## Contents

1 Community 3  
2 Contributing 5  
3 Changes 7  
4 Offline Reading 9  
  4.1 Tutorial 9  
  4.2 User Guide 13  
  4.3 API Reference 40  
  4.4 Changelog 61  
  4.5 Upgrading 76  
  4.6 Django Support 84  
5 Indices and tables 87  
Python Module Index 89
**MongoEngine** is an Object-Document Mapper, written in Python for working with MongoDB. To install it, simply run

```
$ pip install -U mongoengine
```

**Tutorial**  A quick tutorial building a tumblelog to get you up and running with MongoEngine.

**User Guide**  The Full guide to MongoEngine - from modeling documents to storing files, from querying for data to firing signals and *everything* between.

**API Reference**  The complete API documentation — the innards of documents, querysets and fields.

**Upgrading**  How to upgrade MongoEngine.

**Django Support**  Using MongoEngine and Django
Community

To get help with using MongoEngine, use the MongoDB Users mailing list or the ever popular stackoverflow.
Yes please! We are always looking for contributions, additions and improvements.
The source is available on GitHub and contributions are always encouraged. Contributions can be as simple as minor tweaks to this documentation, the website or the core.
To contribute, fork the project on GitHub and send a pull request.
See the Changelog for a full list of changes to MongoEngine and Upgrading for upgrade information.

**Note:** Always read and test the upgrade documentation before putting updates live in production ;}

---
4.1 Tutorial

This tutorial introduces **MongoEngine** by means of example — we will walk through how to create a simple **Tumblelog** application. A Tumblelog is a type of blog where posts are not constrained to being conventional text-based posts. As well as text-based entries, users may post images, links, videos, etc. For simplicity’s sake, we’ll stick to text, image and link entries in our application. As the purpose of this tutorial is to introduce MongoEngine, we’ll focus on the data-modelling side of the application, leaving out a user interface.

4.1.1 Getting started

Before we start, make sure that a copy of MongoDB is running in an accessible location — running it locally will be easier, but if that is not an option then it may be run on a remote server. If you haven’t installed mongoengine, simply use pip to install it like so:

```bash
$ pip install mongoengine
```

Before we can start using MongoEngine, we need to tell it how to connect to our instance of **mongod**. For this we use the `connect()` function. If running locally the only argument we need to provide is the name of the MongoDB database to use:

```python
from mongoengine import *
connect('tumblelog')
```

There are lots of options for connecting to MongoDB, for more information about them see the *Connecting to MongoDB* guide.

4.1.2 Defining our documents

MongoDB is **schemaless**, which means that no schema is enforced by the database — we may add and remove fields however we want and MongoDB won’t complain. This makes life a lot easier in many regards, especially when there is a change to the data model. However, defining schemata for our documents can help to iron out bugs involving incorrect types or missing fields, and also allow us to define utility methods on our documents in the same way that traditional ORMs (Object-Relational Mappers) do.

In our Tumblelog application we need to store several different types of information. We will need to have a collection of **users**, so that we may link posts to an individual. We also need to store our different types of **posts** (eg: text, image and link) in the database. To aid navigation of our Tumblelog, posts may have **tags** associated with them, so that the list of posts shown to the user may be limited to posts that have been assigned a specific
tag. Finally, it would be nice if comments could be added to posts. We’ll start with users, as the other document models are slightly more involved.

**Users**

Just as if we were using a relational database with an ORM, we need to define which fields a User may have, and what types of data they might store:

```python
class User(Document):
    email = StringField(required=True)
    first_name = StringField(max_length=50)
    last_name = StringField(max_length=50)
```

This looks similar to how a the structure of a table would be defined in a regular ORM. The key difference is that this schema will never be passed on to MongoDB — this will only be enforced at the application level, making future changes easy to manage. Also, the User documents will be stored in a MongoDB collection rather than a table.

**Posts, Comments and Tags**

Now we’ll think about how to store the rest of the information. If we were using a relational database, we would most likely have a table of posts, a table of comments and a table of tags. To associate the comments with individual posts, we would put a column in the comments table that contained a foreign key to the posts table. We’d also need a link table to provide the many-to-many relationship between posts and tags. Then we’d need to address the problem of storing the specialised post-types (text, image and link). There are several ways we can achieve this, but each of them have their problems — none of them stand out as particularly intuitive solutions.

**Posts**

Happily mongoDB isn’t a relational database, so we’re not going to do it that way. As it turns out, we can use MongoDB’s schemaless nature to provide us with a much nicer solution. We will store all of the posts in one collection and each post type will only store the fields it needs. If we later want to add video posts, we don’t have to modify the collection at all, we just start using the new fields we need to support video posts. This fits with the Object-Oriented principle of inheritance nicely. We can think of Post as a base class, and TextPost, ImagePost and LinkPost as subclasses of Post. In fact, MongoEngine supports this kind of modelling out of the box — all you need do is turn on inheritance by setting allow_inheritance to True in the meta:

```python
class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User)

    meta = {'allow_inheritance': True}

class TextPost(Post):
    content = StringField()

class ImagePost(Post):
    image_path = StringField()

class LinkPost(Post):
    link_url = StringField()
```

We are storing a reference to the author of the posts using a ReferenceField object. These are similar to foreign key fields in traditional ORMs, and are automatically translated into references when they are saved, and dereferenced when they are loaded.
Tags

Now that we have our Post models figured out, how will we attach tags to them? MongoDB allows us to store lists of items natively, so rather than having a link table, we can just store a list of tags in each post. So, for both efficiency and simplicity’s sake, we’ll store the tags as strings directly within the post, rather than storing references to tags in a separate collection. Especially as tags are generally very short (often even shorter than a document’s id), this denormalisation won’t impact very strongly on the size of our database. So let’s take a look that the code our modified Post class:

```python
class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User)
    tags = ListField(StringField(max_length=30))
```

The ListField object that is used to define a Post’s tags takes a field object as its first argument — this means that you can have lists of any type of field (including lists).

**Note:** We don’t need to modify the specialised post types as they all inherit from Post.

Comments

A comment is typically associated with one post. In a relational database, to display a post with its comments, we would have to retrieve the post from the database, then query the database again for the comments associated with the post. This works, but there is no real reason to be storing the comments separately from their associated posts, other than to work around the relational model. Using MongoDB we can store the comments as a list of embedded documents directly on a post document. An embedded document should be treated no differently that a regular document; it just doesn’t have its own collection in the database. Using MongoEngine, we can define the structure of embedded documents, along with utility methods, in exactly the same way we do with regular documents:

```python
class Comment(EmbeddedDocument):
    content = StringField()
    name = StringField(max_length=120)
```

We can then store a list of comment documents in our post document:

```python
class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User)
    tags = ListField(StringField(max_length=30))
    comments = ListField(EmbeddedDocumentField(Comment))
```

Handling deletions of references

The ReferenceField object takes a keyword reverse_delete_rule for handling deletion rules if the reference is deleted. To delete all the posts if a user is deleted set the rule:

```python
class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User, reverse_delete_rule=CASCADE)
    tags = ListField(StringField(max_length=30))
    comments = ListField(EmbeddedDocumentField(Comment))
```

See ReferenceField for more information.

**Note:** MapFields and DictFields currently don’t support automatic handling of deleted references
4.1.3 Adding data to our Tumblelog

Now that we’ve defined how our documents will be structured, let’s start adding some documents to the database. Firstly, we’ll need to create a User object:

```python
ross = User(email='ross@example.com', first_name='Ross', last_name='Lawley').save()
```

**Note:** We could have also defined our user using attribute syntax:

```python
ross = User(email='ross@example.com')
ross.first_name = 'Ross'
ross.last_name = 'Lawley'
ross.save()
```

Now that we’ve got our user in the database, let’s add a couple of posts:

```python
post1 = TextPost(title='Fun with MongoEngine', author=john)
post1.content = 'Took a look at MongoEngine today, looks pretty cool.'
post1.tags = ['mongodb', 'mongoengine']
post1.save()

post2 = LinkPost(title='MongoEngine Documentation', author=ross)
post2.link_url = 'http://docs.mongoengine.com/'
post2.tags = ['mongoengine']
post2.save()
```

**Note:** If you change a field on a object that has already been saved, then call `save()` again, the document will be updated.

4.1.4 Accessing our data

So now we’ve got a couple of posts in our database, how do we display them? Each document class (i.e. any class that inherits either directly or indirectly from `Document`) has an `objects` attribute, which is used to access the documents in the database collection associated with that class. So let’s see how we can get our posts’ titles:

```python
for post in Post.objects:
    print post.title
```

Retrieving type-specific information

This will print the titles of our posts, one on each line. But What if we want to access the type-specific data (link_url, content, etc.)? One way is simply to use the `objects` attribute of a subclass of `Post`:

```python
for post in TextPost.objects:
    print post.content
```

Using TextPost's `objects` attribute only returns documents that were created using TextPost. Actually, there is a more general rule here: the `objects` attribute of any subclass of `Document` only looks for documents that were created using that subclass or one of its subclasses.

So how would we display all of our posts, showing only the information that corresponds to each post’s specific type? There is a better way than just using each of the subclasses individually. When we used Post’s `objects` attribute earlier, the objects being returned weren’t actually instances of Post — they were instances of the subclass of Post that matches the post’s type. Let’s look at how this works in practice:

```python
for post in Post.objects:
    print post.title
    print '=' * len(post.title)
```
if isinstance(post, TextPost):
    print post.content

if isinstance(post, LinkPost):
    print 'Link:', post.link_url

print

This would print the title of each post, followed by the content if it was a text post, and “Link: <url>” if it was a link post.

## Searching our posts by tag

The objects attribute of a Document is actually a QuerySet object. This lazily queries the database only when you need the data. It may also be filtered to narrow down your query. Let’s adjust our query so that only posts with the tag “mongodb” are returned:

```python
for post in Post.objects(tags='mongodb'):
    print post.title
```

There are also methods available on QuerySet objects that allow different results to be returned, for example, calling first() on the objects attribute will return a single document, the first matched by the query you provide. Aggregation functions may also be used on QuerySet objects:

```python
num_posts = Post.objects(tags='mongodb').count()
print 'Found %d posts with tag "mongodb"' % num_posts
```

## Learning more about mongoengine

If you got this far you’ve made a great start, so well done! The next step on your mongoengine journey is the full user guide, where you can learn indepth about how to use mongoengine and mongodb.

### 4.2 User Guide

#### 4.2.1 Installing MongoEngine

To use MongoEngine, you will need to download MongoDB and ensure it is running in an accessible location. You will also need PyMongo to use MongoEngine, but if you install MongoEngine using setuptools, then the dependencies will be handled for you.

MongoEngine is available on PyPI, so to use it you can use pip:

```
$ pip install mongoengine
```

Alternatively, if you don’t have setuptools installed, download it from PyPi and run

```
$ python setup.py install
```

To use the bleeding-edge version of MongoEngine, you can get the source from GitHub and install it as above:

```
$ git clone git://github.com/mongoengine/mongoengine
$ cd mongoengine
$ python setup.py install
```
4.2.2 Connecting to MongoDB

To connect to a running instance of mongod, use the `connect()` function. The first argument is the name of the database to connect to:

```python
from mongoengine import connect
connect('project1')
```

By default, MongoEngine assumes that the mongod instance is running on localhost on port 27017. If MongoDB is running elsewhere, you should provide the `host` and `port` arguments to `connect()`:

```python
connect('project1', host='192.168.1.35', port=12345)
```

If the database requires authentication, `username` and `password` arguments should be provided:

```python
connect('project1', username='webapp', password='pwd123')
```

Uri style connections are also supported - just supply the uri as the `host` to `connect()`:

```python
connect('project1', host='mongodb://localhost/database_name')
```

Note that database name from uri has priority over name in `connect()`

**ReplicaSets**

MongoEngine supports `MongoReplicaSetClient` to use them please use a URI style connection and provide the `replicaSet` name in the connection kwargs.

Read preferences are supported through the connection or via individual queries by passing the `read_preference`

```python
Bar.objects().read_preference(ReadPreference.PRIMARY)
Bar.objects(read_preference=ReadPreference.PRIMARY)
```

**Multiple Databases**

Multiple database support was added in MongoEngine 0.6. To use multiple databases you can use `connect()` and provide an `alias` name for the connection - if no `alias` is provided then “default” is used.

In the background this uses `register_connection()` to store the data and you can register all aliases up front if required.

Individual documents can also support multiple databases by providing a `db_alias` in their meta data. This allows `DBRef` objects to point across databases and collections. Below is an example schema, using 3 different databases to store data:

```python
class User(Document):
    name = StringField()
    meta = {"db_alias": "user-db"}

class Book(Document):
    name = StringField()
    meta = {"db_alias": "book-db"}

class AuthorBooks(Document):
    author = ReferenceField(User)
    book = ReferenceField(Book)
    meta = {"db_alias": "users-books-db"}
```
Switch Database Context Manager

Sometimes you may want to switch the database to query against for a class for example, archiving older data into a separate database for performance reasons.

The `switch_db` context manager allows you to change the database alias for a given class allowing quick and easy access to the same User document across databases:

```python
from mongoengine.context_managers import switch_db
class User(Document):
    name = StringField()
    meta = {"db_alias": "user-db"}

with switch_db(User, 'archive-user-db') as User:
    User(name="Ross").save()  # Saves the 'archive-user-db'
```

**Note:** Make sure any aliases have been registered with `register_connection()` before using the context manager.

There is also a switch collection context manager as well. The `switch_collection` context manager allows you to change the collection for a given class allowing quick and easy access to the same Group document across collection:

```python
from mongoengine.context_managers import switch_db
class Group(Document):
    name = StringField()

Group(name="test").save()  # Saves in the default db

with switch_collection(Group, 'group2000') as Group:
    Group(name="hello Group 2000 collection!").save()  # Saves in group2000 collection
```

4.2.3 Defining documents

In MongoDB, a document is roughly equivalent to a row in an RDBMS. When working with relational databases, rows are stored in tables, which have a strict schema that the rows follow. MongoDB stores documents in collections rather than tables - the principle difference is that no schema is enforced at a database level.

Defining a document’s schema

MongoEngine allows you to define schemata for documents as this helps to reduce coding errors, and allows for utility methods to be defined on fields which may be present.

To define a schema for a document, create a class that inherits from `Document`. Fields are specified by adding field objects as class attributes to the document class:

```python
from mongoengine import *
import datetime
class Page(Document):
    title = StringField(max_length=200, required=True)
    date_modified = DateTimeField(default=datetime.datetime.now)
```

As BSON (the binary format for storing data in mongodb) is order dependent, documents are serialized based on their field order.
Dynamic document schemas

One of the benefits of MongoDB is dynamic schemas for a collection, whilst data should be planned and organised (after all explicit is better than implicit!) there are scenarios where having dynamic / expando style documents is desirable.

DynamicDocument documents work in the same way as Document but any data / attributes set to them will also be saved.

```python
from mongoengine import *

class Page(DynamicDocument):
    title = StringField(max_length=200, required=True)

# Create a new page and add tags
>>> page = Page(title='Using MongoEngine')
>>> page.tags = ['mongodb', 'mongoengine']
>>> page.save()

>>> Page.objects(tags='mongoengine').count()
>>> 1
```

Note: There is one caveat on Dynamic Documents: fields cannot start with `_`.

Dynamic fields are stored in creation order after any declared fields.

Fields

By default, fields are not required. To make a field mandatory, set the required keyword argument of a field to True. Fields also may have validation constraints available (such as max_length in the example above). Fields may also take default values, which will be used if a value is not provided. Default values may optionally be a callable, which will be called to retrieve the value (such as in the above example). The field types available are as follows:

- `BinaryField`
- `BooleanField`
- `ComplexDateTimeField`
- `DateTimeField`
- `DecimalField`
- `DictField`
- `DynamicField`
- `EmailField`
- `EmbeddedDocumentField`
- `FileField`
- `FloatField`
- `GenericEmbeddedDocumentField`
- `GenericReferenceField`
- `GeoPointField`
- `ImageField`
- `IntField`
- `ListField`
• MapField
• ObjectIdField
• ReferenceField
• SequenceField
• SortedListField
• StringField
• URLField
• UUIDField

Field arguments

Each field type can be customized by keyword arguments. The following keyword arguments can be set on all fields:

db_field (Default: None) The MongoDB field name.

required (Default: False) If set to True and the field is not set on the document instance, a ValidationError will be raised when the document is validated.

default (Default: None) A value to use when no value is set for this field.

The definition of default parameters follow the general rules on Python, which means that some care should be taken when dealing with default mutable objects (like in ListField or DictField):

```python
class ExampleFirst(Document):
    # Default an empty list
    values = ListField(IntField(), default=list)

class ExampleSecond(Document):
    # Default a set of values
    values = ListField(IntField(), default=lambda: [1,2,3])

class ExampleDangerous(Document):
    # This can make an .append call to add values to the default (and all the following objects), instead to just an object
    values = ListField(IntField(), default=[1,2,3])
```

Note: Unsetting a field with a default value will revert back to the default.

unique (Default: False) When True, no documents in the collection will have the same value for this field.

unique_with (Default: None) A field name (or list of field names) that when taken together with this field, will not have two documents in the collection with the same value.

primary_key (Default: False) When True, use this field as a primary key for the collection. DictField and EmbeddedDocuments both support being the primary key for a document.

choices (Default: None) An iterable (e.g. a list or tuple) of choices to which the value of this field should be limited.

Can be either be a nested tuples of value (stored in mongo) and a human readable key

```python
SIZE = (('S', 'Small'),
        ('M', 'Medium'),
        ('L', 'Large'),
        ('XL', 'Extra Large'),
        ('XXL', 'Extra Extra Large'))
```
class Shirt (Document):
    size = StringField(max_length=3, choices=SIZE)

Or a flat iterable just containing values

SIZE = ('S', 'M', 'L', 'XL', 'XXL')

class Shirt (Document):
    size = StringField(max_length=3, choices=SIZE)

help_text (Default: None) Optional help text to output with the field - used by form libraries

verbose_name (Default: None) Optional human-readable name for the field - used by form libraries

List fields

MongoDB allows the storage of lists of items. To add a list of items to a Document, use the ListField field type. ListField takes another field object as its first argument, which specifies which type elements may be stored within the list:

class Page (Document):
    tags = ListField(StringField(max_length=50))

Embedded documents

MongoDB has the ability to embed documents within other documents. Schemata may be defined for these embedded documents, just as they may be for regular documents. To create an embedded document, just define a document as usual, but inherit from EmbeddedDocument rather than Document:

class Comment (EmbeddedDocument):
    content = StringField()

To embed the document within another document, use the EmbeddedDocumentField field type, providing the embedded document class as the first argument:

class Page (Document):
    comments = ListField(EmbeddedDocumentField(Comment))

comment1 = Comment(content='Good work!')
comment2 = Comment(content='Nice article!')
page = Page(comments=[comment1, comment2])

Dictionary Fields

Often, an embedded document may be used instead of a dictionary – generally this is recommended as dictionaries don’t support validation or custom field types. However, sometimes you will not know the structure of what you want to store; in this situation a DictField is appropriate:

class SurveyResponse (Document):
    date = DateTimeField()
    user = ReferenceField(User)
    answers = DictField()

survey_response = SurveyResponse(date=datetime.now(), user=request.user)
response_form = ResponseForm(request.POST)
survey_response.answers = response_form.cleaned_data()
survey_response.save()

Dictionaries can store complex data, other dictionaries, lists, references to other objects, so are the most flexible field type available.
Reference fields

References may be stored to other documents in the database using the `ReferenceField`. Pass in another document class as the first argument to the constructor, then simply assign document objects to the field:

```python
class User(Document):
    name = StringField()

class Page(Document):
    content = StringField()
    author = ReferenceField(User)

john = User(name="John Smith")
john.save()

post = Page(content="Test Page")
post.author = john
post.save()
```

The `User` object is automatically turned into a reference behind the scenes, and dereferenced when the `Page` object is retrieved.

