Docu Documentation

Release 0.28.2

Andrey Mikhaylenko

January 15, 2013

CONTENTS

Doqu is a lightweight Python framework for document databases. It provides a uniform API for modeling, validation and queries across various kinds of storages.

It is not an ORM as it doesn't map existing schemata to Python objects. Instead, it lets you define schemata on a higher layer built upon a schema-less storage (key/value or document-oriented). You define models as a valuable subset of the whole database and work with only certain parts of existing entities – the parts you need.

Topics:

CHAPTER

ONE

INSTALLATION

As easy as it can be:

```
$ pip install doqu
```

Another way is to use the Mercurial repo:

```
$ hg clone http://bitbucket.org/neithere/doqu
$ cd doqu
$ ./setup.py install
```

You may also need to install some other libraries (see *Extensions*).

CHAPTER

TWO

TUTORIAL

Warning: this document must be rewritten from scratch

2.1 What does Doqu do?

2.2 Why document-oriented?

2.3 Why not just use the library X for database Y?

Native Python bindings exist for most databases. It is preferable to use a dedicated library if you are absolutely sure that your code will never be used with another database. But there are two common use cases when *Doqu* is much more preferable:

1. prototyping: if you are unsure about which database fits your requirements best and wish to test various databases against your code, just write your code with *Doqu* and then try switching backends to see which performs best. Then optimize the code for it.
2. reusing the code: if you expect the module to be plugged into an application with unpredictable settings, use *Doqu*.

Of course we are talking about document databases. For relational databases you would use an ORM.

2.4 What are "backends"?

Warning: this section is out of date

Docu can be used with a multitude of databases providing a uniform API for retrieving, storing, removing and searching of records. To couple Docu with a database, a storage/query backend is needed.

A "**backend**" is a module that provides two classes: *Storage* and *Query*. Both must conform to the basic specifications (see basic specs below). Backends may not be able to implement all default methods; they may also provide some extra methods.

The **Storage** class is an interface for the database. It allows to add, read, create and update records by primary keys. You will not use this class directly in your code.

The **Query** class is what you will talk to when filtering objects of a model. There are no constraints on how the search conditions should be represented. This is likely to cause some problems when you switch from one backend to another. Some guidlines will be probably defined to address the issue of portability. For now we try to ensure that all default backends share the conventions defined by the Tokyo Tyrant backend.

2.5 Switching backends

> **Warning:** this section is out of date

Let's assume we have a Tokyo Cabinet database. You can choose the TC backend to use the DB file *directly* or access the same file *through the manager*. The first option is great for development and some other cases where you would use SQLite; the second option is important for most production environments where multiple connections are expected. The good news is that there's no more import and export, dump/load sequences, create/alter/drop and friends. Having tested the application against the database *storage.tct* with Cabinet backend, just run *ttserver storage.tct* and switch the backend config.

Let's create our application:

```
import docu
import settings
from models import Country, Person

storage = docu.get_storage(settings.DATABASE)

print Person.objects(storage)    # prints all Person objects from DB
```

Now define settings for both backends (settings.py):

```
# direct access to the database (simple, not scalable)
TOKYO_CABINET_DATABASE = {
    'backend': 'docu.ext.tokyo_cabinet',
    'kind': 'TABLE',
    'path': 'storage.tct',
}

# access through the Tyrant manager (needs daemon, scalable)
TOKYO_TYRANT_DATABASE = {
    'backend': 'docu.ext.tokyo_tyrant',
    'host': 'localhost',
    'port': 1978,
}

# this is the *only* line you need to change in order to change the backend
DATABASE = TOKYO_CABINET_DATABASE
```

2.6 A few words on what a model is

> **Warning:** this section is out of date

First off, what is a model? Well, it's something that represents an object. The object can be stored in a database. We can fetch it from there, modify and push back.

How is a model different from a Python dictionary then? Easy. Dictionaries know nothing about where the data came from, what parts of it are important for us, how the values should be converted to and fro, and how should the data be validated before it is stored somewhere. A model of an apple *does* know what properties should an object have to be a Proper Apple; what can be done the apple so that it does not stop being a Proper Apple; and where does the apple belong so it won't be in the way when it isn't needed anymore.

In other words, the model is an answer to questions *what*, *where* and *how* about a document. And a dictionary *is* a document (or, more precisely, a simple representation of the document in given environment).

2.7 Working with documents

> **Warning:** this section is out of date

A *document* is basically a "dictionary on steroids". Let's create a document:

```
>>> from docu import *
>>> document = Document(foo=123, bar='baz')
>>> document['foo']
123
>>> document['foo'] = 456
```

Well, any dictionary can do that. But wait:

```
>>> db = get_db(backend='docu.ext.shove')
>>> document.save(db)
'new-primary-key'
>>> Document.objects(db)
[<Document: instance>]
>>> fetched = Document.objects(db)[0]
>>> document == fetched
True
>>> fetched['bar']
'baz'
```

Aha, so `Document` supports persistence! Nice. By the way, how about some syntactic sugar? Here:

```
class MyDoc(Document):
    use_dot_notation = True
```

That's the same good old *Document* but with "dot notation" switched on. It allows access to keys with `__getattr__` as well as with `__getitem__`:

```
>>> my_doc = MyDoc(foo=123)
>>> my_doc.foo
123
```

Of course this will only work with alphanumeric keys.

