`**`()`
Returns a `QuerySet` containing the immediate children of this model instance, in tree order.

The benefit of using this method over the reverse relation provided by the ORM to the instance's children is that a database query can be avoided in the case where the instance is a leaf node (it has no children).

If called from a template where the tree has been walked by the `cache_tree_children` filter, no database query is required.

`MPTTModel.`**`get_descendant_count`**`()`
Returns the number of descendants this model instance has.

`MPTTModel.`**`get_descendants`**`(`*include_self=False*`)`
Creates a `QuerySet` containing descendants of this model instance, in tree order.

If `include_self` is `True`, the `QuerySet` will also include this model instance.

`MPTTModel.`**`get_family`**`()`
Returns a `QuerySet` containing the ancestors, the model itself and the descendants, in tree order.

`MPTTModel.`**`get_leafnodes`**`(`*include_self=False*`)`
Creates a `QuerySet` containing leafnodes of this model instance, in tree order.

If `include_self` is `True`, the `QuerySet` will also include this model instance (if it is a leaf node)

`MPTTModel.`**`get_level`**`()`
Returns the level of this node (distance from root)

`MPTTModel.`**`get_next_sibling`**`(`**filter_args*, ***filter_kwargs*`)`
Returns this model instance's next sibling in the tree, or `None` if it doesn't have a next sibling.

`MPTTModel.`**`get_previous_sibling`**`(`**filter_args*, ***filter_kwargs*`)`
Returns this model instance's previous sibling in the tree, or `None` if it doesn't have a previous sibling.

`MPTTModel.`**`get_root`**`()`
Returns the root node of this model instance's tree.

`MPTTModel.`**`get_siblings`**`(`*include_self=False*`)`
Creates a `QuerySet` containing siblings of this model instance. Root nodes are considered to be siblings of other root nodes.

If `include_self` is `True`, the `QuerySet` will also include this model instance.

`MPTTModel.`**`insert_at`**`(`*target*, *position=u'first-child'*, *save=False*, *allow_existing_pk=False*`)`
Convenience method for calling `TreeManager.insert_node` with this model instance.

`MPTTModel.`**`is_ancestor_of`**`(`*other*, *include_self=False*`)`
Returns `True` if this model is an ancestor of the given node, `False` otherwise. If include_self is True, also returns True if the two nodes are the same node.

`MPTTModel.`**`is_child_node`**`()`
Returns `True` if this model instance is a child node, `False` otherwise.

`MPTTModel.`**`is_descendant_of`**`(`*other*, *include_self=False*`)`
Returns `True` if this model is a descendant of the given node, `False` otherwise. If include_self is True, also returns True if the two nodes are the same node.

`MPTTModel.`**`is_leaf_node`**`()`
Returns `True` if this model instance is a leaf node (it has no children), `False` otherwise.

MPTTModel.**is_root_node**()
Returns True if this model instance is a root node, False otherwise.

MPTTModel.**move_to**(*target*, *position=u'first-child'*)
Convenience method for calling TreeManager.move_node with this model instance.

NOTE: This is a low-level method; it does NOT respect MPTTMeta.order_insertion_by. In most cases you should just move the node yourself by setting node.parent.

MPTTModel.**save**(**args*, ***kwargs*)
If this is a new node, sets tree fields up before it is inserted into the database, making room in the tree structure as neccessary, defaulting to making the new node the last child of its parent.

It the node's left and right edge indicators already been set, we take this as indication that the node has already been set up for insertion, so its tree fields are left untouched.

If this is an existing node and its parent has been changed, performs reparenting in the tree structure, defaulting to making the node the last child of its new parent.

In either case, if the node's class has its order_insertion_by tree option set, the node will be inserted or moved to the appropriate position to maintain ordering by the specified field.

class mptt.models.**MPTTModelBase**
Metaclass for MPTT models

classmethod **register**(*meta*, *cls*, ***kwargs*)
For the weird cases when you need to add tree-ness to an *existing* class. For other cases you should subclass MPTTModel instead of calling this.

class mptt.models.**MPTTOptions**(*opts=None*, ***kwargs*)
Options class for MPTT models. Use this as an inner class called MPTTMeta:

```
class MyModel(MPTTModel):
    class MPTTMeta:
        order_insertion_by = ['name']
        parent_attr = 'myparent'
```

get_ordered_insertion_target(*node*, *parent*)
Attempts to retrieve a suitable right sibling for node underneath parent (which may be None in the case of root nodes) so that ordering by the fields specified by the node's class' order_insertion_by option is maintained.