To add a `ReferenceField` that references the document being defined, use the string `'self'` in place of the document class as the argument to `ReferenceField`'s constructor. To reference a document that has not yet been defined, use the name of the undefined document as the constructor’s argument:

```python
class Employee(Document):
    name = StringField()
    boss = ReferenceField('self')
    profile_page = ReferenceField('ProfilePage')

class ProfilePage(Document):
    content = StringField()
```

One to Many with ListFields  If you are implementing a one to many relationship via a list of references, then the references are stored as DBRefs and to query you need to pass an instance of the object to the query:

```python
class User(Document):
    name = StringField()

class Page(Document):
    content = StringField()
    authors = ListField(ReferenceField(User))

bob = User(name="Bob Jones").save()
john = User(name="John Smith").save()

Page(content="Test Page", authors=[bob, john]).save()
Page(content="Another Page", authors=[john]).save()

# Find all pages Bob authored
Page.objects(authors__in=[bob])

# Find all pages that both Bob and John have authored
Page.objects(authors__all=[bob, john])

# Remove Bob from the authors for a page.
Page.objects(id='...').update_one(pull__authors=bob)

# Add John to the authors for a page.
Page.objects(id='...').update_one(push__authors= john)
```
Dealing with deletion of referred documents  By default, MongoDB doesn’t check the integrity of your data, so deleting documents that other documents still hold references to will lead to consistency issues. Mongoengine’s ReferenceField adds some functionality to safeguard against these kinds of database integrity problems, providing each reference with a delete rule specification. A delete rule is specified by supplying the reverse_delete_rule attributes on the ReferenceField definition, like this:

```python
class Employee(Document):
    ...  
    profile_page = ReferenceField('ProfilePage', reverse_delete_rule=mongoengine.NULLIFY)
```

The declaration in this example means that when an Employee object is removed, the ProfilePage that belongs to that employee is removed as well. If a whole batch of employees is removed, all profile pages that are linked are removed as well.

Its value can take any of the following constants:

- **mongoengine.DO NOTHING** This is the default and won’t do anything. Deletes are fast, but may cause database inconsistency or dangling references.
- **mongoengine.DENY** Deletion is denied if there still exist references to the object being deleted.
- **mongoengine.NULLIFY** Any object’s fields still referring to the object being deleted are removed (using MongoDB’s “unset” operation), effectively nullifying the relationship.
- **mongoengine.CASCADE** Any object containing fields that are refererring to the object being deleted are deleted first.
- **mongoengine.PULL** Removes the reference to the object (using MongoDB’s “pull” operation) from any object’s fields of ListField(ReferenceField).

**Warning:** A safety note on setting up these delete rules! Since the delete rules are not recorded on the database level by MongoDB itself, but instead at runtime, in-memory, by the MongoEngine module, it is of the utmost importance that the module that declares the relationship is loaded **BEFORE** the delete is invoked. If, for example, the Employee object lives in the payroll app, and the ProfilePage in the people app, it is extremely important that the people app is loaded before any employee is removed, because otherwise, MongoEngine could never know this relationship exists.

In Django, be sure to put all apps that have such delete rule declarations in their models.py in the INSTALLED_APPS tuple.

**Warning:** Signals are not triggered when doing cascading updates / deletes - if this is required you must manually handle the update / delete.

Generic reference fields  A second kind of reference field also exists, GenericReferenceField. This allows you to reference any kind of Document, and hence doesn’t take a Document subclass as a constructor argument:

```python
class Link(Document):
    url = StringField()

class Post(Document):
    title = StringField()

class Bookmark(Document):
    bookmark_object = GenericReferenceField()

link = Link(url='http://hmarr.com/mongoengine/')
link.save()

post = Post(title='Using MongoEngine')
post.save()
```
Bookmark(bookmark_object=link).save()
Bookmark(bookmark_object=post).save()

**Note:** Using *GenericReferenceField*s is slightly less efficient than the standard *ReferenceField*s, so if you will only be referencing one document type, prefer the standard *ReferenceField*.

### Uniqueness constraints

MongoEngine allows you to specify that a field should be unique across a collection by providing `unique=True` to a Field's constructor. If you try to save a document that has the same value for a unique field as a document that is already in the database, a `OperationError` will be raised. You may also specify multi-field uniqueness constraints by using `unique_with`, which may be either a single field name, or a list or tuple of field names:

```python
class User(Document):
    username = StringField(unique=True)
    first_name = StringField()
    last_name = StringField(unique_with='first_name')
```

### Skipping Document validation on save

You can also skip the whole document validation process by setting `validate=False` when calling the `save()` method:

```python
class Recipient(Document):
    name = StringField()
    email = EmailField()

recipient = Recipient(name='admin', email='root@localhost')
recipient.save()  # will raise a ValidationError while
recipient.save(validate=False)  # won't
```

### Document collections

Document classes that inherit directly from *Document* will have their own *collection* in the database. The name of the collection is by default the name of the class, coverted to lowercase (so in the example above, the collection would be called *page*). If you need to change the name of the collection (e.g. to use MongoEngine with an existing database), then create a class dictionary attribute called *meta* on your document, and set *collection* to the name of the collection that you want your document class to use:

```python
class Page(Document):
    title = StringField(max_length=200, required=True)
    meta = {'collection': 'cmsPage'}
```

### Capped collections

A *Document* may use a Capped Collection by specifying `max_documents` and `max_size` in the *meta* dictionary. `max_documents` is the maximum number of documents that is allowed to be stored in the collection, and `max_size` is the maximum size of the collection in bytes. If `max_size` is not specified and `max_documents` is, `max_size` defaults to 10000000 bytes (10MB). The following example shows a Log document that will be limited to 1000 entries and 2MB of disk space:

```python
class Log(Document):
    ip_address = StringField()
    meta = {'max_documents': 1000, 'max_size': 2000000}
```
Indexes

You can specify indexes on collections to make querying faster. This is done by creating a list of index specifications called `indexes` in the `meta` dictionary, where an index specification may either be a single field name, a tuple containing multiple field names, or a dictionary containing a full index definition. A direction may be specified on fields by prefixing the field name with a `+` (for ascending) or a `-` sign (for descending). Note that direction only matters on multi-field indexes.

```python
class Page(Document):
    title = StringField()
    rating = StringField()
    meta = {
        'indexes': ['title', ('title', '-rating')]
    }
```

If a dictionary is passed then the following options are available:

- **fields (Default: None)** The fields to index. Specified in the same format as described above.
- **cls (Default: True)** If you have polymorphic models that inherit and have `allow_inheritance` turned on, you can configure whether the index should have the `_cls` field added automatically to the start of the index.
- **sparse (Default: False)** Whether the index should be sparse.
- **unique (Default: False)** Whether the index should be unique.
- **expireAfterSeconds (Optional)** Allows you to automatically expire data from a collection by setting the time in seconds to expire the a field.

**Note:** Inheritance adds extra fields indices see: [Document inheritance](#).

Global index default options

There are a few top level defaults for all indexes that can be set:

```python
class Page(Document):
    title = StringField()
    rating = StringField()
    meta = {
        'index_options': {},
        'index_background': True,
        'index_drop_dups': True,
        'index_cls': False
    }
```

- **index_options (Optional)** Set any default index options - see the full options list
- **index_background (Optional)** Set the default value for if an index should be indexed in the background
- **index_drop_dups (Optional)** Set the default value for if an index should drop duplicates
- **index_cls (Optional)** A way to turn off a specific index for `_cls`.

Compound Indexes and Indexing sub documents

Compound indexes can be created by adding the Embedded field or dictionary field name to the index definition. Sometimes it's more efficient to index parts of Embedded / dictionary fields, in this case use ‘dot’ notation to identify the value to index eg: `rank.title`
Geospatial indexes

The best geo index for mongodb is the new “2dsphere”, which has an improved spherical model and provides better performance and more options when querying. The following fields will explicitly add a “2dsphere” index:

- `PointField`
- `LineStringField`
- `PolygonField`

As “2dsphere” indexes can be part of a compound index, you may not want the automatic index but would prefer a compound index. In this example we turn off auto indexing and explicitly declare a compound index on `location` and `datetime`:

```python
class Log(Document):
    location = PointField(auto_index=False)
    datetime = DateTimeField()
    meta = {
        'indexes': [("location", "2dsphere"), ("datetime", 1)]
    }
```

Pre MongoDB 2.4 Geo

Note: For MongoDB < 2.4 this is still current, however the new 2dsphere index is a big improvement over the previous 2D model - so upgrading is advised.

Geospatial indexes will be automatically created for all `GeoPointField`s.

It is also possible to explicitly define geospatial indexes. This is useful if you need to define a geospatial index on a subfield of a `DictField` or a custom field that contains a point. To create a geospatial index you must prefix the field with the * sign.

```python
class Place(Document):
    location = DictField()
    meta = {
        'indexes': ["*location.point"],
    }
```

Time To Live indexes

A special index type that allows you to automatically expire data from a collection after a given period. See the official ttl documentation for more information. A common usecase might be session data:

```python
class Session(Document):
    created = DateTimeField(default=datetime.now)
    meta = {
        'indexes': [  
            {'fields': ['created'], 'expireAfterSeconds': 3600}  
        ]
    }
```

Warning: TTL indexes happen on the MongoDB server and not in the application code, therefore no signals will be fired on document deletion. If you need signals to be fired on deletion, then you must handle the deletion of Documents in your application code.
Comparing Indexes

Use `mongoengine.Document.compare_indexes()` to compare actual indexes in the database to those that your document definitions define. This is useful for maintenance purposes and ensuring you have the correct indexes for your schema.

Ordering

A default ordering can be specified for your `QuerySet` using the `ordering` attribute of `meta`. Ordering will be applied when the `QuerySet` is created, and can be overridden by subsequent calls to `order_by()`.

```python
from datetime import datetime

class BlogPost(Document):
    title = StringField()
    published_date = DateTimeField()

    meta = {
        'ordering': ['published_date']
    }

blog_post_1 = BlogPost(title='Blog Post #1')
blog_post_1.published_date = datetime(2010, 1, 5, 0, 0, 0)

blog_post_2 = BlogPost(title='Blog Post #2')
blog_post_2.published_date = datetime(2010, 1, 6, 0, 0, 0)

blog_post_3 = BlogPost(title='Blog Post #3')
blog_post_3.published_date = datetime(2010, 1, 7, 0, 0, 0)

blog_post_1.save()
blog_post_2.save()
blog_post_3.save()

# get the "first" BlogPost using default ordering
# from BlogPost.meta.ordering
latest_post = BlogPost.objects.first()
assert latest_post.title == 'Blog Post #3'

# override default ordering, order BlogPosts by "published_date"
first_post = BlogPost.objects.order_by('+published_date').first()
assert first_post.title == 'Blog Post #1'
```

Shard keys

If your collection is sharded, then you need to specify the shard key as a tuple, using the `shard_key` attribute of `mongoengine.Document.meta`. This ensures that the shard key is sent with the query when calling the `save()` or `update()` method on an existing `mongoengine.Document` instance:

```python
class LogEntry(Document):
    machine = StringField()
    app = StringField()
    timestamp = DateTimeField()
    data = StringField()

    meta = {
        'shard_key': ('machine', 'timestamp'),
    }
```
Document inheritance

To create a specialised type of a `Document` you have defined, you may subclass it and add any extra fields or methods you may need. As this is new class is not a direct subclass of `Document`, it will not be stored in its own collection; it will use the same collection as its superclass uses. This allows for more convenient and efficient retrieval of related documents - all you need do is set `allow_inheritance` to True in the `meta` data for a document:

```python
# Stored in a collection named 'page'
class Page(Document):
    title = StringField(max_length=200, required=True)

    meta = {'allow_inheritance': True}

# Also stored in the collection named 'page'
class DatedPage(Page):
    date = DateTimeField()
```

**Note:** From 0.8 onwards you must declare `allow_inheritance` defaults to False, meaning you must set it to True to use inheritance.

Working with existing data

As MongoEngine no longer defaults to needing `_cls` you can quickly and easily get working with existing data. Just define the document to match the expected schema in your database

```python
# Will work with data in an existing collection named 'cmsPage'
class Page(Document):
    title = StringField(max_length=200, required=True)

    meta = {
        'collection': 'cmsPage'
    }
```

If you have wildly varying schemas then using a `DynamicDocument` might be more appropriate, instead of defining all possible field types.

If you use `Document` and the database contains data that isn’t defined then that data will be stored in the `document._data` dictionary.

Abstract classes

If you want to add some extra functionality to a group of Document classes but you don’t need or want the overhead of inheritance you can use the `abstract` attribute of `-mongoengine.Document.meta`. This won’t turn on `Document inheritance` but will allow you to keep your code DRY:

```python
class BaseDocument(Document):
    meta = {
        'abstract': True,
    }

    def check_permissions(self):
        ...

class User(BaseDocument):
    ...
```

Now the User class will have access to the inherited `check_permissions` method and won’t store any of the extra `_cls` information.
4.2.4 Documents instances

To create a new document object, create an instance of the relevant document class, providing values for its fields as its constructor keyword arguments. You may provide values for any of the fields on the document:

```python
>>> page = Page(title="Test Page")
>>> page.title
'Test Page'
```

You may also assign values to the document’s fields using standard object attribute syntax:

```python
>>> page.title = "Example Page"
>>> page.title
'Example Page'
```

Saving and deleting documents

MongoEngine tracks changes to documents to provide efficient saving. To save the document to the database, call the `save()` method. If the document does not exist in the database, it will be created. If it does already exist, then any changes will be updated atomically. For example:

```python
>>> page = Page(title="Test Page")
>>> page.save()  # Performs an insert
>>> page.title = "My Page"
>>> page.save()  # Performs an atomic set on the title field.
```

Note: Changes to documents are tracked and on the whole perform `set` operations.

- `list_field.push(0)` - `sets` the resulting list
- `del(list_field)` - `unsets` whole list

With lists it's preferable to use `Doc.update(push__list_field=0)` as this stops the whole list being updated - stopping any race conditions.

See also:

* Atomic updates

Pre save data validation and cleaning

MongoEngine allows you to create custom cleaning rules for your documents when calling `save()`. By providing a custom `clean()` method you can do any pre validation / data cleaning.

This might be useful if you want to ensure a default value based on other document values for example:

```python
class Essay(Document):
    status = StringField(choices=('Published', 'Draft'), required=True)
    pub_date = DateTimeField()

    def clean(self):
        """Ensures that only published essays have a 'pub_date' and
        automatically sets the pub_date if published and not set""
        if self.status == 'Draft' and self.pub_date is not None:
            msg = 'Draft entries should not have a publication date.'
            raise ValidationError(msg)
        # Set the pub_date for published items if not set.
        if self.status == 'Published' and self.pub_date is None:
            self.pub_date = datetime.now()
```

Note: Cleaning is only called if validation is turned on and when calling `save()`.
Cascading Saves

If your document contains `ReferenceField` or `GenericReferenceField` objects, then by default the `save()` method will not save any changes to those objects. If you want all references to also be saved also, noting each save is a separate query, then passing `cascade` as True to the save method will cascade any saves.

Deleting documents

To delete a document, call the `delete()` method. Note that this will only work if the document exists in the database and has a valid id.

Document IDs

Each document in the database has a unique id. This may be accessed through the `id` attribute on `Document` objects. Usually, the id will be generated automatically by the database server when the object is save, meaning that you may only access the `id` field once a document has been saved:

```python
>>> page = Page(title="Test Page")
>>> page.id
>>> page.save()
>>> page.id
ObjectId('123456789abcdef000000000')
```

Alternatively, you may define one of your own fields to be the document’s “primary key” by providing `primary_key=True` as a keyword argument to a field's constructor. Under the hood, MongoEngine will use this field as the id; in fact id is actually aliased to your primary key field so you may still use id to access the primary key if you want:

```python
>>> class User(Document):
...     email = StringField(primary_key=True)
...     name = StringField()
...     ...
>>> bob = User(email='bob@example.com', name='Bob')
>>> bob.save()
>>> bob.id == bob.email == 'bob@example.com'
True
```

You can also access the document’s “primary key” using the `pk` field; in is an alias to `id`:

```python
>>> page = Page(title="Another Test Page")
>>> page.save()
>>> page.id == page.pk
```

Note: If you define your own primary key field, the field implicitly becomes required, so a `ValidationError` will be thrown if you don’t provide it.

4.2.5 Querying the database

`Document` classes have an `objects` attribute, which is used for accessing the objects in the database associated with the class. The `objects` attribute is actually a `QuerySetManager`, which creates and returns a new `QuerySet` object on access. The `QuerySet` object may be iterated over to fetch documents from the database:

```python
# Prints out the names of all the users in the database
for user in User.objects:
    print user.name
```

Note: As of MongoEngine 0.8 the querysets utilise a local cache. So iterating it multiple times will only cause a
single query. If this is not the desired behaviour you can call `no_cache` (version 0.8.3+) to return a non-caching queryset.