Now let's say we are going to make a little address book. We don't want any "foo" or "bar, just the relevant information. And the "foo" key should not be allowed in such documents. Can we restrict the *structure* to certain keys and data types? Let's see:

```
class Person(Document):
    structure = {'name': unicode, 'email': unicode}
```

Great, now the names and values are controlled. The document will raise an exception when someone, say, attempts to put a number instead of the email.

Note: Any built-in type will do; some classes are also accepted (like *datetime.date* et al). Even Document instances are accepted: they are interpreted as references. The exact set of supported types and classes is defined per storage backend because the data must be (de)serialized. It is possible to register custom converters in runtime.

(Note that the values can be *None*.) But what if we need to mark some fields as required? Or what if the email is indeed a unicode string but its content has nothing to do with RFC 5322? We need to prevent malformed data from being saved into the database. That's the daily job for *validators*:

```
from docu.validators import *

class Person(Document):
    structure = {
        'name': unicode,
        'email': unicode,
    }
    validators = {
        'name': [required()],
        'email': [optional(), email()],
    }
```

This will only allow correct data into the storage.

Note: At this point you may ask why are the definitions so verbose. Why not Field classes à la Django? Well, they *can* be added on top of what's described here. Actually Docu ships with *Document Fields* so you can easily write:

```
class Person(Document):
    name = Field(unicode, required=True)
    email = EmailField()    # this class is not implemented but can be
```

Why isn't this approach used by default? Well, it turned out that such classes introduce more problems than they solve. Too much magic, you know. Also, they quickly become a name + clutter thing. Compact but unreadable. So we adopted the MongoKit approach, i.e. semantic grouping of attributes. And — guess what? — the document classes became **much** easier to understand. Despite the definitions are a bit longer. And remember, it is always possible to add syntax sugar, but it's usually extremely hard to *remove* it.

And now, surprise: validators do an extra favour for us! Look:

```
XXX an example of query; previously defined documents are not shown because
records are filtered by validators
```

2.8 More questions?

If you can't find the answer to your questions on Docu in the documentation, feel free to ask in the discussion group.

———— XXXXXXXXXX The part below is outdated ————-

The Document behaves Let's observe the object thoroughly and conclude that colour is an important distinctive feature of this... um, sort of thing:

```
class Thing(Document):
    structure = {
        'colour': unicode
    }
```

Great, now *that's* a model. It recognizes a property as significant. Now we can compare, search and distinguish *objects* by colour (and its presence or lack). Obviously, if colour is an applicable property for an object, then it *belongs* to this model.

A more complete example which will look familiar to those who had ever used an ORM (e.g. the Django one):

```
import datetime
from docu import *

class Country(Document):
    structure = {
        'name': unicode    # any Python type; default is unicode
    }
    validators = {
        'type': [AnyOf(['country'])]
    }

    def __unicode__(self):
        return self['name']

class Person(Document):
    structure = {
        'first_name': unicode,
        'last_name': unicode,
        'gender': unicode,
        'birth_date': datetime.date,
        'birth_place': Country,    # reference to another model
    }
    validators = {
        'first_name': [required()],
        'last_name': [required()],
    }
    use_dot_notation = True

    def __unicode__(self):
        return u'{first_name} {last_name}'.format(**self)

    @property
    def age(self):
        return (datetime.datetime.now().date() - self.birth_date).days / 365
```

The interesting part is the Meta subclass. It contains a must_have attribute which actually binds the model to a subset of data in the storage. `{'first_name__exists': True}` states that a data row/document/... must have the field *first_name* defined (not necessarily non-empty). You can easily define any other query conditions (currently with respect to the backend's syntax but we hope to unify things). When you create an empty model instance, it will have all the "must haves" pre-filled if they are not complex lookups (e.g. *Country* will have its *type* set to *True*, but we cannot do that with *Person*'s constraints).

2.9 Inheritance

> **Warning:** this section is out of date

Let's define another model:

```
class Woman(Person):
    class Meta:
        must_have = {'gender': 'female'}
```

Or even that one:

```
today = datetime.datetime.now()
day_16_years_back = now - datetime.timedelta(days=16*365)

class Child(Person):
    parent = Reference(Person)

    class Meta:
        must_have = {'birth_date__gte': day_16_years_back}
```

Note that our *Woman* or *Child* models are subclasses of *Person* model. They inherit all attributes of *Person*. Moreover, *Person*'s metaclass is inherited too. The *must_have* dictionaries of *Child* and *Woman* models are *merged* into the parent model's dictionary, so when we query the database for records described by the *Woman* model, we get all records that have *first_name* and *last_name* defined and *gender* set to "female". When we edit a *Person* instance, we do not care about the *parent* attribute; we actually don't even have access to it.