Returns None if no suitable sibling can be found.

get_raw_field_value(*instance*, *field_name*)
Gets the value of the given fieldname for the instance. This is not the same as getattr(). This function will return IDs for foreignkeys etc, rather than doing a database query.

insertion_target_filters(*instance*, *order_insertion_by*)
Creates a filter which matches suitable right siblings for node, where insertion should maintain ordering according to the list of fields in order_insertion_by.

For example, given an order_insertion_by of ['field1', 'field2', 'field3'], the resulting filter should correspond to the following SQL:

```
field1 > %s
OR (field1 = %s AND field2 > %s)
OR (field1 = %s AND field2 = %s AND field3 > %s)
```

left_attr = u'lft'

level_attr = u'level'

order_insertion_by = []

parent_attr = u'parent'

right_attr = u'rght'

set_raw_field_value(*instance*, *field_name*, *value*)
Sets the value of the given fieldname for the instance. This is not the same as setattr(). This function requires an ID for a foreignkey (etc) rather than an instance.

tree_id_attr = u'tree_id'

update_mptt_cached_fields(*instance*)

Caches (in an instance._mptt_cached_fields dict) the original values of:

- parent pk
- fields specified in order_insertion_by

These are used in pre_save to determine if the relevant fields have changed, so that the MPTT fields need to be updated.

mptt.models.**classproperty**

class mptt.models.**classpropertytype**(*name*, *bases=()*, *members={}*)

11.1.7 mptt.utils

Utilities for working with lists of model instances which represent trees.

mptt.utils.**previous_current_next**(*items*)
From http://www.wordaligned.org/articles/zippy-triples-served-with-python

Creates an iterator which returns (previous, current, next) triples, with `None` filling in when there is no previous or next available.

mptt.utils.**tree_item_iterator**(*items*, *ancestors=False*)
Given a list of tree items, iterates over the list, generating two-tuples of the current tree item and a `dict` containing information about the tree structure around the item, with the following keys:

'new_level' `True` if the current item is the start of a new level in the tree, `False` otherwise.

'closed_levels' A list of levels which end after the current item. This will be an empty list if the next item is at the same level as the current item.

If `ancestors` is `True`, the following key will also be available:

'ancestors' A list of unicode representations of the ancestors of the current node, in descending order (root node first, immediate parent last).

For example: given the sample tree below, the contents of the list which would be available under the `'ancestors'` key are given on the right:

```
Books                   ->  []
   Sci-fi               ->  [u'Books']
      Dystopian Futures ->  [u'Books', u'Sci-fi']
```

mptt.utils.**drilldown_tree_for_node**(*node*, *rel_cls=None*, *rel_field=None*, *count_attr=None*, *cumulative=False*)
Creates a drilldown tree for the given node. A drilldown tree consists of a node's ancestors, itself and its immediate children, all in tree order.

Optional arguments may be given to specify a `Model` class which is related to the node's class, for the purpose of adding related item counts to the node's children:

rel_cls A `Model` class which has a relation to the node's class.

rel_field The name of the field in `rel_cls` which holds the relation to the node's class.

count_attr The name of an attribute which should be added to each child in the drilldown tree, containing a count of how many instances of `rel_cls` are related through `rel_field`.

cumulative If `True`, the count will be for each child and all of its descendants, otherwise it will be for each child itself.

exception `mptt.`**`AlreadyRegistered`**

Deprecated - don't use this anymore. It's never thrown, you don't need to catch it

`mptt.`**`register`**(**args*, ***kwargs*)

Registers a model class as an MPTTModel, adding MPTT fields and adding MPTTModel to __bases__. This is equivalent to just subclassing MPTTModel, but works for an already-created model.

m