### Filtering queries

The query may be filtered by calling the `QuerySet` object with field lookup keyword arguments. The keys in the keyword arguments correspond to fields on the `Document` you are querying:

```python
# This will return a QuerySet that will only iterate over users whose 'country' field is set to 'uk'
uk_users = User.objects(country='uk')
```

Fields on embedded documents may also be referred to using field lookup syntax by using a double-underscore in place of the dot in object attribute access syntax:

```python
# This will return a QuerySet that will only iterate over pages that have been written by a user whose 'country' field is set to 'uk'
uk_pages = Page.objects(author__country='uk')
```

### Query operators

Operators other than equality may also be used in queries; just attach the operator name to a key with a double-underscore:

```python
# Only find users whose age is 18 or less
young_users = Users.objects(age__lte=18)
```

Available operators are as follows:

- `ne` – not equal to
- `lt` – less than
- `lte` – less than or equal to
- `gt` – greater than
- `gte` – greater than or equal to
- `not` – negate a standard check, may be used before other operators (e.g. `Q(age__not__mod=5)`)
- `in` – value is in list (a list of values should be provided)
- `nin` – value is not in list (a list of values should be provided)
- `mod` – value % x == y, where x and y are two provided values
- `all` – every item in list of values provided is in array
- `size` – the size of the array is
- `exists` – value for field exists

### String queries

The following operators are available as shortcuts to querying with regular expressions:

- `exact` – string field exactly matches value
- `iexact` – string field exactly matches value (case insensitive)
- `contains` – string field contains value
- `icontains` – string field contains value (case insensitive)
• `startswith` – string field starts with value
• `istartswith` – string field starts with value (case insensitive)
• `endswith` – string field ends with value
• `iendswith` – string field ends with value (case insensitive)
• `match` – performs an `$elemMatch` so you can match an entire document within an array

**Geo queries**

There are a few special operators for performing geographical queries. The following were added in 0.8 for: `PointField`, `LineStringField` and `PolygonField`:

- **`geo_within`** – Check if a geometry is within a polygon. For ease of use it accepts either a geojson geometry or just the polygon coordinates eg:

```python
loc.objects(point__geo_within=[[40, 5], [40, 6], [41, 6], [40, 5]])
loc.objects(point__geo_within={"type": "Polygon",
    "coordinates": [[40, 5], [40, 6], [41, 6], [40, 5]])
```

- **`geo_within_box`** – simplified `geo_within` searching with a box eg:

```python
loc.objects(point__geo_within_box=[(-125.0, 35.0), (-100.0, 40.0)])
loc.objects(point__geo_within_box=[<bottom left coordinates>, <upper right coordinates>])
```

- **`geo_within_polygon`** – simplified `geo_within` searching within a simple polygon eg:

```python
loc.objects(point__geo_within_polygon=[[40, 5], [40, 6], [41, 6], [40, 5]])
loc.objects(point__geo_within_polygon=[[<x1>, <y1>],
                                      [<x2>, <y2>],
                                      [<x3>, <y3>]])
```

- **`geo_within_center`** – simplified `geo_within` the flat circle radius of a point eg:

```python
loc.objects(point__geo_within_center=[(-125.0, 35.0), 1])
loc.objects(point__geo_within_center=[[<x>, <y>], <radius>])
```

- **`geo_within_sphere`** – simplified `geo_within` the spherical circle radius of a point eg:

```python
loc.objects(point__geo_within_sphere=[(-125.0, 35.0), 1])
loc.objects(point__geo_within_sphere=[[<x>, <y>], <radius>])
```

- **`geo_intersects`** – selects all locations that intersect with a geometry eg:

```python
# Inferred from provided points lists:
loc.objects(poly__geo_intersects=[40, 6])
loc.objects(poly__geo_intersects=[[40, 5], [40, 6]])
loc.objects(poly__geo_intersects=[[40, 5], [40, 6], [41, 6], [41, 5], [40, 5]])

# With geoJson style objects
loc.objects(poly__geo_intersects="{"type": "Point", "coordinates": [40, 6]})
loc.objects(poly__geo_intersects="{"type": "LineString",
    "coordinates": [[40, 5], [40, 6]]}
loc.objects(poly__geo_intersects="{"type": "Polygon",
    "coordinates": [[40, 5], [40, 6], [41, 6], [41, 5], [40, 5]]})
```

- **`near`** – Find all the locations near a given point:

```python
loc.objects(point__near=[40, 5])
loc.objects(point__near={"type": "Point", "coordinates": [40, 5]})
```

You can also set the maximum distance in meters as well::
The older 2D indexes are still supported with the `GeoPointField`:

- **within_distance** – provide a list containing a point and a maximum distance (e.g. [41.342, -87.653], 5)
- **within_spherical_distance** – Same as above but using the spherical geo model (e.g. [41.342, -87.653], 5/earth_radius)
- **near** – order the documents by how close they are to a given point
- **near_sphere** – Same as above but using the spherical geo model
- **within_box** – filter documents to those within a given bounding box (e.g. [(35.0, -125.0), (40.0, -100.0)])
- **within_polygon** – filter documents to those within a given polygon (e.g. [(41.91,-87.69), (41.92,-87.68), (41.91,-87.65), (41.89,-87.65)]).

**Note:** Requires Mongo Server 2.0

- **max_distance** – can be added to your location queries to set a maximum distance.

### Querying lists

On most fields, this syntax will look up documents where the field specified matches the given value exactly, but when the field refers to a `ListField`, a single item may be provided, in which case lists that contain that item will be matched:

```python
class Page(Document):
    tags = ListField(StringField())

# This will match all pages that have the word 'coding' as an item in the tags list
Page.objects(tags=['coding'])
```

It is possible to query by position in a list by using a numerical value as a query operator. So if you wanted to find all pages whose first tag was `db`, you could use the following query:

```python
Page.objects(tags__0='db')
```

If you only want to fetch part of a list eg: you want to paginate a list, then the `slice` operator is required:

```python
# comments - skip 5, limit 10
Page.objects.fields(slice__comments=[5, 10])
```

For updating documents, if you don’t know the position in a list, you can use the $ positional operator

```python
Post.objects(comments__by="joe").update(**{'inc__comments__$__votes': 1})
```

However, this doesn’t map well to the syntax so you can also use a capital S instead

```python
Post.objects(comments__by="joe").update(inc__comments__S__votes=1)
```

**.. note::** Due to Mongo currently the $ operator only applies to the first matched item in the query.

### Raw queries

It is possible to provide a raw PyMongo query as a query parameter, which will be integrated directly into the query. This is done using the `__raw__` keyword argument:
Limiting and skipping results

Just as with traditional ORMs, you may limit the number of results returned, or skip a number or results in your query. `limit()` and `skip()` methods are available on `QuerySet` objects, but the preferred syntax for achieving this is using array-slicing syntax:

```python
# Only the first 5 people
users = User.objects[:5]

# All except for the first 5 people
users = User.objects[5:]

# 5 users, starting from the 10th user found
users = User.objects[10:15]
```

You may also index the query to retrieve a single result. If an item at that index does not exist, an `IndexError` will be raised. A shortcut for retrieving the first result and returning `None` if no result exists is provided (`first()`):

```python
>>> # Make sure there are no users
>>> User.drop_collection()
IndexError: list index out of range
>>> User.objects.first() == None
True
>>> User(name='Test User').save()
>>> User.objects[0] == User.objects.first()
True
```

Retrieving unique results

To retrieve a result that should be unique in the collection, use `get()`. This will raise `DoesNotExist` if no document matches the query, and `MultipleObjectsReturned` if more than one document matched the query. These exceptions are merged into your document definitions eg: `MyDoc.DoesNotExist`

A variation of this method exists, `get_or_create()`, that will create a new document with the query arguments if no documents match the query. An additional keyword argument, `defaults` may be provided, which will be used as default values for the new document, in the case that it should need to be created:

```python
>>> a, created = User.objects.get_or_create(name='User A', defaults={'age': 30})
>>> b, created = User.objects.get_or_create(name='User A', defaults={'age': 40})
>>> a.name == b.name and a.age == b.age
True
```

Default Document queries

By default, the `objects` attribute on a document returns a `QuerySet` that doesn’t filter the collection – it returns all objects. This may be changed by defining a method on a document that modifies a queryset. The method should accept two arguments – `doc_cls` and `queryset`. The first argument is the `Document` class that the method is defined on (in this sense, the method is more like a `classmethod()` than a regular method), and the second argument is the initial queryset. The method needs to be decorated with `queryset_manager()` in order for it to be recognised.
You don’t need to call your method `objects` – you may define as many custom manager methods as you like:

```python
class BlogPost(Document):
    title = StringField()
    published = BooleanField()

    @queryset_manager
    def live_posts(doc_cls, queryset):
        return queryset.filter(published=True)

BlogPost(title='test1', published=False).save()
BlogPost(title='test2', published=True).save()
assert len(BlogPost.objects) == 2
assert len(BlogPost.live_posts()) == 1
```

### Custom QuerySets

Should you want to add custom methods for interacting with or filtering documents, extending the `QuerySet` class may be the way to go. To use a custom `QuerySet` class on a document, set `queryset_class` to the custom class in a `Document`'s `meta` dictionary:

```python
class AwesomerQuerySet (QuerySet):
    def get_awesome(self):
        return self.filter(awesome=True)

class Page (Document):
    meta = {'queryset_class': AwesomerQuerySet}

# To call:
Page.objects.get_awesome()
```

New in version 0.4.

### Aggregation

MongoDB provides some aggregation methods out of the box, but there are not as many as you typically get with an RDBMS. MongoEngine provides a wrapper around the built-in methods and provides some of its own, which are implemented as Javascript code that is executed on the database server.

### Counting results

Just as with limiting and skipping results, there is a method on `QuerySet objects` – `count()`, but there is also a more Pythonic way of achieving this:

```
num_users = len(User.objects)
```
Further aggregation

You may sum over the values of a specific field on documents using `sum()`:

```
yearly_expense = Employee.objects.sum('salary')
```

**Note:** If the field isn’t present on a document, that document will be ignored from the sum.

To get the average (mean) of a field on a collection of documents, use `average()`:

```
mean_age = User.objects.average('age')
```

As MongoDB provides native lists, MongoEngine provides a helper method to get a dictionary of the frequencies of items in lists across an entire collection – `item_frequencies()`. An example of its use would be generating “tag-clouds”:

```
class Article(Document):
    tag = ListField(StringField())

# After adding some tagged articles...
tag_freqs = Article.objects.item_frequencies('tag', normalize=True)

from operator import itemgetter
top_tags = sorted(tag_freqs.items(), key=itemgetter(1), reverse=True)[:10]
```

Query efficiency and performance

There are a couple of methods to improve efficiency when querying, reducing the information returned by the query or efficient dereferencing.

Retrieving a subset of fields

Sometimes a subset of fields on a `Document` is required, and for efficiency only these should be retrieved from the database. This issue is especially important for MongoDB, as fields may often be extremely large (e.g. a `ListField` of `EmbeddedDocument`s, which represent the comments on a blog post. To select only a subset of fields, use `only()`, specifying the fields you want to retrieve as its arguments. Note that if fields that are not downloaded are accessed, their default value (or `None` if no default value is provided) will be given:

```
>>> class Film(Document):
...    title = StringField()
...    year = IntField()
...    rating = IntField(default=3)
...    
>>> Film(title='The Shawshank Redemption', year=1994, rating=5).save()

>>> f = Film.objects.only('title').first()

>>> f.title
'The Shawshank Redemption'

>>> f.year  # None

>>> f.rating  # default value
3
```

**Note:** The `exclude()` is the opposite of `only()` if you want to exclude a field.

If you later need the missing fields, just call `reload()` on your document.
Getting related data

When iterating the results of `ListField` or `DictField` we automatically dereference any DBRef objects as efficiently as possible, reducing the number the queries to mongo.

There are times when that efficiency is not enough, documents that have `ReferenceField` objects or `GenericReferenceField` objects at the top level are expensive as the number of queries to MongoDB can quickly rise.

To limit the number of queries use `select_related()` which converts the QuerySet to a list and dereferences as efficiently as possible. By default `select_related()` only dereferences any references to the depth of 1 level. If you have more complicated documents and want to dereference more of the object at once then increasing the `max_depth` will dereference more levels of the document.

Turning off dereferencing

Sometimes for performance reasons you don’t want to automatically dereference data. To turn off dereferencing of the results of a query use `no_dereference()` on the queryset like so:

```python
post = Post.objects.no_dereference().first()
assert(isinstance(post.author, ObjectId))
```

You can also turn off all dereferencing for a fixed period by using the `no_dereference` context manager:

```python
with no_dereference(Post) as Post:
    post = Post.objects.first()
    assert(isinstance(post.author, ObjectId))
assert(isinstance(post.author, User))
```

Advanced queries

Sometimes calling a `QuerySet` object with keyword arguments can’t fully express the query you want to use – for example if you need to combine a number of constraints using `and` and `or`. This is made possible in MongoEngine through the `Q` class. A `Q` object represents part of a query, and can be initialised using the same keyword-argument syntax you use to query documents. To build a complex query, you may combine `Q` objects using the `&` (and) and `|` (or) operators. To use a `Q` object, pass it in as the first positional argument to `Document.objects` when you filter it by calling it with keyword arguments:

```python
# Get published posts
Post.objects(Q(published=True) | Q(publish_date__lte=datetime.now()))

# Get top posts
Post.objects((Q(featured=True) & Q(hits__gte=1000)) | Q(hits__gte=5000))
```

**Warning:** You have to use bitwise operators. You cannot use `or` and to combine queries as `Q(a=a) or Q(b=b)` is not the same as `Q(a=a) | Q(b=b)`. As `Q(a=a)` equates to true `Q(a=a) or Q(b=b)` is the same as `Q(a=a)`.

Atomic updates

Documents may be updated atomically by using the `update_one()` and `update()` methods on a `QuerySet()`. There are several different “modifiers” that you may use with these methods:

- `set` – set a particular value
- `unset` – delete a particular value (since MongoDB v1.3+)
- `inc` – increment a value by a given amount
- `dec` – decrement a value by a given amount
- `push` – append a value to a list
- `push_all` – append several values to a list
- `pop` – remove the first or last element of a list
- `pull` – remove a value from a list
- `pull_all` – remove several values from a list
- `add_to_set` – add value to a list only if its not in the list already

The syntax for atomic updates is similar to the querying syntax, but the modifier comes before the field, not after it:

```python
>>> post = BlogPost(title='Test', page_views=0, tags=['database'])
>>> post.save()
>>> BlogPost.objects(id=post.id).update_one(inc__page_views=1)
>>> post.reload()  # the document has been changed, so we need to reload it
>>> post.page_views
1
>>> BlogPost.objects(id=post.id).update_one(set__title='Example Post')
>>> post.reload()
>>> post.title
'Example Post'
>>> BlogPost.objects(id=post.id).update_one(push__tags='nosql')
>>> post.reload()
>>> post.tags
['database', 'nosql']
```

Note: In version 0.5 the `save()` runs atomic updates on changed documents by tracking changes to that document.

The positional operator allows you to update list items without knowing the index position, therefore making the update a single atomic operation. As we cannot use the `$` syntax in keyword arguments it has been mapped to `S`:

```python
>>> post = BlogPost(title='Test', page_views=0, tags=['database', 'mongo'])
>>> post.save()
>>> BlogPost.objects(id=post.id, tags='mongo').update(set__tags__S='mongodb')
>>> post.reload()
>>> post.tags
['database', 'mongodb']
```

Note: Currently only top level lists are handled, future versions of mongodb / pymongo plan to support nested positional operators. See The `$` positional operator.

Server-side javascript execution

Javascript functions may be written and sent to the server for execution. The result of this is the return value of the Javascript function. This functionality is accessed through the `exec_js()` method on `QuerySet()` objects. Pass in a string containing a Javascript function as the first argument.

The remaining positional arguments are names of fields that will be passed into you Javascript function as its arguments. This allows functions to be written that may be executed on any field in a collection (e.g. the `sum()` method, which accepts the name of the field to sum over as its argument). Note that field names passed in in this manner are automatically translated to the names used on the database (set using the `name` keyword argument to a field constructor).
Keyword arguments to `exec_js()` are combined into an object called `options`, which is available in the Javascript function. This may be used for defining specific parameters for your function.

Some variables are made available in the scope of the Javascript function:

- **collection** – the name of the collection that corresponds to the `Document` class that is being used; this should be used to get the `Collection` object from `db` in Javascript code
- **query** – the query that has been generated by the `QuerySet` object; this may be passed into the `find()` method on a `Collection` object in the Javascript function
- **options** – an object containing the keyword arguments passed into `exec_js()`

The following example demonstrates the intended usage of `exec_js()` by defining a function that sums over a field on a document (this functionality is already available through `sum()` but is shown here for sake of example):

```python
def sum_field(document, field_name, include_negatives=True):
    code = ""
    function(sumField) {
        var total = 0.0;
        db[collection].find(query).forEach(function(doc) {
            var val = doc[sumField];
            if (val >= 0.0 || options.includeNegatives) {
                total += val;
            }
        });
        return total;
    }
    options = {'includeNegatives': include_negatives}
    return document.objects.exec_js(code, field_name, **options)
```

As fields in MongoEngine may use different names in the database (set using the `db_field` keyword argument to a `Field` constructor), a mechanism exists for replacing MongoEngine field names with the database field names in Javascript code. When accessing a field on a collection object, use square-bracket notation, and prefix the MongoEngine field name with a tilde. The field name that follows the tilde will be translated to the name used in the database. Note that when referring to fields on embedded documents, the name of the `EmbeddedDocumentField`, followed by a dot, should be used before the name of the field on the embedded document. The following example shows how the substitutions are made:

```python
class Comment(EmbeddedDocument):
    content = StringField(db_field='body')

class BlogPost(Document):
    title = StringField(db_field='doctitle')
    comments = ListField(EmbeddedDocumentField(Comment), name='cs')

    # Returns a list of dictionaries. Each dictionary contains a value named
    # "document", which corresponds to the "title" field on a BlogPost, and
    # "comment", which corresponds to an individual comment. The substitutions
    # made are shown in the comments.
    BlogPost.objects.exec_js(""
    function() {
        var comments = [];
        db[collection].find(query).forEach(function(doc) {
            // doc[~comments] -> doc["cs"]
            var docComments = doc[~comments];
            for (var i = 0; i < docComments.length; i++) {
                // doc[~comments][i] -> doc["cs"][i]
                var comment = doc[~comments][i];
                comments.push({
                    // doc[~title] -> doc["doctitle"]
                    'document': doc[~title],
                    // doc[~comment] -> doc["cs"]
                    'comment': comment,
                });
            }
        });
        return comments;
    }
```

36 Chapter 4. Offline Reading
4.2.6 GridFS

New in version 0.4.

Writing

GridFS support comes in the form of the FileField field object. This field acts as a file-like object and provides a couple of different ways of inserting and retrieving data. Arbitrary metadata such as content type can also be stored alongside the files. In the following example, a document is created to store details about animals, including a photo:

```python
class Animal(Document):
    genus = StringField()
    family = StringField()
    photo = FileField()

marmot = Animal(genus='Marmota', family='Sciuridae')
marmot_photo = open('marmot.jpg', 'rb')
marmot.photo.put(marmot_photo, content_type = 'image/jpeg')
marmot.save()
```

Retrieval

So using the FileField is just like using any other field. The file can also be retrieved just as easily:

```python
marmot = Animal.objects(genus='Marmota').first()
photo = marmot.photo.read()
content_type = marmot.photo.content_type
```

Streaming

Streaming data into a FileField is achieved in a slightly different manner. First, a new file must be created by calling the new_file() method. Data can then be written using write():

```python
marmot.photo.new_file()
marmot.photo.write('some_image_data')
marmot.photo.write('some_more_image_data')
marmot.photo.close()
marmot.save()
```

Deletion

Deleting stored files is achieved with the delete() method:
marmot.photo.delete()

**Warning:** The FileField in a Document actually only stores the ID of a file in a separate GridFS collection. This means that deleting a document with a defined FileField does not actually delete the file. You must be careful to delete any files in a Document as above before deleting the Document itself.

### Replacing files

Files can be replaced with the `replace()` method. This works just like the `put()` method so even metadata can (and should) be replaced:

```python
another_marmot = open('another_marmot.png', 'rb')
marmot.photo.replace(another_marmot, content_type='image/png')
```

### 4.2.7 Signals

New in version 0.5.