2.10 Model is a query, not a container

Warning: this section is out of date

We can even deal with data described above without model inheritance. Consider this valid model – *LivingBeing*:

```
class LivingBeing(Model):
    species = Property()
    birth_date = Property()

    class Meta:
        must_have = {'birth_date__exists': True}
```

The data described by *LivingBeing* overlaps the data described by *Person*. Some people have their birth dates not deifined and *Person* allows that. However, *LivingBeing* requires this attribute, so not all people will appear in a query by this model. At the same time *LivingBeing* does not require names, so anybody and anything, named or nameless, but ever born, is a "living being". Updating a record through any of these models will not touch data that the model does not know. For instance, saving an entity as a *LivingBeing* will not remove its name or parent, and working with it as a *Child* will neither expose nor destroy the information about species.

These examples illustrate how models are more "views" than "schemata".

Now let's try these models with a Tokyo Cabinet database:

```
>>> db = docu.get_db(
...     backend = 'docu.ext.tokyo_cabinet',
...     path = 'test.tct'
... )
>>> guido = Person(first_name='Guido', last_name='van Rossum')
>>> guido
<Person Guido van Rossum>
>>> guido.first_name
Guido
>>> guido.birth_date = datetime.date(1960, 1, 31)
```

```
>>> guido.save(db)     # returns the autogenerated primary key
'person_0'
>>> ppl_named_guido = Person.objects(db).where(first_name='Guido')
>>> ppl_named_guido
[<Person Guido van Rossum>]
>>> guido = ppl_named_guido[0]
>>> guido.age     # calculated on the fly -- datetime conversion works
49
>>> guido.birth_place = Country(name='Netherlands')
>>> guido.save()     # model instance already knows the storage it belongs to
'person_0'
>>> guido.birth_place
<Country Netherlands>
>>> Country.objects(db)     # yep, it was saved automatically with Guido
[<Country Netherlands>]
>>> larry = Person(first_name='Larry', last_name='Wall')
>>> larry.save(db)
'person_2'
>>> Person.objects(db)
[<Person Guido van Rossum>, <Person Larry Wall>]
```

...and so on.

Note that relations are supported out of the box.

CHAPTER

THREE

GLOSSARY

storage A place where data is stored. Provides a single namespace. Key/value stores can be represented with a single storage object, some other databases will require multiple storage objects (e.g. each "database" of CouchDB or each "collection" of MongoDB). Docu does not use nested namespaces because in document databases they mean nothing anyway.

Doqu offers a uniform API for different databases by providing "storage adapters". See *Backend API* for technical details and *Extensions* for a list of adapters bundled with Docu.

record A piece of data identified by an arbitrary unique primary key in a *storage*. In key/value stores the body of the record will be called "value" (usually serialized to a string); in other databases it is called "document" (also serialized as JSON, BSON, etc.). To avoid confusion we call all these things "records". In Python the record is represented as a dictionary of *fields*.

field A named property of a *record* or *document*. Records are actually containers for fields. There can be only one field with given name in the same record/document.

document An dictionary with metadata. Can be associated with a *record* in a *storage*. The structure can be restricted by *schema*. Optional *validators* determine how should the document look before it can be saved into the storage, or what records can be associated with documents of given class. Special behaviour can abe added with methods of the Document subclass (see *Document API*).

The simplest document is just a dictionary with some metadata. The metadata can be empty or contain information about where the document comes from, what does its *record* look like, etc.

A document without schema or validators is equal to its record. A document *with* schema is only equal to the record if they have the same sets of fields and these fields are valid (i.e. have correct data types and pass certain tests).

As you see, there is a difference between documents and records but sometimes it's very subtle.

schema A mapping of field names to Python data types. Prescribes the structure of a *document*.

validator Contains a certain test. When associated with a *field* of a *document*, determines whether given value is suitable for the field and, therefore, whether the document is valid in general. An invalid document cannot be saved to the *storage*. A validator can also contribute to the *document query*. See *Validators* for details on how this works.

document query A query that yields all *records* within given *storage* that can be associated with certain *document*. A document without *validators* does not add any conditions to the query, i.e. yields *all* records whatever structure they have. Validators can require that some fields are present or pass certain tests.

CHAPTER

FOUR

VALIDATORS

A validator simply takes an input and verifies it fulfills some criterion, such as a maximum length for a string. If the validation fails, a `ValidationError` is raised. This simple system allows chaining any number of validators on fields.

The module is heavily inspired by (and partially ripped off from) the WTForms validators. However, ours serve a bit different purpose. First, error messages are not needed here (the errors will not be displayed to end users). Second, these validators include **query filtering** capabilities.

Usage example:

```
class Person(Document):
    validators = {
        'first_name': [required(), length(min=2)],
        'age': [number_range(min=18)],
    }
```

This document will raise `ValidationError` if you attempt to save it with wrong values. You can call `Document.is_valid()` to ensure everything is OK.