**Note:** Signal support is provided by the excellent `blinker` library. If you wish to enable signal support this library must be installed, though it is not required for MongoEngine to function.

#### Overview

Signals are found within the `mongoengine.signals` module. Unless specified signals receive no additional arguments beyond the `sender` class and `document` instance. Post-signals are only called if there were no exceptions raised during the processing of their related function.

Available signals include:

- **pre_init** Called during the creation of a new `Document` or `EmbeddedDocument` instance, after the constructor arguments have been collected but before any additional processing has been done to them. (i.e. assignment of default values.) Handlers for this signal are passed the dictionary of arguments using the `values` keyword argument and may modify this dictionary prior to returning.

- **post_init** Called after all processing of a new `Document` or `EmbeddedDocument` instance has been completed.

- **pre_save** Called within `save()` prior to performing any actions.

- **pre_save_post_validation** Called within `save()` after validation has taken place but before saving.

- **post_save** Called within `save()` after all actions (validation, insert/update, cascades, clearing dirty flags) have completed successfully. Passed the additional boolean keyword argument `created` to indicate if the save was an insert or an update.

- **pre_delete** Called within `delete()` prior to attempting the delete operation.

- **post_delete** Called within `delete()` upon successful deletion of the record.

- **pre_bulk_insert** Called after validation of the documents to insert, but prior to any data being written. In this case, the `document` argument is replaced by a `documents` argument representing the list of documents being inserted.

- **post_bulk_insert** Called after a successful bulk insert operation. As per `pre_bulk_insert`, the `document` argument is omitted and replaced with a `documents` argument. An additional boolean argument, `loaded`, identifies the contents of `documents` as either `Document` instances when `True` or simply a list of primary key values for the inserted records if `False`. 

38 Chapter 4. Offline Reading
Attaching Events

After writing a handler function like the following:

```python
import logging
from datetime import datetime
from mongoengine import *
from mongoengine import signals
def update_modified(sender, document):
    document.modified = datetime.utcnow()
```

You attach the event handler to your `Document` or `EmbeddedDocument` subclass:

```python
class Record(Document):
    modified = DateTimeField()
signals.pre_save.connect(update_modified)
```

While this is not the most elaborate document model, it does demonstrate the concepts involved. As a more complete demonstration you can also define your handlers within your subclass:

```python
class Author(Document):
    name = StringField()

    @classmethod
def pre_save(cls, sender, document, **kwargs):
        logging.debug("Pre Save: %s" % document.name)

    @classmethod
def post_save(cls, sender, document, **kwargs):
        logging.debug("Post Save: %s" % document.name)
        if 'created' in kwargs:
            if kwargs['created']:
                logging.debug("Created")
            else:
                logging.debug("Updated")

signals.pre_save.connect(Author.pre_save, sender=Author)
signals.post_save.connect(Author.post_save, sender=Author)
```

Finally, you can also use this small decorator to quickly create a number of signals and attach them to your `Document` or `EmbeddedDocument` subclasses as class decorators:

```python
def handler(event):
    """Signal decorator to allow use of callback functions as class decorators."""
    def decorator(fn):
        def apply(cls):
            event.connect(fn, sender=cls)
            return cls

        fn.apply = apply
        return fn
    return decorator
```

Using the first example of updating a modification time the code is now much cleaner looking while still allowing manual execution of the callback:

```python
@handler(signals.pre_save)
def update_modified(sender, document):
    document.modified = datetime.utcnow()```
ReferenceFields and Signals

Currently reverse_delete_rules do not trigger signals on the other part of the relationship. If this is required you must manually handle the reverse deletion.

4.3 API Reference

4.3.1 Connecting

```python
def connect(db, alias='default', **kwargs):
    Connect to the database specified by the ‘db’ argument.
    Connection settings may be provided here as well if the database is not running on the default port on localhost. If authentication is needed, provide username and password arguments as well.
    Multiple databases are supported by using aliases. Provide a separate alias to connect to a different instance of mongod.
    Changed in version 0.6: - added multiple database support.
```

```python
def register_connection(alias, name, host=None, port=None, is_slave=False, read_preference=False, slaves=None, username=None, password=None, **kwargs):
    Add a connection.
```

Parameters:
- **alias** – the name that will be used to refer to this connection throughout MongoEngine
- **name** – the name of the specific database to use
- **host** – the host name of the mongod instance to connect to
- **port** – the port that the mongod instance is running on
- **is_slave** – whether the connection can act as a slave **Depreciated pymongo 2.0.1**
- **read_preference** – The read preference for the collection **Added pymongo 2.1**
- **slaves** – a list of aliases of slave connections; each of these must be a registered connection that has is_slave set to True
- **username** – username to authenticate with
- **password** – password to authenticate with
- **kwargs** – allow ad-hoc parameters to be passed into the pymongo driver

4.3.2 Documents

```python
class Document(*args, **values)
```

The base class used for defining the structure and properties of collections of documents stored in MongoDB. Inherit from this class, and add fields as class attributes to define a document’s structure. Individual documents may then be created by making instances of the Document subclass.

By default, the MongoDB collection used to store documents created using a Document subclass will be the name of the subclass converted to lowercase. A different collection may be specified by providing collection to the meta dictionary in the class definition.
A Document subclass may be itself subclassed, to create a specialised version of the document that will be stored in the same collection. To facilitate this behaviour a _cls field is added to documents (hidden though the MongoEngine interface). To disable this behaviour and remove the dependence on the presence of _cls set allow_inheritance to False in the meta dictionary.

A Document may use a Capped Collection by specifying max_documents and max_size in the meta dictionary. max_documents is the maximum number of documents that is allowed to be stored in the collection, and max_size is the maximum size of the collection in bytes. If max_size is not specified and max_documents is, max_size defaults to 10000000 bytes (10MB).

Indexes may be created by specifying indexes in the meta dictionary. The value should be a list of field names or tuples of field names. Index direction may be specified by prefixing the field names with a + or - sign.

Automatic index creation can be disabled by specifying attr: auto_create_index in the meta dictionary. If this is set to False then indexes will not be created by MongoEngine. This is useful in production systems where index creation is performed as part of a deployment system.

By default, _cls will be added to the start of every index (that doesn’t contain a list) if allow_inheritance is True. This can be disabled by either setting cls to False on the specific index or by setting index_cls to False on the meta dictionary for the document.

Initialise a document or embedded document

Parameters

- __auto_convert – Try and cast python objects to Object types
- values – A dictionary of values for the document

objects

A QuerySet object that is created lazily on access.

cascade_save(*args, **kwargs)

Recursively saves any references / generic references on an objects

classmethod compare_indexes()

Compares the indexes defined in MongoEngine with the ones existing in the database. Returns any missing/extra indexes.

delete(**write_concern)

Delete the Document from the database. This will only take effect if the document has been previously saved.

Parameters write_concern – Extra keyword arguments are passed down which will be used as options for the resultant getLastError command. For example, save(..., write_concern={w: 2, fsync: True}, ...) will wait until at least two servers have recorded the write and will force an fsync on the primary server.

classmethod drop_collection()

Drops the entire collection associated with this Document type from the database.

classmethod ensure_index(key_or_list, drop_dups=False, background=False, **kwargs)

Ensure that the given indexes are in place.

Parameters key_or_list – a single index key or a list of index keys (to construct a multi-field index); keys may be prefixed with a + or a - to determine the index ordering

classmethod ensure_indexes()

Checks the document meta data and ensures all the indexes exist.

Global defaults can be set in the meta - see Defining documents

Note: You can disable automatic index creation by setting auto_create_index to False in the documents meta data
**classmethod list_indexes**(go_up=True, go_down=True)

Lists all of the indexes that should be created for given collection. It includes all the indexes from super- and sub-classes.

**my_metaclass**

alias of TopLevelDocumentMetaclass

**classmethod register_delete_rule**(document_cls, field_name, rule)

This method registers the delete rules to apply when removing this object.

**reload**(max_depth=1)

Reloads all attributes from the database.

New in version 0.1.2.

Changed in version 0.6: Now chainable

**save**(force_insert=False, validate=True, clean=True, write_concern=None, cascade=None, cascade_kwargs=None, _refs=None, **kwargs)

Save the Document to the database. If the document already exists, it will be updated, otherwise it will be created.

**Parameters**

- **force_insert** – only try to create a new document, don’t allow updates of existing documents
- **validate** – validates the document; set to False to skip.
- **clean** – call the document clean method, requires validate to be True.
- **write_concern** – Extra keyword arguments are passed down to save() OR insert() which will be used as options for the resultant getLastError command. For example, save(..., write_concern={w: 2, fsync: True}, ...) will wait until at least two servers have recorded the write and will force an fsync on the primary server.
- **cascade** – Sets the flag for cascading saves. You can set a default by setting “cascade” in the document __meta__
- **cascade_kwargs** – (optional) kwargs dictionary to be passed throw to cascading saves. Implies cascade=True.
- **_refs** – A list of processed references used in cascading saves

Changed in version 0.5: In existing documents it only saves changed fields using set / unset. Saves are cascaded and any DBRef objects that have changes are saved as well.

Changed in version 0.6: Added cascading saves

Changed in version 0.8: Cascade saves are optional and default to False. If you want fine grain control then you can turn off using document meta[‘cascade’] = True. Also you can pass different kwargs to the cascade save using cascade_kwargs which overwrites the existing kwargs with custom values.

**select_related**(max_depth=1)

Handles dereferencing of DBRef objects to a maximum depth in order to cut down the number queries to mongodb.

New in version 0.5.

**switch_collection**(collection_name)

Temporarily switch the collection for a document instance.

Only really useful for archiving off data and calling save():

```python
user = User.objects.get(id=user_id)
user.switch_collection('old-users')
user.save()
```

If you need to read from another database see **switch_db**
Parameters `collection_name` – The database alias to use for saving the document

```python
switch_db(db_alias)
```
Temporarily switch the database for a document instance.

Only really useful for archiving off data and calling `save()`:

```python
user = User.objects.get(id=user_id)
user.switch_db('archive-db')
user.save()
```

If you need to read from another database see `switch_db`

Parameters `db_alias` – The database alias to use for saving the document

```python
to_dbref()
```
Returns an instance of `DBRef` useful in `__raw__` queries.

```python
update(**kwargs)
```

Raises `OperationError` if called on an object that has not yet been saved.

class `mongoengine.EmbeddedDocument`(*args, **kwargs)
A Document that isn’t stored in its own collection. `EmbeddedDocuments` should be used as fields on `Document` through the `EmbeddedDocumentField` field type.

A `EmbeddedDocument` subclass may be itself subclassed, to create a specialised version of the embedded document that will be stored in the same collection. To facilitate this behaviour a `_cls` field is added to documents (hidden though the MongoEngine interface). To disable this behaviour and remove the dependence on the presence of `_cls` set `allow_inheritance` to `False` in the `meta` dictionary.

```python
my_metaclass
```
alias of `DocumentMetaclass`

class `mongoengine.DynamicDocument`(*args, **values)
A Dynamic Document class allowing flexible, expandable and uncontrolled schemas. As a `Document` subclass, acts in the same way as an ordinary document but has expando style properties. Any data passed or set against the `DynamicDocument` that is not a field is automatically converted into a `DynamicField` and data can be attributed to that field.

**Note:** There is one caveat on Dynamic Documents: fields cannot start with `_`

Initialise a document or embedded document

Parameters

- `__auto_convert` – Try and will cast python objects to Object types
- `values` – A dictionary of values for the document

```python
my_metaclass
```
alias of `TopLevelDocumentMetaclass`

class `mongoengine.DynamicEmbeddedDocument`(*args, **kwargs)
A Dynamic Embedded Document class allowing flexible, expandable and uncontrolled schemas. See `DynamicDocument` for more information about dynamic documents.

```python
my_metaclass
```
alias of `DocumentMetaclass`

class `mongoengine.document.MapReduceDocument`(*document, **collection, **key, **value)
A document returned from a map/reduce query.

Parameters

- `collection` – An instance of `Collection`
• **key** – Document/result key, often an instance of `ObjectId`. If supplied as an `ObjectId` found in the given `collection`, the object can be accessed via the `object` property.

• **value** – The result(s) for this key.

New in version 0.3.

**object**

Lazy-load the object referenced by `self.key`. `self.key` should be the `primary_key`.

**class** `mongoengine`:` ValidationError (message='', **kwargs)`

Validation exception.

May represent an error validating a field or a document containing fields with validation errors.

**Variables** `errors` – A dictionary of errors for fields within this document or list, or None if the error is for an individual field.

**to_dict()**

Returns a dictionary of all errors within a document

Keys are field names or list indices and values are the validation error messages, or a nested dictionary of errors for an embedded document or list.

4.3.3 Context Managers

**class** `mongoengine.context_managers.switch_db (cls, db_alias)`

`switch_db` alias context manager.

Example

```python
# Register connections
register_connection('default', 'mongoenginetest')
register_connection('testdb-1', 'mongoenginetest2')

class Group(Document):
    name = StringField()

Group(name="test").save()  # Saves in the default db

with switch_db(Group, 'testdb-1') as Group:
    Group(name="hello testdb!").save()  # Saves in testdb-1
```

Construct the `switch_db` context manager

**Parameters**

• **cls** – the class to change the registered db

• **db_alias** – the name of the specific database to use

**class** `mongoengine.context_managers.switch_collection (cls, collection_name)`

`switch_collection` alias context manager.

Example

```python
class Group(Document):
    name = StringField()

Group(name="test").save()  # Saves in the default db

with switch_collection(Group, 'group1') as Group:
    Group(name="hello testdb!").save()  # Saves in group1 collection
```

Construct the `switch_collection` context manager
Parameters

- `cls` – the class to change the registered db
- `collection_name` – the name of the collection to use

```python
class mongoengine.context_managers.no_dereference(cls)

no_dereference context manager.

Turns off all dereferencing in Documents for the duration of the context manager:
```
with no_dereference(Group) as Group:
    Group.objects.find()
```

Construct the no_dereference context manager.

Parameters `cls` – the class to turn dereferencing off on

```python
class mongoengine.context_managers.query_counter

Query_counter context manager to get the number of queries.

Construct the query_counter.
```

4.3.4 Querying

```python
class mongoengine.queryset.QuerySet(document, collection)

The default queryset, that builds queries and handles a set of results returned from a query.

Wraps a MongoDB cursor, providing `Document` objects as the results.
```

```python
__call__(q_obj=None, class_check=True, slave_okay=False, read_preference=None, **query)

Filter the selected documents by calling the `QuerySet` with a query.
```

Parameters

- `q_obj` – a `Q` object to be used in the query; the `QuerySet` is filtered multiple times with different `Q` objects, only the last one will be used
- `class_check` – If set to False bypass class name check when querying collection
- `slave_okay` – if True, allows this query to be run against a replica secondary.
- `query` – Django-style query keyword arguments

Params read_preference if set, overrides connection-level read_preference from `ReplicaSetConnection`.

```python
all()

Returns all documents.
```

```python
all_fields()

Include all fields. Reset all previously calls of .only() or .exclude().
```

```python
post = BlogPost.objects.exclude("comments").all_fields()
```

New in version 0.5.

```python
as_pymongo(coerce_types=False)

Instead of returning Document instances, return raw values from pymongo.
```

Parameters coerce_type – Field types (if applicable) would be use to coerce types.

```python
average(field)

Average over the values of the specified field.
```

Parameters field – the field to average over; use dot-notation to refer to embedded document fields

Changed in version 0.5: - updated to map_reduce as db.eval doesn't work with sharding.
**clone()**

Creates a copy of the current QuerySet

New in version 0.5.

**clone_into(cls)**

Creates a copy of the current BaseQuerySet into another child class

**count(with_limit_and_skip=True)**

Count the selected elements in the query.

**Parameters (optional)** (with_limit_and_skip) – take any limit() or skip() that has been applied to this cursor into account when getting the count

**create(**kwargs**)**

Create new object. Returns the saved object instance.

New in version 0.4.

**delete(write_concern=None, _from_doc_delete=False)**

Delete the documents matched by the query.

**Parameters**

- **write_concern** – Extra keyword arguments are passed down which will be used as options for the resultant getLastError command. For example, save(..., write_concern={w: 2, fsync: True}, ...) will wait until at least two servers have recorded the write and will force an fsync on the primary server.

- **_from_doc_delete** – True when called from document delete therefore signals will have been triggered so don’t loop.

**distinct(field)**

Return a list of distinct values for a given field.

**Parameters** **field** – the field to select distinct values from

**Note:** This is a command and won’t take ordering or limit into account.

New in version 0.4.

Changed in version 0.5: - Fixed handling references

Changed in version 0.6: - Improved db_field reference handling

**ensure_index(**kwargs**)**

Deprecated use Document.ensure_index()

**exclude(**fields**)**

Opposite to .only(), exclude some document’s fields.

```python
post = BlogPost.objects(...).exclude("comments")
```

**Note:** exclude() is chainable and will perform a union :: So with the following it will exclude both: title and author.name:

```python
post = BlogPost.objects.exclude("title").exclude("author.name")
```

**all_fields()** will reset any field filters.

**Parameters** **fields** – fields to exclude

New in version 0.5.

**exec_js(code, **fields, **options)**

Execute a Javascript function on the server. A list of fields may be provided, which will be translated to their correct names and supplied as the arguments to the function. A few extra variables are added
to the function’s scope: `collection`, which is the name of the collection in use; `query`, which is an object representing the current query; and `options`, which is an object containing any options specified as keyword arguments.

As fields in MongoEngine may use different names in the database (set using the `db_field` keyword argument to a `Field` constructor), a mechanism exists for replacing MongoEngine field names with the database field names in Javascript code. When accessing a field, use square-bracket notation, and prefix the MongoEngine field name with a tilde (~).

**Parameters**

- `code` – a string of Javascript code to execute
- `fields` – fields that you will be using in your function, which will be passed in to your function as arguments
- `options` – options that you want available to the function (accessed in Javascript through the `options` object)

**explain**

Return an explain plan record for the `QuerySet`'s cursor.

**Parameters**

- `format` – format the plan before returning it

**fields**

Manipulate how you load this document’s fields. Used by `.only()` and `.exclude()` to manipulate which fields to retrieve. Fields also allows for a greater level of control for example:

Retrieving a Subrange of Array Elements:

You can use the `$slice` operator to retrieve a subrange of elements in an array. For example to get the first 5 comments:

```javascript
post = BlogPost.objects(...).fields(slice__comments=5)
```

**Parameters**

- `kwargs` – A dictionary identifying what to include

New in version 0.5.

**filter**

An alias of `__call__()`

**first**

Retrieve the first object matching the query.

**from_json**

Converts json data to unsaved objects

**get**

Retrieve the matching object raising `MultipleObjectsReturned` or `DocumentName.MultipleObjectsReturned` exception if multiple results and `DoesNotExist` or `DocumentName.DoesNotExist` if no results are found.

New in version 0.3.

**get_or_create**

Retrieve unique object or create, if it doesn’t exist. Returns a tuple of (object, created), where object is the retrieved or created object and created is a boolean specifying whether a new object was created. Raises `MultipleObjectsReturned` or `DocumentName.MultipleObjectsReturned` if multiple results are found. A new document will be created if the document doesn’t exists; a dictionary of default values for the new document may be provided as a keyword argument called `defaults`.

**Note:** This requires two separate operations and therefore a race condition exists. Because there are no transactions in MongoDB other approaches should be investigated, to ensure you don’t accidently duplicate data when using this method. This is now scheduled to be removed before 1.0
Parameters

- **write_concern** – optional extra keyword arguments used if we have to create a new document. Passes any write_concern onto `save()`.
- **auto_save** – if the object is to be saved automatically if not found.

Deprecated since version 0.8.

Changed in version 0.6: - added `auto_save`

New in version 0.3.

**hint**(index=None)

Added ‘hint’ support, telling Mongo the proper index to use for the query.

Judicious use of hints can greatly improve query performance. When doing a query on multiple fields (at least one of which is indexed) pass the indexed field as a hint to the query.

Hinting will not do anything if the corresponding index does not exist. The last hint applied to this cursor takes precedence over all others.

New in version 0.5.

**in_bulk**(object_ids)

Retrieve a set of documents by their ids.

Parameters **object_ids** – a list or tuple of `ObjectId`

Return type dict of ObjectIds as keys and collection-specific Document subclasses as values.

New in version 0.3.

**insert**(docs_or_docs, load_bulk=True, write_concern=None)

bulk insert documents

Parameters

- **docs_or_doc** – a document or list of documents to be inserted
- **(optional)** *(load_bulk)* – If True returns the list of document instances
- **write_concern** – Extra keyword arguments are passed down to `insert()` which will be used as options for the resultant `getLastError` command. For example, `insert(...., (w: 2, fsync: True))` will wait until at least two servers have recorded the write and will force an fsync on each server being written to.

By default returns document instances, set `load_bulk` to False to return just `ObjectId`

New in version 0.5.

**item_frequencies**(field, normalize=False, map_reduce=True)

Returns a dictionary of all items present in a field across the whole queried set of documents, and their corresponding frequency. This is useful for generating tag clouds, or searching documents.

**Note:** Can only do direct simple mappings and cannot map across `ReferenceField` or `GenericReferenceField` for more complex counting a manual map reduce call would be required.

If the field is a `ListField`, the items within each list will be counted individually.

Parameters

- **field** – the field to use
- **normalize** – normalize the results so they add to 1.0
- **map_reduce** – Use map_reduce over exec_js
**limit** *(n)*
Limit the number of returned documents to *n*. This may also be achieved using array-slicing syntax (e.g. `User.objects[:5]).

**Parameters**
- **n** – the maximum number of objects to return

**map_reduce** *(map_f, reduce_f, output, finalize_f=None, limit=None, scope=None)*
Perform a map/reduce query using the current query spec and ordering. While `map_reduce` respects `QuerySet` chaining, it must be the last call made, as it does not return a mutable `QuerySet`.

See the `test_map_reduce()` and `test_map_advanced()` tests in `tests.queryset.QuerySetTest` for usage examples.

**Parameters**
- **map_f** – map function, as `Code` or string
- **reduce_f** – reduce function, as `Code` or string
- **output** – output collection name, if set to ‘inline’ will try to use `inline_map_reduce` This can also be a dictionary containing output options see: [http://docs.mongodb.org/manual/reference/command/mapReduce/#dbcmd.mapReduce](http://docs.mongodb.org/manual/reference/command/mapReduce/#dbcmd.mapReduce)
- **finalize_f** – finalize function, an optional function that performs any post-reduction processing.
- **scope** – values to insert into map/reduce global scope. Optional.
- **limit** – number of objects from current query to provide to map/reduce method

Returns an iterator yielding `MapReduceDocument`.

**Note:** Map/Reduce changed in server version >= 1.7.4. The PyMongo `map_reduce()` helper requires PyMongo version >= 1.11.

Changed in version 0.5: - removed `keep_temp` keyword argument, which was only relevant for MongoDB server versions older than 1.7.4

New in version 0.3.

**next**()
Wrap the result in a `Document` object.

**no_cache**()
Convert to a non_caching queryset
New in version 0.8.3: Convert to non caching queryset

**no_dereference**()
Turn off any dereferencing for the results of this queryset.

**no_sub_classes**()
Only return instances of this document and not any inherited documents

**none**()
Helper that just returns a list

**only**(*fields*)
Load only a subset of this document’s fields.

```python
post = BlogPost.objects(...).only("title", "author.name")
```

**Note:** `only()` is chainable and will perform a union :: So with the following it will fetch both: `title` and `author.name`.
post = BlogPost.objects.only("title").only("author.name")

```
all_fields() will reset any field filters.

Parameters fields – fields to include
New in version 0.3.
Changed in version 0.5: - Added subfield support
```

```
order_by(*keys)
Order the QuerySet by the keys. The order may be specified by prepending each of the keys by a + or a -. Ascending order is assumed.

Parameters keys – fields to order the query results by; keys may be prefixed with + or - to determine the ordering direction
```

```
read_preference(read_preference)
Change the read_preference when querying.

Parameters read_preference – override ReplicaSetConnection-level preference.
```

```
rewind()
Rewind the cursor to its unevaluated state.
New in version 0.3.
```

```
scalar(*fields)
Instead of returning Document instances, return either a specific value or a tuple of values in order.
Can be used along with no_dereference() to turn off dereferencing.

Note: This effects all results and can be unset by calling scalar without arguments. Calls only automatically.

Parameters fields – One or more fields to return instead of a Document.
```

```
select_related(max_depth=1)
Handles dereferencing of DBRef objects or ObjectId a maximum depth in order to cut down the number queries to mongodb.
New in version 0.5.
```

```
skip(n)
Skip n documents before returning the results. This may also be achieved using array-slicing syntax (e.g. User.objects[5:]).

Parameters n – the number of objects to skip before returning results
```

```
slave_okay(enabled)
Enable or disable the slave_okay when querying.

Parameters enabled – whether or not the slave_okay is enabled
```

```
snapshot(enabled)
Enable or disable snapshot mode when querying.

Parameters enabled – whether or not snapshot mode is enabled
```

```
sum(field)
Sum over the values of the specified field.

Parameters field – the field to sum over; use dot-notation to refer to embedded document fields
```
timeout (enabled)
Enable or disable the default mongod timeout when querying.

Parameters enabled – whether or not the timeout is used

.. versionchanged:: 0.5 - made chainable

to_json (*args, **kwargs)
Converts a queryset to JSON

update (upsert=False, multi=True, write_concern=None, full_result=False, **update)
Perform an atomic update on the fields matched by the query.

Parameters

• upsert – Any existing document with that “_id” is overwritten.
• multi – Update multiple documents.
• write_concern – Extra keyword arguments are passed down which will be used as options for the resultant getLastError command. For example, save(..., write_concern={w: 2, fsync: True}, ...) will wait until at least two servers have recorded the write and will force an fsync on the primary server.
• full_result – Return the full result rather than just the number updated.
• update – Django-style update keyword arguments

New in version 0.2.

update_one (upsert=False, write_concern=None, **update)
Perform an atomic update on first field matched by the query.

Parameters

• upsert – Any existing document with that “_id” is overwritten.
• write_concern – Extra keyword arguments are passed down which will be used as options for the resultant getLastError command. For example, save(..., write_concern={w: 2, fsync: True}, ...) will wait until at least two servers have recorded the write and will force an fsync on the primary server.
• update – Django-style update keyword arguments

New in version 0.2.

values_list (*fields)
An alias for scalar

where (where_clause)
Filter QuerySet results with a $where clause (a Javascript expression). Performs automatic field name substitution like mongoengine.queryset.Queryset.exec_js().

Note: When using this mode of query, the database will call your function, or evaluate your predicate clause, for each object in the collection.

New in version 0.5.

with_id (object_id)
Retrieve the object matching the id provided. Uses object_id only and raises InvalidQueryError if a filter has been applied. Returns None if no document exists with that id.

Parameters object_id – the value for the id of the document to look up

Changed in version 0.6: Raises InvalidQueryError if filter has been set

class mongoengine.queryset.QuerySetNoCache (document, collection)
A non caching QuerySet

4.3. API Reference
__call__(q_obj=None, class_check=True, slave_okay=False, read_preference=None, **query)

Filter the selected documents by calling the QuerySet with a query.

Parameters

- **q_obj** – a Q object to be used in the query; the QuerySet is filtered multiple times with different Q objects, only the last one will be used.
- **class_check** – If set to False bypass class name check when querying collection
- **slave_okay** – if True, allows this query to be run against a replica secondary.
- **query** – Django-style query keyword arguments

Params read_preference if set, overrides connection-level read_preference from ReplicaSetConnection.

cache()

Convert to a caching queryset

New in version 0.8.3: Convert to caching queryset

mongoengine.queryset.queryset_manager(func)

Decorator that allows you to define custom QuerySet managers on Document classes. The manager must be a function that accepts a Document class as its first argument, and a QuerySet as its second argument. The method function should return a QuerySet, probably the same one that was passed in, but modified in some way.

4.3.5 Fields

class mongoengine.base.fields.BaseField(db_field=None, name=None, required=False, default=None, unique=False, unique_with=None, primary_key=False, validation=None, choices=None, verbose_name=None, help_text=None)

A base class for fields in a MongoDB document. Instances of this class may be added to subclasses of Document to define a document’s schema.

Changed in version 0.5: - added verbose and help text

Parameters

- **db_field** – The database field to store this field in (defaults to the name of the field)
- **name** – Deprecated - use db_field
- **required** – If the field is required. Whether it has to have a value or not. Defaults to False.
- **default** – (optional) The default value for this field if no value has been set (or if the value has been unset). It Can be a callable.
- **unique** – Is the field value unique or not. Defaults to False.
- **unique_with** – (optional) The other field this field should be unique with.
- **primary_key** – Mark this field as the primary key. Defaults to False.
- **validation** – (optional) A callable to validate the value of the field. Generally this is deprecated in favour of the FIELD.validate method
- **choices** – (optional) The valid choices
- **verbose_name** – (optional) The verbose name for the field. Designed to be human readable and is often used when generating model forms from the document model.
- **help_text** – (optional) The help text for this field and is often used when generating model forms from the document model.
class mongoengine.fields.StringField(regex=None, max_length=None, min_length=None, **kwargs)
A unicode string field.

class mongoengine.fields.URLField(verify_exists=False, url_regex=None, **kwargs)
A field that validates input as an URL.
New in version 0.3.

class mongoengine.fields.EmailField(regex=None, max_length=None, min_length=None, **kwargs)
A field that validates input as an E-Mail-Address.
New in version 0.4.

class mongoengine.fields.IntegerField(min_value=None, max_value=None, **kwargs)
An 32-bit integer field.

class mongoengine.fields.LongField(min_value=None, max_value=None, **kwargs)
An 64-bit integer field.

class mongoengine.fields.FloatField(min_value=None, max_value=None, **kwargs)
An floating point number field.

class mongoengine.fields.DecimalField(min_value=None, max_value=None, force_string=False, precision=2, rounding='ROUND_HALF_UP', **kwargs)
A fixed-point decimal number field.
Changed in version 0.8.
New in version 0.3.

Parameters

• min_value – Validation rule for the minimum acceptable value.
• max_value – Validation rule for the maximum acceptable value.
• force_string – Store as a string.
• precision – Number of decimal places to store.
• rounding – The rounding rule from the python decimal libary:
  – decimal.ROUND_CEILING (towards Infinity)
  – decimal.ROUND_DOWN (towards zero)
  – decimal.ROUND_FLOOR (towards -Infinity)
  – decimal.ROUND_HALF_DOWN (to nearest with ties going towards zero)
  – decimal.ROUND_HALF_EVEN (to nearest with ties going to nearest even integer)
  – decimal.ROUND_HALF_UP (to nearest with ties going away from zero)
  – decimal.ROUND_UP (away from zero)
  – decimal.ROUND_05UP (away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise towards zero)

Defaults to: decimal.ROUND_HALF_UP

class mongoengine.fields.BooleanField(dbi_field=None, name=None, required=False, default=None, unique=False, unique_with=None, primary_key=False, validation=None, choices=None, verbose_name=None, help_text=None)
A boolean field type.
New in version 0.1.2.

Parameters
• **db_field** – The database field to store this field in (defaults to the name of the field)

• **name** – Deprecated - use db_field

• **required** – If the field is required. Whether it has to have a value or not. Defaults to False.

• **default** – (optional) The default value for this field if no value has been set (or if the value has been unset). It Can be a callable.

• **unique** – Is the field value unique or not. Defaults to False.

• **unique_with** – (optional) The other field this field should be unique with.

• **primary_key** – Mark this field as the primary key. Defaults to False.

• **validation** – (optional) A callable to validate the value of the field. Generally this is deprecated in favour of the FIELD.validate method

• **choices** – (optional) The valid choices

• **verbose_name** – (optional) The verbose name for the field. Designed to be human readable and is often used when generating model forms from the document model.

• **help_text** – (optional) The help text for this field and is often used when generating model forms from the document model.

```python
class mongoengine.fields.DateTimeField(
   db_field=None, name=None, required=False,
   default=None, unique=False, unique_with=None,
   primary_key=False, validation=None,
   choices=None, verbose_name=None,
   help_text=None)
```

A datetime field.

Uses the python-dateutil library if available alternatively use time.strptime to parse the dates. Note: python-dateutil’s parser is fully featured and when installed you can utilise it to convert varying types of date formats into valid python datetime objects.

**Note:** Microseconds are rounded to the nearest millisecond. Pre UTC microsecond support is effectively broken. Use **ComplexDateTimeField** if you need accurate microsecond support.

**Parameters**

• **db_field** – The database field to store this field in (defaults to the name of the field)

• **name** – Deprecated - use db_field

• **required** – If the field is required. Whether it has to have a value or not. Defaults to False.

• **default** – (optional) The default value for this field if no value has been set (or if the value has been unset). It Can be a callable.

• **unique** – Is the field value unique or not. Defaults to False.

• **unique_with** – (optional) The other field this field should be unique with.

• **primary_key** – Mark this field as the primary key. Defaults to False.

• **validation** – (optional) A callable to validate the value of the field. Generally this is deprecated in favour of the FIELD.validate method

• **choices** – (optional) The valid choices

• **verbose_name** – (optional) The verbose name for the field. Designed to be human readable and is often used when generating model forms from the document model.

• **help_text** – (optional) The help text for this field and is often used when generating model forms from the document model.
class mongoengine.fields.ComplexDateTimeField (separator=',', **kwargs)
ComplexDateTimeField handles microseconds exactly instead of rounding like DateTimeField does.

Derives from a StringField so you can do gte and lte filtering by using lexicographical comparison when filtering / sorting strings.

The stored string has the following format:

YYYY,MM,DD,HH,MM,SS,NNNNNN

Where NNNNNN is the number of microseconds of the represented datetime. The , as the separator can be easily modified by passing the separator keyword when initializing the field.

New in version 0.5.

class mongoengine.fields.EmbeddedDocumentField (document_type, **kwargs)
An embedded document field - with a declared document_type. Only valid values are subclasses of EmbeddedDocument.

class mongoengine.fields.GenericEmbeddedDocumentField (db_field=None, name=None, required=False, default=None, unique=False, unique_with=None, primary_key=False, validation=None, choices=None, verbose_name=None, help_text=None)
A generic embedded document field - allows any EmbeddedDocument to be stored.
Only valid values are subclasses of EmbeddedDocument.

Note: You can use the choices param to limit the acceptable EmbeddedDocument types

Parameters

- **db_field** – The database field to store this field in (defaults to the name of the field)
- **name** – Deprecated - use db_field
- **required** – If the field is required. Whether it has to have a value or not. Defaults to False.
- **default** – (optional) The default value for this field if no value has been set (or if the value has been unset). It Can be a callable.
- **unique** – Is the field value unique or not. Defaults to False.
- **unique_with** – (optional) The other field this field should be unique with.
- **primary_key** – Mark this field as the primary key. Defaults to False.
- **validation** – (optional) A callable to validate the value of the field. Generally this is deprecated in favour of the FIELD.validate method
- **choices** – (optional) The valid choices
- **verbose_name** – (optional) The verbose name for the field. Designed to be human readable and is often used when generating model forms from the document model.
- **help_text** – (optional) The help text for this field and is often used when generating model forms from the document model.

class mongoengine.fields.DynamicField (db_field=None, name=None, required=False, default=None, unique=False, unique_with=None, primary_key=False, validation=None, choices=None, verbose_name=None, help_text=None)
A truly dynamic field type capable of handling different and varying types of data.
Used by `DynamicDocument` to handle dynamic data

**Parameters**

- **`db_field`** – The database field to store this field in (defaults to the name of the field)
- **`name`** – Deprecated - use `db_field`
- **`required`** – If the field is required. Whether it has to have a value or not. Defaults to False.
- **`default`** – (optional) The default value for this field if no value has been set (or if the value has been unset). It can be a callable.
- **`unique`** – Is the field value unique or not. Defaults to False.
- **`unique_with`** – (optional) The other field this field should be unique with.
- **`primary_key`** – Mark this field as the primary key. Defaults to False.
- **`validation`** – (optional) A callable to validate the value of the field. Generally this is deprecated in favour of the `FIELD.validate` method
- **`choices`** – (optional) The valid choices
- **`verbose_name`** – (optional) The verbose name for the field. Designed to be human readable and is often used when generating model forms from the document model.
- **`help_text`** – (optional) The help text for this field and is often used when generating model forms from the document model.

```python
class mongoengine.fields.ListField(field=None, **kwargs)
```

A list field that wraps a standard field, allowing multiple instances of the field to be used as a list in the database.

If using with ReferenceFields see: *One to Many with ListFields*

**Note:** Required means it cannot be empty - as the default for ListFields is `[]`

```python
class mongoengine.fields.SortedListField(field, **kwargs)
```

A ListField that sorts the contents of its list before writing to the database in order to ensure that a sorted list is always retrieved.

**Warning:** There is a potential race condition when handling lists. If you set / save the whole list then other processes trying to save the whole list as well could overwrite changes. The safest way to append to a list is to perform a push operation.

New in version 0.4.

Changed in version 0.6: - added reverse keyword

```python
class mongoengine.fields.DictField(basecls=None, field=None, *args, **kwargs)
```

A dictionary field that wraps a standard Python dictionary. This is similar to an embedded document, but the structure is not defined.

**Note:** Required means it cannot be empty - as the default for ListFields is `[]`

New in version 0.3.

Changed in version 0.5: - Can now handle complex / varying types of data

```python
class mongoengine.fields.MapField(field=None, *args, **kwargs)
```

A field that maps a name to a specified field type. Similar to a DictField, except the ‘value’ of each item must match the specified field type.

New in version 0.5.
class mongoengine.fields.ReferenceField(document_type, dbref=False, reverse_delete_rule=0, **kwargs)

A reference to a document that will be automatically dereferenced on access (lazily).