Now let's query the database for all objects of *Person*:

```
Person.objects(db)
```

Doqu does not deal with tables or collections, it follows the DRY (Don't Repeat Yourself) principle and uses the same validators to determine what database records belong to given document class. The schema defined above is alone equivalent to the following query:

```
...where(first_name__exists=True, age__gte=18).where_not(first_name='')
```

This is actually the base query available as `Person.objects(db)`.

Note: not all validators affect document-related queries. See detailed documentation on each validator.

---

exception doqu.validators.**StopValidation**
Causes the validation chain to stop.

If StopValidation is raised, no more validators in the validation chain are called.

exception doqu.validators.**ValidationError**
Raised when a validator fails to validate its input.

class doqu.validators.**Email**
Validates an email address. Note that this uses a very primitive regular expression and should only be used in instances where you later verify by other means, such as email activation or lookups.

Adds conditions to the document-related queries: the field must match the pattern.

doqu.validators.**email**
alias of Email

class doqu.validators.**EqualTo**(*name*)
Compares the values of two fields.

Parameters **name** – The name of the other field to compare to.

doqu.validators.**equal_to**
alias of EqualTo

class doqu.validators.**Equals**(*other_value*)
Compares the value to another value.

Parameters **other_value** – The other value to compare to.

Adds conditions to the document-related queries.

doqu.validators.**equals**
alias of Equals

class doqu.validators.**Exists**
Ensures given field exists in the record. This does not affect validation of a document with pre-defined structure but does affect queries.

Adds conditions to the document-related queries.

doqu.validators.**exists**
alias of Exists

class doqu.validators.**IPAddress**
Validates an IP(v4) address.

Adds conditions to the document-related queries: the field must match the pattern.

doqu.validators.**ip_address**
alias of IPAddress

class doqu.validators.**Length**(*min=None*, *max=None*)
Validates the length of a string.

Parameters

- **min** – The minimum required length of the string. If not provided, minimum length will not be checked.
- **max** – The maximum length of the string. If not provided, maximum length will not be checked.

doqu.validators.**length**
alias of Length

class doqu.validators.**NumberRange**(*min=None*, *max=None*)
Validates that a number is of a minimum and/or maximum value, inclusive. This will work with any comparable number type, such as floats and decimals, not just integers.

Parameters

- **min** – The minimum required value of the number. If not provided, minimum value will not be checked.
- **max** – The maximum value of the number. If not provided, maximum value will not be checked.

Adds conditions to the document-related queries.

doqu.validators.**number_range**
alias of NumberRange

class doqu.validators.**Optional**
Allows empty value (i.e. `bool(value) == False`) and terminates the validation chain for this field (i.e. no more validators are applied to it). Note that errors raised prior to this validator are not suppressed.

doqu.validators.**optional**
alias of Optional

class doqu.validators.**Required**
Requires that the value is not empty, i.e. `bool(value)` returns *True*. The *bool* values can also be *False* (but not anything else).

Adds conditions to the document-related queries: the field must exist and be not equal to an empty string.

doqu.validators.**required**
alias of Required

class doqu.validators.**Regexp**(*pattern*, *flags=0*)
Validates the field against a user provided regexp.

Parameters

- **regex** – The regular expression string to use.
- **flags** – The regexp flags to use, for example *re.IGNORECASE* or *re.UNICODE*.

Note: the pattern must be provided as string because compiled patterns cannot be used in database lookups.

Adds conditions to the document-related queries: the field must match the pattern.

doqu.validators.**regexp**
alias of Regexp

class doqu.validators.**URL**(*require_tld=True*)
Simple regexp based url validation. Much like the email validator, you probably want to validate the url later by other means if the url must resolve.

Parameters **require_tld** – If true, then the domain-name portion of the URL must contain a .tld suffix. Set this to false if you want to allow domains like *localhost*.

Adds conditions to the document-related queries: the field must match the pattern.

doqu.validators.**url**
alias of URL

class doqu.validators.**AnyOf**(*choices*)
Compares the incoming data to a sequence of valid inputs.

Parameters **choices** – A sequence of valid inputs.

Adds conditions to the document-related queries.

doqu.validators.**any_of**
alias of AnyOf

class doqu.validators.**NoneOf**(*choices*)
Compares the incoming data to a sequence of invalid inputs.

Parameters **choices** – A sequence of invalid inputs.

Adds conditions to the document-related queries.

doqu.validators.**none_of**
alias of NoneOf

CHAPTER

FIVE

UTILITIES

Various useful functions. Some can be imported from `doqu.utils`, some are available directly at `doqu`.

These utilities are either stable and well-tested or possible changes in their API are not considered harmful (i.e. they are marginal). Important functions which design is likely to change or which lack proper tests are located in `doqu.future`.

`doqu.utils.`**`dump_doc`**(*self*, *raw=False*, *as_repr=False*, *align=True*, *keys=None*, *exclude=None*)

Returns a multi-line string with document keys and values nicely formatted and aligned.