Use the reverse_delete_rule to handle what should happen if the document the field is referencing is deleted. EmbeddedDocuments, DictFields and MapFields do not support reverse_delete_rules and an InvalidDocumentError will be raised if trying to set on one of these Document / Field types.

The options are:

• DO NOTHING - don’t do anything (default).
• NULLIFY - Updates the reference to null.
• CASCADE - Deletes the documents associated with the reference.
• DENY - Prevent the deletion of the reference object.
• PULL - Pull the reference from a ListField of references

Alternative syntax for registering delete rules (useful when implementing bi-directional delete rules)

```python
class Bar(Document):
    content = StringField()
    foo = ReferenceField('Foo')
Bar.register_delete_rule(Foo, 'bar', NULLIFY)
```

Note: reverse_delete_rules do not trigger pre / post delete signals to be triggered.

Changed in version 0.5: added reverse_delete_rule

Initialises the Reference Field.

Parameters

• dbref – Store the reference as DBRef or as the ObjectId
• reverse_delete_rule – Determines what to do when the referring object is deleted

class mongoengine.fields.GenericReferenceField(db_field=None, name=None, required=False, default=None, unique=False, unique_with=None, primary_key=False, validation=None, choices=None, verbose_name=None, help_text=None)

A reference to any Document subclass that will be automatically dereferenced on access (lazily).

Note:

• Any documents used as a generic reference must be registered in the document registry. Importing the model will automatically register it.
• You can use the choices param to limit the acceptable Document types

New in version 0.3.

Parameters

• db_field – The database field to store this field in (defaults to the name of the field)
• name – Deprecated - use db_field
• required – If the field is required. Whether it has to have a value or not. Defaults to False.
• default – (optional) The default value for this field if no value has been set (or if the value has been unset). It Can be a callable.
• **unique** – Is the field value unique or not. Defaults to False.
• **unique_with** – (optional) The other field this field should be unique with.
• **primary_key** – Mark this field as the primary key. Defaults to False.
• **validation** – (optional) A callable to validate the value of the field. Generally this is deprecated in favour of the `FIELD.validate` method
• **choices** – (optional) The valid choices
• **verbose_name** – (optional) The verbose name for the field. Designed to be human readable and is often used when generating model forms from the document model.
• **help_text** – (optional) The help text for this field and is often used when generating model forms from the document model.

```python
class mongoengine.fields.BinaryField(max_bytes=None, **kwargs)
A binary data field.
```

```python
class mongoengine.fields.FileField(db_alias='default', collection_name='fs', **kwargs)
A GridFS storage field.
```

Changed in version 0.4.
Changed in version 0.5: added optional size param for read
Changed in version 0.6: added db_alias for multidb support

```python
class mongoengine.fields.ImageField(size=None, thumbnail_size=None, collection_name='images', **kwargs)
A Image File storage field.
```

@size (width, height, force): max size to store images, if larger will be automatically resized ex: size=(800, 600, True)

@thumbnail (width, height, force): size to generate a thumbnail
New in version 0.6.

```python
class mongoengine.fields.SequenceField(collection_name=None, db_alias=None, sequence_name=None, value_decorator=None, *args, **kwargs)
Provides a sequential counter see: http://www.mongodb.org/display/DOCS/Object+IDs#ObjectIDs-SequenceNumbers
```

**Note:** Although traditional databases often use increasing sequence numbers for primary keys. InMongoDB, the preferred approach is to use Object IDs instead. The concept is that in a very large cluster of machines, it is easier to create an object ID than have global, uniformly increasing sequence numbers.

Use any callable as `value_decorator` to transform calculated counter into any value suitable for your needs, e.g. string or hexadecimal representation of the default integer counter value.

New in version 0.5.
Changed in version 0.8: added `value_decorator`

```python
class mongoengine.fields.ObjectIdField(db_field=None, name=None, required=False, default=None, unique=False, unique_with=None, primary_key=False, validation=None, choices=None, verbose_name=None, help_text=None)
A field wrapper around MongoDB’s ObjectId.
```

**Parameters**

• **db_field** – The database field to store this field in (defaults to the name of the field)
• **name** – Deprecated - use db_field
• **required** – If the field is required. Whether it has to have a value or not. Defaults to False.

• **default** – (optional) The default value for this field if no value has been set (or if the value has been unset). It Can be a callable.

• **unique** – Is the field value unique or not. Defaults to False.

• **unique_with** – (optional) The other field this field should be unique with.

• **primary_key** – Mark this field as the primary key. Defaults to False.

• **validation** – (optional) A callable to validate the value of the field. Generally this is deprecated in favour of the `FIELD.validate` method

• **choices** – (optional) The valid choices

• **verbose_name** – (optional) The verbose name for the field. Designed to be human readable and is often used when generating model forms from the document model.

• **help_text** – (optional) The help text for this field and is often used when generating model forms from the document model.

```python
class mongoengine.fields.UUIDField(binary=True, **kwargs)
A UUID field.
New in version 0.6.
Store UUID data in the database

Parameters

• **binary** – if False store as a string.

Changed in version 0.8.0.
Changed in version 0.6.19.
```

```python
class mongoengine.fields.GeoPointField(db_field=None, name=None, required=False, default=None, unique=False, unique_with=None, primary_key=False, validation=None, choices=None, verbose_name=None, help_text=None)
A list storing a latitude and longitude.
New in version 0.4.

Parameters

• **db_field** – The database field to store this field in (defaults to the name of the field)

• **name** – Deprecated - use db_field

• **required** – If the field is required. Whether it has to have a value or not. Defaults to False.

• **default** – (optional) The default value for this field if no value has been set (or if the value has been unset). It Can be a callable.

• **unique** – Is the field value unique or not. Defaults to False.

• **unique_with** – (optional) The other field this field should be unique with.

• **primary_key** – Mark this field as the primary key. Defaults to False.

• **validation** – (optional) A callable to validate the value of the field. Generally this is deprecated in favour of the `FIELD.validate` method

• **choices** – (optional) The valid choices

• **verbose_name** – (optional) The verbose name for the field. Designed to be human readable and is often used when generating model forms from the document model.
```
• help_text – (optional) The help text for this field and is often used when generating model forms from the document model.

class mongoengine.fields.PointField(auto_index=True, *args, **kwargs)
A geo json field storing a latitude and longitude.

The data is represented as:

```json
{ "type" : "Point" ,
  "coordinates" : [x, y]}
```

You can either pass a dict with the full information or a list to set the value.

Requires mongodb >= 2.4 .. versionadded:: 0.8

Parameters

- auto_index – Automatically create a “2dsphere” index. Defaults to True.

class mongoengine.fields.LineStringField(auto_index=True, *args, **kwargs)
A geo json field storing a line of latitude and longitude coordinates.

The data is represented as:

```json
{ "type" : "LineString" ,
  "coordinates" : [[x1, y1], [x1, y1] ... [xn, yn]]}
```

You can either pass a dict with the full information or a list of points.

Requires mongodb >= 2.4 .. versionadded:: 0.8

Parameters

- auto_index – Automatically create a “2dsphere” index. Defaults to True.

class mongoengine.fields.PolygonField(auto_index=True, *args, **kwargs)
A geo json field storing a polygon of latitude and longitude coordinates.

The data is represented as:

```json
{ "type" : "Polygon" ,
  "coordinates" : [[[x1, y1], [x1, y1] ... [xn, yn]],
               [[x1, y1], [x1, y1] ... [xn, yn]]}
```

You can either pass a dict with the full information or a list of LineStrings. The first LineString being the outside and the rest being holes.

Requires mongodb >= 2.4 .. versionadded:: 0.8

Parameters

- auto_index – Automatically create a “2dsphere” index. Defaults to True.

class mongoengine.fields.GridFSBase

class mongoengine.fields.GridFSProxy (grid_id=None, key=None, instance=None, db_alias='default', collection_name='fs')
Proxy object to handle writing and reading of files to and from GridFS

New in version 0.4.

Changed in version 0.5: - added optional size param to read

Changed in version 0.6: - added collection name param

class mongoengine.fields.ImageGridFSProxy (grid_id=None, key=None, instance=None, db_alias='default', collection_name='fs')
Proxy for ImageField

versionadded: 0.6

class mongoengine.fields.ImproperlyConfigured
4.3.6 Misc

mongoengine.common._import_class(cls_name)

Cache mechanism for imports.

Due to complications of circular imports mongoengine needs to do lots of inline imports in functions. This is inefficient as classes are imported repeated throughout the mongoengine code. This is compounded by some recursive functions requiring inline imports.

mongoengine.common provides a single point to import all these classes. Circular imports aren’t an issue as it dynamically imports the class when first needed. Subsequent calls to the _import_class() can then directly retrieve the class from the mongoengine.common._class_registry_cache.

4.4 Changelog

4.4.1 Changes in 0.8.7

- Calling reload on deleted / nonexistant documents raises DoesNotExist (#538)
- Stop ensure_indexes running on a secondaries (#555)
- Fix circular import issue with django auth (#531) (#545)

4.4.2 Changes in 0.8.6

- Fix django auth import (#531)

4.4.3 Changes in 0.8.5

- Fix multi level nested fields getting marked as changed (#523)
- Django 1.6 login fix (#522) (#527)
- Django 1.6 session fix (#509)
- EmbeddedDocument._instance is now set when settng the attribute (#506)
- Fixed EmbeddedDocument with ReferenceField equality issue (#502)
- Fixed GenericReferenceField serialization order (#499)
- Fixed count and none bug (#498)
- Fixed bug with .only() and DictField with digit keys (#496)
- Added user_permissions to Django User object (#491, #492)
- Fix updating Geo Location fields (#488)
- Fix handling invalid dict field value (#485)
- Added app_label to MongoUser (#484)
- Use defaults when host and port are passed as None (#483)
- Fixed distinct casting issue with ListField of EmbeddedDocuments (#470)
- Fixed Django 1.6 sessions (#454, #480)
4.4.4 Changes in 0.8.4

- Remove database name necessity in uri connection schema (#452)
- Fixed “$pull” semantics for nested ListFields (#447)
- Allow fields to be named the same as query operators (#445)
- Updated field filter logic - can now exclude subclass fields (#443)
- Fixed dereference issue with embedded listfield referencefields (#439)
- Fixed slice when using inheritance causing fields to be excluded (#437)
- Fixed _get_db() attribute after a Document.switch_db() (#441)
- Dynamic Fields store and recompose Embedded Documents / Documents correctly (#449)
- Handle dynamic fieldnames that look like digits (#434)
- Added get_user_document and improve mongo_auth module (#423)
- Added str representation of GridFSProxy (#424)
- Update transform to handle docs erroneously passed to unset (#416)
- Fixed indexing - turn off __cls (#414)
- Fixed dereference threading issue in ComplexField.__get__() (#412)
- Fixed QuerySetNoCache.count() caching (#410)
- Don’t follow references in _get_changed_fields (#422, #417)
- Allow args and kwargs to be passed through to_json (#420)

4.4.5 Changes in 0.8.3

- Fixed EmbeddedDocuments with id also storing _id (#402)
- Added get_proxy_object helper to filefields (#391)
- Added QuerySetNoCache and QuerySet.no_cache() for lower memory consumption (#365)
- Fixed sum and average mapreduce dot notation support (#375, #376, #393)
- Fixed as_pymongo to return the id (#386)
- Document.select_related() now respects db_alias (#377)
- Reload uses shard_key if applicable (#384)
- Dynamic fields are ordered based on creation and stored in _fields_ordered (#396)

  **Potential breaking change:** [http://docs.mongoengine.org/en/latest/upgrade.html#to-0-8-3](http://docs.mongoengine.org/en/latest/upgrade.html#to-0-8-3)

- Fixed pickling dynamic documents _dynamic_fields (#387)
- Fixed ListField setslice and delslice dirty tracking (#390)
- Added Django 1.5 PY3 support (#392)
- Added match (SelemMatch) support for EmbeddedDocuments (#379)
- Fixed weakref being valid after reload (#374)
- Fixed queryset.get() respecting no_dereference (#373)
- Added full_result kwarg to update (#380)
4.4.6 Changes in 0.8.2

- Added compare_indexes helper (#361)
- Fixed cascading saves which weren’t turned off as planned (#291)
- Fixed Datastructures so instances are a Document or EmbeddedDocument (#363)
- Improved cascading saves write performance (#361)
- Fixed ambiguity and differing behaviour regarding field defaults (#349)
- ImageFields now include PIL error messages if invalid error (#353)
- Added lock when calling doc.Delete() for when signals have no sender (#350)
- Reload forces read preference to be PRIMARY (#355)
- Querysets are now least restrictive when querying duplicate fields (#332, #333)
- FileField now honouring db_alias (#341)
- Removed customised __set__ change tracking in ComplexBaseField (#344)
- Removed unused var in _get_changed_fields (#347)
- Added pre_save_post_validation signal (#345)
- DateTimeField now auto converts valid datetime isostrings into dates (#343)
- DateTimeField now uses dateutil for parsing if available (#343)
- Fixed Doc.objects(read_preference=X) not setting read preference (#352)
- Django session ttl index expiry fixed (#329)
- Fixed pickle.loads (#342)
- Documentation fixes

4.4.7 Changes in 0.8.1

- Fixed Python 2.6 django auth importlib issue (#326)
- Fixed pickle unsaved document regression (#327)

4.4.8 Changes in 0.8.0

- Fixed querying ReferenceField custom_id (#317)
- Fixed pickle issues with collections (#316)
- Added get_next_value preview for SequenceFields (#319)
- Added no_sub_classes context manager and queryset helper (#312)
- Querysets now utilise a local cache
- Changed __len__ behaviour in the queryset (#247, #311)
- Fixed querying string versions of ObjectIds issue with ReferenceField (#307)
- Added $setOnInsert support for upserts (#308)
- Upserts now possible with just query parameters (#309)
- Upserting is the only way to ensure docs are saved correctly (#306)
- Fixed register_delete_rule inheritance issue
- Fix cloning of sliced querysets (#303)
• Fixed update_one write concern (#302)
• Updated minimum requirement for pymongo to 2.5
• Add support for new geojson fields, indexes and queries (#299)
• If values cant be compared mark as changed (#287)
• Ensure as_pymongo() and to_json honour only() and exclude() (#293)
• Document serialization uses field order to ensure a strict order is set (#296)
• DecimalField now stores as float not string (#289)
• UUIDField now stores as a binary by default (#292)
• Added Custom User Model for Django 1.5 (#285)
• Cascading saves now default to off (#291)
• ReferenceField now store ObjectID’s by default rather than DBRef (#290)
• Added ImageField support for inline replacements (#86)
• Added SequenceField.set_next_value(value) helper (#159)
• Updated only() behaviour - now like exclude it is chainable (#202)
• Added with_limit_and_skip support to count() (#235)
• Objects queryset manager now inherited (#256)
• Updated connection to use MongoClient (#262, #274)
• Fixed db_alias and inherited Documents (#143)
• Documentation update for document errors (#124)
• Deprecated get_or_create (#35)
• Updated inheritable objects created by upsert now contain _cls (#118)
• Added support for creating documents with embedded documents in a single operation (#6)
• Added to_json and from_json to Document (#1)
• Added to_json and from_json to QuerySet (#131)
• Updated index creation now tied to Document class (#102)
• Added none() to queryset (#127)
• Updated SequenceFields to allow post processing of the calculated counter value (#141)
• Added clean method to documents for pre validation data cleaning (#60)
• Added support setting for read preference at a query level (#157)
• Added _instance to EmbeddedDocuments pointing to the parent (#139)
• Inheritance is off by default (#122)
• Remove _types and just use _cls for inheritance (#148)
• Only allow QNode instances to be passed as query objects (#199)
• Dynamic fields are now validated on save (#153) (#154)
• Added support for multiple slices and made slicing chainable. (#170) (#190) (#191)
• Fixed GridFSProxy __getattr__ behaviour (#196)
• Fix Django timezone support (#151)
• Simplified Q objects, removed QueryTreeTransformerVisitor (#98) (#171)
• FileFields now copyable (#198)
• Querysets now return clones and are no longer edit in place (#56)
• Added support for $maxDistance (#179)
• Uses getLastError to test created on updated saves (#163)
• Fixed inheritance and unique index creation (#140)
• Fixed reverse delete rule with inheritance (#197)
• Fixed validation for GenericReferences which haven’t been dereferenced
• Added switch_db context manager (#106)
• Added switch_db method to document instances (#106)
• Added no_dereference context manager (#82) (#61)
• Added switch_collection context manager (#220)
• Added switch_collection method to document instances (#220)
• Added support for compound primary keys (#149) (#121)
• Fixed overriding objects with custom manager (#58)
• Added no_dereference method for querysets (#82) (#61)
• Undefined data should not override instance methods (#49)
• Added Django Group and Permission (#142)
• Added Doc class and pk to Validation messages (#69)
• Fixed Documents deleted via a queryset don’t call any signals (#105)
• Added the “get_decoded” method to the MongoSession class (#216)
• Fixed invalid choices error bubbling (#214)
• Updated Save so it calls $set and Sunset in a single operation (#211)
• Fixed inner queryset looping (#204)

4.4.9 Changes in 0.7.10

• Fix UnicodeEncodeError for dbref (#278)
• Allow construction using positional parameters (#268)
• Updated EmailField length to support long domains (#243)
• Added 64-bit integer support (#251)
• Added Django sessions TTL support (#224)
• Fixed issue with numerical keys in MapField(EmbeddedDocumentField()) (#240)
• Fixed clearing _changed_fields for complex nested embedded documents (#237, #239, #242)
• Added “id” back to _data dictionary (#255)
• Only mark a field as changed if the value has changed (#258)
• Explicitly check for Document instances when dereferencing (#261)
• Fixed order_by chaining issue (#265)
• Added dereference support for tuples (#250)
• Resolve field name to db field name when using distinct(#260, #264, #269)
• Added kwargs to doc.save to help interop with django (#223, #270)
• Fixed cloning querysets in PY3

4.4. Changelog
4.4.10 Changes in 0.7.9

- Better fix handling for old style _types
- Embedded SequenceFields follow collection naming convention

4.4.11 Changes in 0.7.8

- Fix sequence fields in embedded documents (#166)
- Fix query chaining with .order_by() (#176)
- Added optional encoding and collection config for Django sessions (#180, #181, #183)
- Fixed EmailField so can add extra validation (#173, #174, #187)
- Fixed bulk inserts can now handle custom pk’s (#192)
- Added as_pymongo method to return raw or cast results from pymongo (#193)

4.4.12 Changes in 0.7.7

- Fix handling for old style _types

4.4.13 Changes in 0.7.6

- Unicode fix for repr (#133)
- Allow updates with match operators (#144)
- Updated URLField - now can have a override the regex (#136)
- Allow Django AuthenticationBackends to work with Django user (hmarr/mongoengine#573)
- Fixed reload issue with ReferenceField where dbref=False (#138)

4.4.14 Changes in 0.7.5

- ReferenceFields with dbref=False use ObjectId instead of strings (#134) See ticket for upgrade notes (#134)

4.4.15 Changes in 0.7.4

- Fixed index inheritance issues - firmed up testcases (#123) (#125)

4.4.16 Changes in 0.7.3

- Reverted EmbeddedDocuments meta handling - now can turn off inheritance (#119)