Parameters

- **raw** – If *True*, uses "raw" values, as fetched from the database (note that this will fail for unsaved documents). If not, the values are obtained in the normal way, i.e. by *__getitem__()*. Default is *False*.
- **align** – If *True*, the keys and values are aligned into two columns of equal width. If *False*, no padding is used. Default is *True*.
- **keys** – a list of document keys to show. By default all existing keys are included.
- **exclude** – a list of keys to exclude. By default no keys are excluded.

Prarm as_repr If *True*, uses *repr()* for values; if not, coerces them to Unicode. Default if *False*.

`doqu.utils.`**`get_db`**(*settings_dict=None*, ***settings_kwargs*)

Storage adapter factory. Expects path to storage backend module and optional backend-specific settings. Returns storage adapter instance. If required underlying library is not found, exception *pkg_resources.DistributionNotFound* is raised with package name and version as the message.

Parameters **backend** – string, dotted path to a Doqu storage backend (e.g. *doqu.ext.tokyo_tyrant*). See *Extensions* for a list of bundled backends or *Backend API* for backend API reference.

Usage:

```
import doqu

db = doqu.get_db(backend='doqu.ext.shelve', path='test.db')

query = SomeDocument.objects(db)
```

Settings can be also passed as a dictionary:

```
SETTINGS = {
    'backend': 'doqu.ext.tokyo_cabinet',
    'path': 'test.tct',
}
```

```
db = doqu.get_db(SETTINGS)
```

The two methods can be combined to override certain settings:

```
db = doqu.get_db(SETTINGS, path='another_db.tct')
```

doqu.utils.**camel_case_to_underscores**(*class_name*)

Returns a pretty readable name based on the class name. For example, "SomeClass" is translated to "some_class".

doqu.utils.**load_fixture**(*path*, *db=None*)

Reads given file (assuming it is in a known format), loads it into given storage adapter instance and returns that instance.

Parameters

- **path** – absolute or relative path to the fixture file; user constructions ("~/foo") will be expanded.
- **db** – a storage adapter instance (its class must conform to the BaseStorageAdapter API). If not provided, a memory storage will be created.

Usage:

```
import doqu

db = doqu.load_fixture('account.csv')

query = SomeDocument.objects(db)
```

CHAPTER

SIX

EXTENSIONS

Doqu ships with some batteries included.

6.1 Database backends

6.1.1 Shelve extension

A storage/query backend for shelve which is bundled with Python.

status stable

database any dbm-style database supported by shelve

dependencies the Python standard library

suitable for "smart" interface to a key/value store, small volume

A "shelf" is a persistent, dictionary-like object. The difference with "dbm" databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects — anything that the pickle module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

This extension wraps the standard Python library and provides `Document` support and uniform query API.

Note: The query methods are inefficient as they involve iterating over the full set of records and making per-row comparison without indexing. This backend is not suitable for applications that depend on queries and require decent speed. However, it is an excellent tool for existing DBM databases or for environments and cases where external dependencies are not desired.

---

class doqu.ext.shelve_db.**StorageAdapter**(***kw*)
Provides unified Doqu API for MongoDB (see `doqu.backend_base.BaseStorageAdapter`).

Parameters **path** – relative or absolute path to the database file (e.g. *test.db*)

clear()
Clears the whole storage from data, resets autoincrement counters.

connect()
Connects to the database. Raises RuntimeError if the connection is not closed yet. Use `reconnect()` to explicitly close the connection and open it again.

delete(*key*)
Deletes record with given primary key.

disconnect()
Closes internal store and removes the reference to it. If the backend works with a file, then all pending changes are saved now.

find(*doc_class=<type 'dict'>*, ***conditions*)
Returns instances of given class, optionally filtered by given conditions.

Parameters

- **doc_class** – Document class. Default is *dict*. Normally you will want a more advanced class, such as `Document` or its more concrete subclasses (with explicit structure and validators).
- **conditions** – key/value pairs, same as in `where()`.

Note: By default this returns a tuple of `(key, data_dict)` per item. However, this can be changed if *doc_class* provides the method *from_storage()*. For example, `Document` has the notion of "saved state" so it can store the key within. Thus, only a single *Document* object is returned per item.

---

get(*key*, *doc_class=<type 'dict'>*)
Returns document instance for given document class and primary key. Raises KeyError if there is no item with given key in the database.

Parameters

- **key** – a numeric or string primary key (as supported by the backend).
- **doc_class** – a document class to wrap the data into. Default is *dict*.

get_many(*keys*, *doc_class=<type 'dict'>*)
Returns an iterator of documents with primary keys from given list. Basically this is just a simple wrapper around `get()` but some backends can reimplement the method in a much more efficient way.

get_or_create(*doc_class=<type 'dict'>*, ***conditions*)
Queries the database for records associated with given document class and conforming to given extra conditions. If such records exist, picks the first one (the order may be random depending on the database). If there are no such records, creates one.

Returns the document instance and a boolean value "created".

query_adapter
alias of `QueryAdapter`

reconnect()
Gracefully closes current connection (if it's not broken) and connects again to the database (e.g. reopens the file).

save(*key*, *data*)
Saves given data with given primary key into the storage. Returns the primary key.

Parameters

- **key** – the primary key for given object; if *None*, will be generated.
- **data** – a *dict* containing all properties to be saved.

Note that you must provide current primary key for a record which is already in the database in order to update it instead of copying it.

sync()
Synchronizes the storage to disk immediately if the backend supports this operation. Normally the data is

synchronized either on `save()`, or on timeout, or on `disconnect()`. This is strictly backend-specific. If a backend does not support the operation, *NotImplementedError* is raised.

class doqu.ext.shelve_db.**QueryAdapter**(**args*, ***kw*)

The Query class.

count()

Same as `__len__` but a bit faster.

delete()

Deletes all records that match current query.

order_by(*names*, *reverse=False*)

Defines order in which results should be retrieved.

Parameters

- **names** – the names of columns by which the ordering should be done. Can be an iterable with strings or a single string.
- **reverse** – If *True*, direction changes from ascending (default) to descending.

Examples:

```
q.order_by('name')                  # ascending
q.order_by('name', reverse=True)    # descending
```

If multiple names are provided, grouping is done from left to right.

Note: while you can specify the direction of sorting, it is not possible to do it on per-name basis due to backend limitations.

Warning: ordering implementation for this database is currently inefficient.

values(*name*)

Returns an iterator that yields distinct values for given column name.

Supports date parts (i.e. *date__month=7*).

Note: this is currently highly inefficient because the underlying library does not support columns mode (*tctdbiternext3*). Moreover, even current implementation can be optimized by removing the overhead of creating full-blown document objects.

Note: unhashable values (like lists) are silently ignored.

---

where(***conditions*)

Returns Query instance filtered by given conditions. The conditions are specified by backend's underlying API.

where_not(***conditions*)

Returns Query instance. Inverted version of *where()*.

6.1.2 Shove extension

A storage/query backend for shove which is bundled with Python.

status beta

database any supported by shove: *storage* — Amazon S3 Web Service, Berkeley Source Database, Filesystem, Firebird, FTP, DBM, Durus, Memory, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, SQLite, Subversion, Zope Object Database (ZODB); *caching* — Filesystem, Firebird, memcached, Memory, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, SQLite

dependencies shove

suitable for "smart" interface to a key/value store; temporary memory storage

This extension wraps the *shove* library and provides the uniform query API along with support for `Document` API.

Note: Regardless of the underlying storage, Shove serializes the records and only offers access by primary key. This means that efficient queries are impossible even with RDBMS; moreover, such databases are more likely to perform slower than simple key/value stores. The *Docu* queries with Shove involve iterating over the full set of records on client side and making per-row comparison without proper indexing.

That said, the backend is considered not suitable for applications that depend on queries and require decent speed of lookups by value. However, it can be very useful as a memory storage (e.g. to analyze a JSON dump or calculate some data on the fly) or as an improved interface to an existing pure key/value storage which is mostly used without advanced queries.

---

class `doqu.ext.shove_db.`**`StorageAdapter`**`(`***kw*`)`
All parametres are optional. Here are the most common:

Parameters

- **store_uri** – URI for the data store
- **cache_uri** – URI for the caching instance

The URI format for a backend is documented in its module (see the shove documentation). The URI form is the same as SQLAlchemy's.

`clear``()`
Clears the whole storage from data, resets autoincrement counters.

`connect``()`
Connects to the database. Raises RuntimeError if the connection is not closed yet. Use `reconnect()` to explicitly close the connection and open it again.

`delete``(`*key*`)`
Deletes record with given primary key.

`disconnect``()`
Closes internal store and removes the reference to it. If the backend works with a file, then all pending changes are saved now.

`find``(`*doc_class=<type 'dict'>*, ***conditions*`)`
Returns instances of given class, optionally filtered by given conditions.

Parameters

- **doc_class** – Document class. Default is *dict*. Normally you will want a more advanced class, such as `Document` or its more concrete subclasses (with explicit structure and validators).
- **conditions** – key/value pairs, same as in `where()`.

Note: By default this returns a tuple of `(key, data_dict)` per item. However, this can be changed if *doc_class* provides the method *from_storage()*. For example, `Document` has the notion of "saved state" so it can store the key within. Thus, only a single *Document* object is returned per item.

---

get(*key*, *doc_class=<type 'dict'>*)
Returns document instance for given document class and primary key. Raises KeyError if there is no item with given key in the database.

Parameters

- **key** – a numeric or string primary key (as supported by the backend).
- **doc_class** – a document class to wrap the data into. Default is *dict*.

get_many(*keys*, *doc_class=<type 'dict'>*)
Returns an iterator of documents with primary keys from given list. Basically this is just a simple wrapper around `get()` but some backends can reimplement the method in a much more efficient way.

get_or_create(*doc_class=<type 'dict'>*, ***conditions*)
Queries the database for records associated with given document class and conforming to given extra conditions. If such records exist, picks the first one (the order may be random depending on the database). If there are no such records, creates one.