4.4.17 Changes in 0.7.2

- Update index spec generation so its not destructive (#113)
4.4.18 Changes in 0.7.1

• Fixed index spec inheritance (#111)

4.4.19 Changes in 0.7.0

• Updated queryset.delete so you can use with skip / limit (#107)
• Updated index creation allows kwargs to be passed through refs (#104)
• Fixed Q object merge edge case (#109)
• Fixed reloading on sharded documents (hmarr/mongoengine#569)
• Added NotUniqueError for duplicate keys (#62)
• Added custom collection / sequence naming for SequenceFields (#92)
• Fixed UnboundLocalError in composite index with pk field (#88)
• Updated ReferenceField’s to optionally store ObjectId strings this will become the default in 0.8 (#89)
• Added FutureWarning - save will default to cascade=False in 0.8
• Added example of indexing embedded document fields (#75)
• Fixed ImageField resizing when forcing size (#80)
• Add flexibility for fields handling bad data (#78)
• Embedded Documents no longer handle meta definitions
• Use weakref proxies in base lists / dicts (#74)
• Improved queryset filtering (hmarr/mongoengine#554)
• Fixed Dynamic Documents and Embedded Documents (hmarr/mongoengine#561)
• Fixed abstract classes and shard keys (#64)
• Fixed Python 2.5 support
• Added Python 3 support (thanks to Laine Heron)

4.4.20 Changes in 0.6.20

• Added support for distinct and db_alias (#59)
• Improved support for chained querysets when constraining the same fields (hmarr/mongoengine#554)
• Fixed BinaryField lookup re (#48)

4.4.21 Changes in 0.6.19

• Added Binary support to UUID (#47)
• Fixed MapField lookup for fields without declared lookups (#46)
• Fixed BinaryField python value issue (#48)
• Fixed SequenceField non numeric value lookup (#41)
• Fixed queryset manager issue (#52)
• Fixed FileField compairison (hmarr/mongoengine#547)
4.4.22 Changes in 0.6.18

• Fixed recursion loading bug in _get_changed_fields

4.4.23 Changes in 0.6.17

• Fixed issue with custom queryset manager expecting explicit variable names

4.4.24 Changes in 0.6.16

• Fixed issue where db_alias wasn’t inherited

4.4.25 Changes in 0.6.15

• Updated validation error messages
• Added support for null / zero / false values in item_frequencies
• Fixed cascade save edge case
• Fixed geo index creation through reference fields
• Added support for args / kwargs when using @queryset_manager
• Deref list custom id fix

4.4.26 Changes in 0.6.14

• Fixed error dict with nested validation
• Fixed Int/Float fields and not equals None
• Exclude tests from installation
• Allow tuples for index meta
• Fixed use of str in instance checks
• Fixed unicode support in transform update
• Added support for add_to_set and each

4.4.27 Changes in 0.6.13

• Fixed EmbeddedDocument db_field validation issue
• Fixed StringField unicode issue
• Fixes __repr__ modifying the cursor

4.4.28 Changes in 0.6.12

• Fixes scalar lookups for primary_key
• Fixes error with _delta handling DBRefs
4.4.29 Changes in 0.6.11

- Fixed inconsistency handling None values field attrs
- Fixed map_field embedded db_field issue
- Fixed .save() _delta issue with DbRefs
- Fixed Django TestCase
- Added cmp to Embedded Document
- Added PULL reverse_delete_rule
- Fixed CASCADE delete bug
- Fixed db_field data load error
- Fixed recursive save with FileField

4.4.30 Changes in 0.6.10

- Fixed basedict / baselist to return super(...)
- Promoted BaseDynamicField to DynamicField

4.4.31 Changes in 0.6.9

- Fixed sparse indexes on inherited docs
- Removed FileField auto deletion, needs more work maybe 0.7

4.4.32 Changes in 0.6.8

- Fixed FileField losing reference when no default set
- Removed possible race condition from FileField (grid_file)
- Added assignment to save, can now do: b = MyDoc(**kwargs).save()
- Added support for pull operations on nested EmbeddedDocuments
- Added support for choices with GenericReferenceFields
- Added support for choices with GenericEmbeddedDocumentFields
- Fixed Django 1.4 sessions first save data loss
- FileField now automatically delete files on .delete()
- Fix for GenericReference to_mongo method
- Fixed connection regression
- Updated Django User document, now allows inheritance

4.4.33 Changes in 0.6.7

- Fixed indexing on ‘_id’ or ‘pk’ or ‘id’
- Invalid data from the DB now raises a InvalidDocumentError
- Cleaned up the Validation Error - docs and code
- Added meta auto_create_index so you can disable index creation
- Added write concern options to inserts
4.4.34 Changes in 0.6.6

- Django 1.4 fixed (finally)
- Added tests for Django

4.4.35 Changes in 0.6.5

- More Django updates

4.4.36 Changes in 0.6.4

- Refactored connection / fixed replicasetconnection
- Bug fix for unknown connection alias error message
- Sessions support Django 1.3 and Django 1.4
- Minor fix for ReferenceField

4.4.37 Changes in 0.6.3

- Updated sessions for Django 1.4
- Bug fix for updates where listfields contain embedded documents
- Bug fix for collection naming and mixins

4.4.38 Changes in 0.6.2

- Updated documentation for ReplicaSet connections
- Hack round _types issue with SERVER-5247 - querying other arrays may also cause problems.

4.4.39 Changes in 0.6.1

- Fix for replicaSet connections

4.4.40 Changes in 0.6

- Added FutureWarning to inherited classes not declaring ‘allow_inheritance’ as the default will change in 0.7
- Added support for covered indexes when inheritance is off
- No longer always upsert on save for items with a ‘_id’
- Error raised if update doesn’t have an operation
- DeReferencing is now thread safe
- Errors raised if trying to perform a join in a query
• Updates can now take __raw__ queries
• Added custom 2D index declarations
• Added replicaSet connection support
• Updated deprecated imports from pymongo (safe for pymongo 2.2)
• Added uri support for connections
• Added scalar for efficiently returning partial data values (aliased to values_list)
• Fixed limit skip bug
• Improved Inheritance / Mixin
• Added sharding support
• Added pymongo 2.1 support
• Fixed Abstract documents can now declare indexes
• Added db_alias support to individual documents
• Fixed GridFS documents can now be pickled
• Added Now raises an InvalidDocumentError when declaring multiple fields with the same db_field
• Added InvalidQueryError when calling with_id with a filter
• Added support for DBRefs in distinct()
• Fixed issue saving False booleans
• Fixed issue with dynamic documents deltas
• Added Reverse Delete Rule support to ListFields - MapFields aren’t supported
• Added customisable cascade kwarg options
• Fixed Handle None values for non-required fields
• Removed Document._get_subclasses() - no longer required
• Fixed bug requiring subclasses when not actually needed
• Fixed deletion of dynamic data
• Added support for the $elementMatch operator
• Added reverse option to SortedListFields
• Fixed dereferencing - multi directional list dereferencing
• Fixed issue creating indexes with recursive embedded documents
• Fixed recursive lookup in _unique_with_indexes
• Fixed passing ComplexField defaults to constructor for ReferenceFields
• Fixed validation of DictField Int keys
• Added optional cascade saving
• Fixed dereferencing - max_depth now taken into account
• Fixed document mutation saving issue
• Fixed positional operator when replacing embedded documents
• Added Non-Django Style choices back (you can have either)
• Fixed __repr__ of a sliced queryset
• Added recursive validation error of documents / complex fields
• Fixed breaking during queryset iteration

4.4. Changelog
• Added pre and post bulk-insert signals
• Added ImageField - requires PIL
• Fixed Reference Fields can be None in get_or_create / queries
• Fixed accessing pk on an embedded document
• Fixed calling a queryset after drop_collection now recreates the collection
• Add field name to validation exception messages
• Added UUID field
• Improved efficiency of .get()
• Updated ComplexFields so if required they won’t accept empty lists / dicts
• Added spec file for rpm-based distributions
• Fixed ListField so it doesn’t accept strings
• Added DynamicDocument and EmbeddedDynamicDocument classes for expando schemas

4.4.41 Changes in v0.5.2

• A Robust Circular reference bugfix

4.4.42 Changes in v0.5.1

• Fixed simple circular reference bug

4.4.43 Changes in v0.5

• Added InvalidDocumentError - so Document core methods can’t be overwritten
• Added GenericEmbeddedDocument - so you can embed any type of embeddable document
• Added within_polygon support - for those with mongodb 1.9
• Updated sum / average to use map_reduce as db.eval doesn’t work in sharded environments
• Added where() - filter to allowing users to specify query expressions as Javascript
• Added SequenceField - for creating sequential counters
• Added update() convenience method to a document
• Added cascading saves - so changes to Referenced documents are saved on .save()
• Added select_related() support
• Added support for the positional operator
• Updated geo index checking to be recursive and check in embedded documents
• Updated default collection naming convention
• Added Document Mixin support
• Fixed queryet __repr__ mid iteration
• Added hint() support, so cantell Mongo the proper index to use for the query
• Fixed issue with inconsistent setting of __cls breaking inherited referencing
• Added help_text and verbose_name to fields to help with some form libs
• Updated item_frequencies to handle embedded document lookups
- Added delta tracking now only sets / unsets explicitly changed fields
- Fixed saving so sets updated values rather than overwrites
- Added ComplexDateTimeField - Handles datetimes correctly with microseconds
- Added ComplexBaseField - for improved flexibility and performance
- Added get_FIELD_display() method for easy choice field displaying
- Added queryset.slave_okay(enabled) method
- Updated queryset.timeout(enabled) and queryset.snapshot(enabled) to be chainable
- Added insert method for bulk inserts
- Added blinker signal support
- Added query_counter context manager for tests
- Added map_reduce method item_frequencies and set as default (as db.eval doesn’t work in sharded environments)
- Added inline_map_reduce option to map_reduce
- Updated connection exception so it provides more info on the cause.
- Added searching multiple levels deep in DictField
- Added DictField entries containing strings to use matching operators
- Added MapField, similar to DictField
- Added Abstract Base Classes
- Added Custom Objects Managers
- Added sliced subfields updating
- Added NotRegistered exception if dereferencing Document not in the registry
- Added a write concern for save, update, update_one and get_or_create
- Added slicing / subarray fetching controls
- Fixed various unique index and other index issues
- Fixed threaded connection issues
- Added spherical geospatial query operators
- Updated queryset to handle latest version of pymongo map_reduce now requires an output.
- Added Document __hash__, __ne__ for pickling
- Added FileField optional size arg for read method
- Fixed FileField seek and tell methods for reading files
- Added QuerySet.clone to support copying querysets
- Fixed item_frequencies when using name thats the same as a native js function
- Added reverse delete rules
- Fixed issue with unset operation
- Fixed Q-object bug
- Added QuerySet.all_fields resets previous .only() and .exclude()
- Added QuerySet.exclude
- Added django style choices
- Fixed order and filter issue
• Added QuerySet.only subfield support
• Added creation_counter to BaseField allowing fields to be sorted in the way the user has specified them
• Fixed various errors
• Added many tests

4.4.44 Changes in v0.4

• Added GridFSStorage Django storage backend
• Added FileField for GridFS support
• New Q-object implementation, which is no longer based on Javascript
• Added SortedListField
• Added EmailField
• Added GeoPointField
• Added exact and iexact match operators to QuerySet
• Added get_document_or_404 and get_list_or_404 Django shortcuts
• Added new query operators for Geo queries
• Added not query operator
• Added new update operators: pop and add_to_set
• Added __raw__ query parameter
• Added support for custom querysets
• Fixed document inheritance primary key issue
• Added support for querying by array element position
• Base class can now be defined for DictField
• Fixed MRO error that occured on document inheritance
• Added QuerySet.distinct, QuerySet.create, QuerySet.snapshot, QuerySet.timeout and QuerySet.all
• Subsequent calls to connect() now work
• Introduced min_length for StringField
• Fixed multi-process connection issue
• Other minor fixes

4.4.45 Changes in v0.3

• Added MapReduce support
• Added contains, startswith and endswith query operators (and case-insensitive versions that are prefixed with ‘i’)
• Deprecated fields’ name parameter, replaced with db_field
• Added QuerySet.only for only retrieving specific fields
• Added QuerySet.in_bulk() for bulk querying using ids
• QuerySets now have a rewind() method, which is called automatically when the iterator is exhausted, allowing QuerySets to be reused
- Added DictField
- Added URLField
- Added DecimalField
- Added BinaryField
- Added GenericReferenceField
- Added get() and get_or_create() methods to QuerySet
- ReferenceFields may now reference the document they are defined on (recursive references) and documents that have not yet been defined
- Document objects may now be compared for equality (equal if _ids are equal and documents are of same type)
- QuerySet update methods now have an upsert parameter
- Added field name substitution for Javascript code (allows the user to use the Python names for fields in JS, which are later substituted for the real field names)
- Q objects now support regex querying
- Fixed bug where referenced documents within lists weren’t properly dereferenced
- ReferenceFields may now be queried using their _id
- Fixed bug where EmbeddedDocuments couldn’t be non-polymorphic
- queryset_manager functions now accept two arguments – the document class as the first and the query-set as the second
- Fixed bug where QuerySet.exec_js ignored Q objects
- Other minor fixes

4.4.46 Changes in v0.2.2

- Fixed bug that prevented indexes from being used on ListFields
- Document.filter() added as an alias to Document.__call__()
- validate() may now be used on EmbeddedDocuments

4.4.47 Changes in v0.2.1

- Added a MongoEngine backend for Django sessions
- Added force_insert to Document.save()
- Improved querying syntax for ListField and EmbeddedDocumentField
- Added support for user-defined primary keys (_id in MongoDB)

4.4.48 Changes in v0.2

- Added Q class for building advanced queries
- Added QuerySet methods for atomic updates to documents
- Fields may now specify unique=True to enforce uniqueness across a collection
- Added option for default document ordering
- Fixed bug in index definitions
4.4.49 Changes in v0.1.3

- Added Django authentication backend
- Added Document.meta support for indexes, which are ensured just before querying takes place
- A few minor bugfixes

4.4.50 Changes in v0.1.2

- Query values may be processed before before being used in queries
- Made connections lazy
- Fixed bug in Document dictionary-style access
- Added BooleanField
- Added Document.reload() method

4.4.51 Changes in v0.1.1

- Documents may now use capped collections

4.5 Upgrading

4.5.1 0.8.7

Calling reload on deleted / nonexistant documents now raises a DoesNotExist exception.

4.5.2 0.8.2 to 0.8.3

Minor change that may impact users:
DynamicDocument fields are now stored in creation order after any declared fields. Previously they were stored alphabetically.

4.5.3 0.7 to 0.8

There have been numerous backwards breaking changes in 0.8. The reasons for these are to ensure that MongoEngine has sane defaults going forward and that it performs the best it can out of the box. Where possible there have been FutureWarnings to help get you ready for the change, but that hasn’t been possible for the whole of the release.

**Warning:** Breaking changes - test upgrading on a test system before putting live. There maybe multiple manual steps in migrating and these are best honed on a staging / test system.

Python and PyMongo

MongoEngine requires python 2.6 (or above) and pymongo 2.5 (or above)
## Inheritance

The inheritance model has changed, we no longer need to store an array of `types` with the model we can just use the classname in `_cls`. This means that you will have to update your indexes for each of your inherited classes like so:

```python
# 1. Declaration of the class
class Animal(Document):
    name = StringField()
    meta = {
        'allow_inheritance': True,
        'indexes': ['name']
    }

# 2. Remove _types
collection = Animal._get_collection()
collection.update({}, {'$unset': {'_types': 1}}, multi=True)

# 3. Confirm extra data is removed
assert collection.find({'_types': {'$exists': True}}).count() == 0

# 4. Remove indexes
info = collection.index_information()
indexes_to_drop = [key for key, value in info.iteritems() if '_types' in dict(value['key'])]
for index in indexes_to_drop:
    collection.drop_index(index)

# 5. Recreate indexes
Animal.ensure_indexes()
```

### Document Definition

The default for inheritance has changed - it is now off by default and `_cls` will not be stored automatically with the class. So if you extend your `Document` or `EmbeddedDocuments` you will need to declare `allow_inheritance` in the meta data like so:

```python
class Animal(Document):
    name = StringField()
    meta = {'allow_inheritance': True}
```

Previously, if you had data in the database that wasn’t defined in the Document definition, it would set it as an attribute on the document. This is no longer the case and the data is set only in the `document._data` dictionary:

```python
>>> from mongoengine import *

>>> class Animal(Document):
...     name = StringField()
...     ...

>>> cat = Animal(name="kit", size="small")

# 0.7
>>> cat.size
u'small'

# 0.8
>>> cat.size
Traceback (most recent call last):
```

---

4.5. Upgrading 77
The Document class has introduced a reserved function `clean()`, which will be called before saving the document. If your document class happens to have a method with the same name, please try to rename it.

```python
def clean(self):    pass
```

**ReferenceField**

ReferenceFields now store ObjectIds by default - this is more efficient than DBRefs as we already know what Document types they reference:

```python
# Old code
class Animal(Document):
    name = ReferenceField('self')

# New code to keep dbrefs
class Animal(Document):
    name = ReferenceField('self', dbref=True)
```

To migrate all the references you need to touch each object and mark it as dirty eg:

```python
# Doc definition
class Person(Document):
    name = StringField()
    parent = ReferenceField('self')
    friends = ListField(ReferenceField('self'))

# Mark all ReferenceFields as dirty and save
for p in Person.objects:
    p._mark_as_changed('parent')
    p._mark_as_changed('friends')
    p.save()
```

An example test migration for ReferenceFields is available on github.

**Note:** Internally mongoengine handles ReferenceFields the same, so they are converted to DBRef on loading and ObjectIds or DBRefs depending on settings on storage.

**UUIDField**

UUIDFields now default to storing binary values:

```python
# Old code
class Animal(Document):
    uuid = UUIDField()

# New code
class Animal(Document):
    uuid = UUIDField(binary=False)
```

To migrate all the uuids you need to touch each object and mark it as dirty eg:

```python
# Doc definition
class Animal(Document):
    uuid = UUIDField()

# Mark all UUIDFields as dirty and save
for a in Animal.objects:
```
An example test migration for UUIDFields is available on github.

DecimalField

DecimalFields now store floats - previously it was storing strings and that made it impossible to do comparisons when querying correctly:

```python
a._mark_as_changed('uuid')
a.save()
```

An example test migration for UUIDFields is available on github.

Cascading Saves

To improve performance document saves will no longer automatically cascade. Any changes to a Document’s references will either have to be saved manually or you will have to explicitly tell it to cascade on save:

```python
for p in Person.objects:
    p._mark_as_changed('balance')
p.save()
```

Storage

Document and Embedded Documents are now serialized based on declared field order. Previously, the data was passed to mongodb as a dictionary and which meant that order wasn’t guaranteed - so things like $addToSet operations on EmbeddedDocument could potentially fail in unexpected ways.