Returns the document instance and a boolean value "created".

query_adapter
alias of `QueryAdapter`

reconnect()
Gracefully closes current connection (if it's not broken) and connects again to the database (e.g. reopens the file).

save(*key*, *data*)
Saves given data with given primary key into the storage. Returns the primary key.

Parameters

- **key** – the primary key for given object; if *None*, will be generated.
- **data** – a *dict* containing all properties to be saved.

Note that you must provide current primary key for a record which is already in the database in order to update it instead of copying it.

sync()
Synchronizes the storage to disk immediately if the backend supports this operation. Normally the data is synchronized either on `save()`, or on timeout, or on `disconnect()`. This is strictly backend-specific. If a backend does not support the operation, *NotImplementedError* is raised.

class doqu.ext.shove_db.**QueryAdapter**(**args*, ***kw*)
The Query class.

count()
Same as `__len__` but a bit faster.

delete()
Deletes all records that match current query.

order_by(*names*, *reverse=False*)
Defines order in which results should be retrieved.

Parameters

- **names** – the names of columns by which the ordering should be done. Can be an iterable with strings or a single string.
- **reverse** – If *True*, direction changes from ascending (default) to descending.

Examples:

```
q.order_by('name')                  # ascending
q.order_by('name', reverse=True)    # descending
```

If multiple names are provided, grouping is done from left to right.

Note: while you can specify the direction of sorting, it is not possible to do it on per-name basis due to backend limitations.

Warning: ordering implementation for this database is currently inefficient.

values (*name*)
Returns an iterator that yields distinct values for given column name.

Supports date parts (i.e. *date__month=7*).

Note: this is currently highly inefficient because the underlying library does not support columns mode (*tctdbiternext3*). Moreover, even current implementation can be optimized by removing the overhead of creating full-blown document objects.

Note: unhashable values (like lists) are silently ignored.

---

where (***conditions*)
Returns Query instance filtered by given conditions. The conditions are specified by backend's underlying API.

where_not (***conditions*)
Returns Query instance. Inverted version of *where()*.

6.1.3 Tokyo Cabinet extension

A storage/query backend for Tokyo Cabinet.

Allows direct access to the database and is thus extremely fast. However, it locks the database and is therefore not suitable for environments where concurrent access is required. Please use Tokyo Tyrant for such environments.

status beta

database Tokyo Cabinet

dependencies tokyo-python, pyrant

suitable for general purpose, embedded

Warning: this module is not intended for production despite it *may* be stable. Bug reports and patches are welcome.

Note: this module should not depend on Pyrant; just needs some refactoring.

Note: support for metasearch is planned.

Usage:

```
>>> import os
>>> import doqu
>>> DB_SETTINGS = {
...     'backend': 'doqu.ext.tokyo_cabinet',
...     'path': '_tc_test.tct',
... }
>>> assert not os.path.exists(DB_SETTINGS['path']), 'test database must not exist'
>>> db = doqu.get_db(DB_SETTINGS)
>>> class Person(doqu.Document):
...     structure = {'name': unicode}
...     def __unicode__(self):
...         u'%(name)s' % self
...
>>> Person.objects(db)    # the database is expected to be empty
[]
>>> db.connection['john'] = {'name': 'John'}
>>> mary = Person(name='Mary')
>>> mary_pk = mary.save(db)
>>> q = Person.objects(db)
>>> q
[<Person John>, <Person Mary>]
>>> q.where(name__matches='^J')
[<Person John>]
>>> q    # the original query was not modified by the descendant
[<Person John>, <Person Mary>]
>>> db.connection.close()
>>> os.unlink(DB_SETTINGS['path'])
```

class doqu.ext.tokyo_cabinet.**StorageAdapter**(***kw*)

Parameters **path** – relative or absolute path to the database file (e.g. *test.tct*)

Note: Currently only *table* flavour of Tokyo Cabinet databases is supported. It is uncertain whether it is worth supporting other flavours as they do not provide query mechanisms other than access by primary key.

clear()
Clears the whole storage from data, resets autoincrement counters.

connect()
Connects to the database. Raises RuntimeError if the connection is not closed yet. Use `reconnect()` to explicitly close the connection and open it again.

delete(*key*)
Deletes record with given primary key.

disconnect()
Closes internal store and removes the reference to it. If the backend works with a file, then all pending changes are saved now.

find(*doc_class=<type 'dict'>*, ***conditions*)
Returns instances of given class, optionally filtered by given conditions.

Parameters

- **doc_class** – Document class. Default is *dict*. Normally you will want a more advanced class, such as `Document` or its more concrete subclasses (with explicit structure and validators).
- **conditions** – key/value pairs, same as in `where()`.

Note: By default this returns a tuple of `(key, data_dict)` per item. However, this can be changed if *doc_class* provides the method *from_storage()*. For example, `Document` has the notion of "saved state" so it can store the key within. Thus, only a single *Document* object is returned per item.