If this impacts you, you may want to rewrite the objects using the `doc.mark_as_dirty('field')` pattern described above. If you are using a compound primary key then you will need to ensure the order is fixed and match your EmbeddedDocument to that order.
Querysets

Attack of the clones

Querysets now return clones and should no longer be considered editable in place. This brings us in line with how Django’s querysets work and removes a long running gotcha. If you edit your querysets inplace you will have to update your code like so:

```python
# Old code:
mammals = Animal.objects(type="mammal")
mammals.filter(order="Carnivora")
[m for m in mammals]  # Returns a cloned queryset that isn't assigned to anything - so this will break in 0.8

# Update example a) assign queryset after a change:
mammals = Animal.objects(type="mammal")
carnivores = mammals.filter(order="Carnivora")  # Reassign the new queryset so filter can be applied
[m for m in carnivores]  # This will return all carnivores

# Update example b) chain the queryset:
mammals = Animal.objects(type="mammal").filter(order="Carnivora")  # The final queryset is assigned
[m for m in mammals]  # This will return all carnivores
```

Len iterates the queryset

If you ever did `len(queryset)` it previously did a `count()` under the covers, this caused some unusual issues. As `len(queryset)` is most often used by `list(queryset)` we now cache the queryset results and use that for the length.

This isn’t as performant as a `count()` and if you aren’t iterating the queryset you should upgrade to use count:

```python
# Old code
len(Animal.objects(type="mammal"))

# New code
Animal.objects(type="mammal").count()
```

`.only()` now inline with `.exclude()`

The behaviour of `.only()` was highly ambiguous, now it works in mirror fashion to `.exclude()`. Chaining `.only()` calls will increase the fields required:

```python
# Old code
Animal.objects().only(['type', 'name']).only('name', 'order')  # Would have returned just `name`

# New code
Animal.objects().only('name')

# Note:
Animal.objects().only(['name']).only('order')  # Now returns `name` *and* `order`
```

Client

PyMongo 2.4 came with a new connection client: `MongoClient` and started the depreciation of the old `Connection`. MongoEngine now uses the latest `MongoClient` for connections. By default operations were `safe` but if you turned them off or used the connection directly this will impact your queries.
Safe  safe has been depreciated in the new MongoClient connection. Please use write_concern instead. As safe always defaulted as True normally no code change is required. To disable confirmation of the write just pass `{"w": 0}` eg:

```python
# Old
Animal(name="Dinosaurs").save(safe=False)
# new code:
Animal(name="Dinosaurs").save(write_concern={"w": 0})
```

Write Concern  write_options has been replaced with write_concern to bring it inline with pymongo. To upgrade simply rename any instances where you used the write_option keyword to write_concern like so:

```python
# Old code:
Animal(name="Dinosaurs").save(write_options={"w": 2})
# new code:
Animal(name="Dinosaurs").save(write_concern={"w": 2})
```

Indexes

Index methods are no longer tied to querysets but rather to the document class. Although QuerySet.ensure_indexes and QuerySet.ensure_index still exist. They should be replaced with ensure_indexes() / ensure_index().

SequenceFields

SequenceField now inherits from BaseField to allow flexible storage of the calculated value. As such MIN and MAX settings are no longer handled.

4.5.4 0.6 to 0.7

Cascade saves

Saves will raise a FutureWarning if they cascade and cascade hasn’t been set to True. This is because in 0.8 it will default to False. If you require cascading saves then either set it in the meta or pass via save eg

```python
# At the class level:
class Person(Document):
    meta = {'cascade': True}

# Or in code:
my_document.save(cascade=True)
```

Note: Remember: cascading saves do not cascade through lists.

ReferenceFields

ReferenceFields now can store references as ObjectId strings instead of DBRefs. This will become the default in 0.8 and if dbref is not set a FutureWarning will be raised.

To explicitly continue to use DBRefs change the dbref flag to True

```python
class Person(Document):
    groups = ListField(ReferenceField(Group, dbref=True))
```
To migrate to using strings instead of DBRefs you will have to manually migrate

```python
# Step 1 - Migrate the model definition
class Group(Document):
    author = ReferenceField(User, dbref=False)
    members = ListField(ReferenceField(User, dbref=False))

# Step 2 - Migrate the data
for g in Group.objects():
    g.author = g.author
    g.members = g.members
    g.save()
```

### item_frequencies

In the 0.6 series we added support for null / zero / false values in item_frequencies. A side effect was to return keys in the value they are stored in rather than as string representations. Your code may need to be updated to handle native types rather than strings keys for the results of item frequency queries.

### BinaryFields

Binary fields have been updated so that they are native binary types. If you previously were doing `str` comparisons with binary field values you will have to update and wrap the value in a `str`.

#### 4.5.5 0.5 to 0.6

Embedded Documents - if you had a `pk` field you will have to rename it from `_id` to `pk` as `pk` is no longer a property of Embedded Documents.

Reverse Delete Rules in Embedded Documents, MapFields and DictFields now throw an InvalidDocument error as they aren’t currently supported.

`Document._get_subclasses` - Is no longer used and the class method has been removed.

`Document.objects.with_id` - now raises an InvalidQueryError if used with a filter.

FutureWarning - A future warning has been added to all inherited classes that don’t define `allow_inheritance` in their meta.

You may need to update pyMongo to 2.0 for use with Sharding.

#### 4.5.6 0.4 to 0.5

There have been the following backwards incompatibilities from 0.4 to 0.5. The main areas of changed are: choices in fields, map_reduce and collection names.

### Choice options:

Are now expected to be an iterable of tuples, with the first element in each tuple being the actual value to be stored. The second element is the human-readable name for the option.

### PyMongo / MongoDB

map reduce now requires pymongo 1.11+
The pymongo `merge_output` and `reduce_output` parameters, have been depreciated.

More methods now use `map_reduce` as `db.eval` is not supported for sharding as such the following have been changed:
• `sum()`
• `average()`
• `item_frequencies()`

**Default collection naming**

Previously it was just lowercase, it’s now much more pythonic and readable as it’s lowercase and underscores,
previously

```python
def class MyAceDocument (Document):
    pass
MyAceDocument._meta['collection'] == myacedocument
```

In 0.5 this will change to

```python
class MyAceDocument (Document):
    pass
MyAceDocument._get_collection_name() == my_ace_document
```

To upgrade use a Mixin class to set meta like so

```python
class BaseMixin (object):
    meta = {
        'collection': lambda c: c.__name__.lower()
    }
class MyAceDocument (Document, BaseMixin):
    pass
MyAceDocument._get_collection_name() == "myacedocument"
```

Alternatively, you can rename your collections eg

```python
from mongoengine.connection import _get_db
from mongoengine.base import _document_registry
def rename_collections():
    db = _get_db()

    failure = False

    collection_names = [d._get_collection_name()
        for d in _document_registry.values()]

    for new_style_name in collection_names:
        if not new_style_name: # embedded documents don't have collections
            continue
        old_style_name = new_style_name.replace('_', ' ')

        if old_style_name == new_style_name:
            continue # Nothing to do

        existing = db.collection_names()
        if old_style_name in existing:
            if new_style_name in existing:
                failure = True
                print "FAILED to rename: $s to $s (already exists)" % (old_style_name, new_style_name)
            else:
                db[old_style_name].rename(new_style_name)
```

4.5. Upgrading
print "Renamed: %s to %s" % (old_style_name,
new_style_name)

if failure:
    print "Upgrading collection names failed"
else:
    print "Upgraded collection names"

mongodb 1.8 > 2.0 +

It’s been reported that indexes may need to be recreated to the newer version of indexes. To do this drop indexes and call ensure_indexes on each model.

4.6 Django Support

Note: Updated to support Django 1.5

4.6.1 Connecting

In your settings.py file, ignore the standard database settings (unless you also plan to use the ORM in your project), and instead call connect() somewhere in the settings module.

Note: If you are not using another Database backend you may need to add a dummy database backend to settings.py eg:

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.dummy'
    }
}

4.6.2 Authentication

MongoEngine includes a Django authentication backend, which uses MongoDB. The User model is a MongoDB Document, but implements most of the methods and attributes that the standard Django User model does - so the two are moderately compatible. Using this backend will allow you to store users in MongoDB but still use many of the Django authentication infrastructure (such as the login_required() decorator and the authenticate() function). To enable the MongoEngine auth backend, add the following to your settings.py file:

AUTHENTICATION_BACKENDS = (  
    'mongoengine.django.auth.MongoEngineBackend',
)

The auth module also contains a get_user() helper function, that takes a user’s id and returns a User object.

New in version 0.1.3.

4.6.3 Custom User model

Django 1.5 introduced Custom user Models which can be used as an alternative to the MongoEngine authentication backend.
The main advantage of this option is that other components relying on `django.contrib.auth` and supporting the new swappable user model are more likely to work. For example, you can use the `createsuperuser` management command as usual.

To enable the custom User model in Django, add `mongoengine.django.mongo_auth` in your `INSTALLED_APPS` and set `mongo_auth.MongoUser` as the custom user user model to use. In your `settings.py` file you will have:

```python
INSTALLED_APPS = (
    ...
    'django.contrib.auth',
    'mongoengine.django.mongo_auth',
    ...
)

AUTH_USER_MODEL = 'mongo_auth.MongoUser'
```

An additional `MONGOENGINE_USER_DOCUMENT` setting enables you to replace the `User` class with another class of your choice:

```python
MONGOENGINE_USER_DOCUMENT = 'mongoengine.django.auth.User'
```

The custom `User` must be a `Document` class, but otherwise has the same requirements as a standard custom user model, as specified in the Django Documentation. In particular, the custom class must define `USERNAME_FIELD` and `REQUIRED_FIELDS` attributes.

### 4.6.4 Sessions

Django allows the use of different backend stores for its sessions. MongoEngine provides a MongoDB-based session backend for Django, which allows you to use sessions in your Django application with just MongoDB. To enable the MongoEngine session backend, ensure that your settings module has `django.contrib.sessions.middleware.SessionMiddleware` in the `MIDDLEWARE_CLASSES` field and `django.contrib.sessions` in your `INSTALLED_APPS`. From there, all you need to do is add the following line into your settings module:

```python
SESSION_ENGINE = 'mongoengine.django.sessions'
SESSION_SERIALIZER = 'mongoengine.django.sessions.BSONSerializer'
```

Django provides session cookie, which expires after `SESSION_COOKIE_AGE` seconds, but doesn’t delete cookie at sessions backend, so `mongoengine.django.sessions` supports mongodb TTL.

**Note:** `SESSION_SERIALIZER` is only necessary in Django 1.6 as the default serializer is based around JSON and doesn’t know how to convert `bson.objectid.ObjectId` instances to strings.

New in version 0.2.1.

### 4.6.5 Storage

With MongoEngine’s support for GridFS via the `FileField`, it is useful to have a Django file storage backend that wraps this. The new storage module is called `GridFSStorage`. Using it is very similar to using the default `FileSystemStorage`.

```python
from mongoengine.django.storage import GridFSStorage
fs = GridFSStorage()
filename = fs.save('hello.txt', 'Hello, World!')
```

All of the Django Storage API methods have been implemented except `path()`. If the filename provided already exists, an underscore and a number (before # the file extension, if one exists) will be appended to the filename until the generated filename doesn’t exist. The `save()` method will return the new filename.
MongoEngine Documentation, Release 0.8.7

>>> fs.exists('hello.txt')
True
>>> fs.open('hello.txt').read()
'Hello, World!
>>> fs.size('hello.txt')
13
>>> fs.url('hello.txt')
'http://your_media_url/hello.txt'
>>> fs.open('hello.txt').name
'hello.txt'
>>> fs.listdir()
([], [u'hello.txt'])

All files will be saved and retrieved in GridFS via the FileDocument document, allowing easy access to the files without the GridFSSStorage backend:

```python
>>> from mongoengine.django.storage import FileDocument
>>> FileDocument.objects()
```

New in version 0.4.

### 4.6.6 Shortcuts

Inspired by the Django shortcut `get_object_or_404`, the `get_document_or_404()` method returns a document or raises an Http404 exception if the document does not exist:

```python
from mongoengine.django.shortcuts import get_document_or_404
admin_user = get_document_or_404(User, username='root')
```

The first argument may be a Document or QuerySet object. All other passed arguments and keyword arguments are used in the query:

```python
foo_email = get_document_or_404(User.objects.only('email'), username='foo', is_active=True).email
```

**Note:** Like with `get()`, a MultipleObjectsReturned will be raised if more than one object is found.

Also inspired by the Django shortcut `get_list_or_404`, the `get_list_or_404()` method returns a list of documents or raises an Http404 exception if the list is empty:

```python
from mongoengine.django.shortcuts import get_list_or_404
active_users = get_list_or_404(User, is_active=True)
```

The first argument may be a Document or QuerySet object. All other passed arguments and keyword arguments are used to filter the query.
Indices and tables

- genindex
- modindex
- search
m
mongoengine.queryset, 45
Symbols
__call__() (mongoengine.queryset.QuerySet method), 45
__call__() (mongoengine.queryset.QuerySetNoCache method), 52
_import_class() (in module mongoengine.common), 61
A
all() (mongoengine.queryset.QuerySet method), 45
all_fields() (mongoengine.queryset.QuerySet method), 45
as_pymongo() (mongoengine.queryset.QuerySet method), 45
average() (mongoengine.queryset.QuerySet method), 45
B
BaseField (class in mongoengine.base.fields), 52
BinaryField (class in mongoengine.fields), 58
BooleanField (class in mongoengine.fields), 53
C
cache() (mongoengine.queryset.QuerySetNoCache method), 52
cascade_save() (mongoengine.Document method), 41
call() (mongoengine.queryset.QuerySet method), 45
call_into() (mongoengine.queryset.QuerySet method), 46
call_indexes() (mongoengine.Document class method), 41
drop_collection() (mongoengine.Document class method), 41
Document (class in mongoengine), 40
DynamicDocument (class in mongoengine), 43
DynamicEmbeddedDocument (class in mongoengine), 43
DynamicField (class in mongoengine.fields), 55
E
EmbeddedDocument (class in mongoengine), 43
EmbeddedDocumentField (class in mongoengine.fields), 55
ensure_index() (mongoengine.Document class method), 41
ensure_index() (mongoengine.queryset.QuerySet method), 41
ensure_indexes() (mongoengine.Document class method), 41
exclude() (mongoengine.queryset.QuerySet method), 46
exec_js() (mongoengine.queryset.QuerySet method), 46
explain() (mongoengine.queryset.QuerySet method), 47
F
fields() (mongoengine.queryset.QuerySet method), 47
FilterField (class in mongoengine.fields), 58
filter() (mongoengine.queryset.QuerySet method), 47
first() (mongoengine.queryset.QuerySet method), 47
FloatField (class in mongoengine.fields), 53
from_json() (mongoengine.queryset.QuerySet method), 47
G
GenericEmbeddedDocumentField (class in mongoengine.fields), 55
GenericReferenceField (class in mongoengine.fields), 57
GeoPointField (class in mongoengine.fields), 59
get() (mongoengine.queryset.QuerySet method), 47
get_or_create() (mongoengine.queryset.QuerySet method), 47
GridFSError (class in mongoengine.fields), 60
GridFSProxy (class in mongoengine.fields), 60

H
hint() (mongoengine.queryset.QuerySet method), 48

I
ImageField (class in mongoengine.fields), 58
ImageGridFSProxy (class in mongoengine.fields), 60
ImproperlyConfigured (class in mongoengine.fields), 60
in_bulk() (mongoengine.queryset.QuerySet method), 48
insert() (mongoengine.queryset.QuerySet method), 48
IntField (class in mongoengine.fields), 53
item_frequencies() (mongoengine.queryset.QuerySet method), 48

L
limit() (mongoengine.queryset.QuerySet method), 49
LineStringField (class in mongoengine.fields), 49
list_indexes() (mongoengine.Document class method), 41
ListField (class in mongoengine.fields), 56
LongField (class in mongoengine.fields), 53

M
map_reduce() (mongoengine.queryset.QuerySet method), 49
MapField (class in mongoengine.fields), 56
MapReduceDocument (class in mongoengine.document), 43
mongoengine.queryset (module), 45
my_metaclass (mongoengine.Document attribute), 42
my_metaclass (mongoengine.DynamicDocument attribute), 43
my_metaclass (mongoengine.DynamicEmbeddedDocument attribute), 43
my_metaclass (mongoengine.EmbeddedDocument attribute), 43

N
next() (mongoengine.queryset.QuerySet method), 49
no_cache() (mongoengine.queryset.QuerySet method), 49
no_dereference (class in mongoengine.context_managers), 45
no_dereference() (mongoengine.queryset.QuerySet method), 49
no_sub_classes() (mongoengine.queryset.QuerySet method), 49
none() (mongoengine.queryset.QuerySet method), 49

O
object (mongoengine.document.MapReduceDocument attribute), 44
ObjectIdField (class in mongoengine.fields), 58
objects (Document attribute), 41
only() (mongoengine.queryset.QuerySet method), 49
order_by() (mongoengine.queryset.QuerySet method), 50

P
PointField (class in mongoengine.fields), 60
PolygonField (class in mongoengine.fields), 60

Q
query_counter (class in mongoengine.context_managers), 45
QuerySet (class in mongoengine.queryset), 45
queryset_manager() (in module mongoengine.queryset), 52
QuerySetNoCache (class in mongoengine.queryset), 51

R
read_preference() (mongoengine.queryset.QuerySet method), 50
ReferenceField (class in mongoengine.fields), 56
register_connection() (in module mongoengine), 40
register_delete_rule() (mongoengine.Document class method), 42
reload() (mongoengine.Document method), 42
rewind() (mongoengine.queryset.QuerySet method), 50

S
save() (mongoengine.Document method), 42
scalar() (mongoengine.queryset.QuerySet method), 50
select_related() (mongoengine.Document method), 42
select_related() (mongoengine.queryset.QuerySet method), 50
SequenceField (class in mongoengine.fields), 58
skip() (mongoengine.queryset.QuerySet method), 50
slave_okay() (mongoengine.queryset.QuerySet method), 50
snapshot() (mongoengine.queryset.QuerySet method), 50
SortedListField (class in mongoengine.fields), 56
StringField (class in mongoengine.fields), 52
sum() (mongoengine.queryset.QuerySet method), 50
switch_collection (class in mongoengine.context_managers), 44
switch_collection() (mongoengine.Document method), 42
switch_db (class in mongoengine.context_managers), 44
switch_db() (mongoengine.Document method), 43

T
timeout() (mongoengine.queryset.QuerySet method), 51
to_dbsref() (mongoengine.Document method), 43
to_dict() (mongoengine.ValidationError method), 44
to_json() (mongoengine.queryset.QuerySet method), 51
update() (mongoengine.Document method), 43
update() (mongoengine.queryset.QuerySet method), 51
update_one() (mongoengine.queryset.QuerySet method), 51
URLField (class in mongoengine.fields), 53
UUIDField (class in mongoengine.fields), 59
Validation занял (class in mongoengine), 44
values_list() (mongoengine.queryset.QuerySet method), 51
where() (mongoengine.queryset.QuerySet method), 51
with_id() (mongoengine.queryset.QuerySet method), 51