---

get(*key*, *doc_class=<type 'dict'>*)
Returns document instance for given document class and primary key. Raises KeyError if there is no item with given key in the database.

Parameters

- **key** – a numeric or string primary key (as supported by the backend).
- **doc_class** – a document class to wrap the data into. Default is *dict*.

get_many(*keys*, *doc_class=<type 'dict'>*)
Returns an iterator of documents with primary keys from given list. Basically this is just a simple wrapper around `get()` but some backends can reimplement the method in a much more efficient way.

get_or_create(*doc_class=<type 'dict'>*, ***conditions*)
Queries the database for records associated with given document class and conforming to given extra conditions. If such records exist, picks the first one (the order may be random depending on the database). If there are no such records, creates one.

Returns the document instance and a boolean value "created".

query_adapter
alias of `QueryAdapter`

reconnect()
Gracefully closes current connection (if it's not broken) and connects again to the database (e.g. reopens the file).

save(*key*, *data*)
Saves given data with given primary key into the storage. Returns the primary key.

Parameters

- **key** – the primary key for given object; if *None*, will be generated.
- **data** – a *dict* containing all properties to be saved.

Note that you must provide current primary key for a record which is already in the database in order to update it instead of copying it.

sync()
Synchronizes the storage to disk immediately if the backend supports this operation. Normally the data is synchronized either on `save()`, or on timeout, or on `disconnect()`. This is strictly backend-specific. If a backend does not support the operation, *NotImplementedError* is raised.

class doqu.ext.tokyo_cabinet.**QueryAdapter**(**args*, ***kw*)
The Query class.

count()
Same as `__len__` but without fetching the records (i.e. faster).

delete()
Deletes all records that match current query.

order_by(*names*, *reverse=False*)
Returns a query object with same conditions but with results sorted by given field. By default the direction of sorting is ascending.

Parameters

- **names** – list of strings: names of fields by which results should be sorted. Some backends may only support a single field for sorting.
- **reverse** – *bool*: if *True*, the direction of sorting is reversed and becomes descending. Default is *False*.

values(*name*)
Returns an iterator that yields distinct values for given column name.

Note: this is currently highly inefficient because the underlying library does not support columns mode (*tctdbiternext3*). Moreover, even current implementation can be optimized by removing the overhead of creating full-blown document objects (though preserving data type is necessary).

---

where(***conditions*)
Returns Query instance filtered by given conditions. The conditions are specified by backend's underlying API.

where_not(***conditions*)
Returns Query instance. Inverted version of *where()*.

6.1.4 Tokyo Tyrant extension

A storage/query backend for Tokyo Tyrant.

status stable

database Tokyo Cabinet, Tokyo Tyrant

dependencies Pyrant

suitable for general purpose

class doqu.ext.tokyo_tyrant.storage.**StorageAdapter**(***kw*)

clear()
Clears the whole storage from data, resets autoincrement counters.

connect()
Connects to the database. Raises RuntimeError if the connection is not closed yet. Use `reconnect()` to explicitly close the connection and open it again.

delete(*key*)
Deletes record with given primary key.

disconnect()
Closes internal store and removes the reference to it. If the backend works with a file, then all pending changes are saved now.

find(*doc_class=<type 'dict'>*, ***conditions*)
Returns instances of given class, optionally filtered by given conditions.

Parameters

- **doc_class** – Document class. Default is *dict*. Normally you will want a more advanced class, such as `Document` or its more concrete subclasses (with explicit structure and validators).
- **conditions** – key/value pairs, same as in `where()`.

Note: By default this returns a tuple of `(key, data_dict)` per item. However, this can be changed if *doc_class* provides the method *from_storage()*. For example, `Document` has the notion of "saved state" so it can store the key within. Thus, only a single *Document* object is returned per item.

---

get(*key*, *doc_class=<type 'dict'>*)
Returns document instance for given document class and primary key. Raises KeyError if there is no item with given key in the database.

Parameters

- **key** – a numeric or string primary key (as supported by the backend).
- **doc_class** – a document class to wrap the data into. Default is *dict*.

get_many(*keys*, *doc_class=<type 'dict'>*)
Returns an iterator of documents with primary keys from given list. Basically this is just a simple wrapper around `get()` but some backends can reimplement the method in a much more efficient way.

get_or_create(*doc_class=<type 'dict'>*, ***conditions*)
Queries the database for records associated with given document class and conforming to given extra conditions. If such records exist, picks the first one (the order may be random depending on the database). If there are no such records, creates one.

Returns the document instance and a boolean value "created".

reconnect()
Gracefully closes current connection (if it's not broken) and connects again to the database (e.g. reopens the file).

save(*key*, *data*)
Saves given data with given primary key into the storage. Returns the primary key.

Parameters

- **key** – the primary key for given object; if *None*, will be generated.
- **data** – a *dict* containing all properties to be saved.

Note that you must provide current primary key for a record which is already in the database in order to update it instead of copying it.

sync()
Synchronizes the storage to disk immediately if the backend supports this operation. Normally the data is synchronized either on `save()`, or on timeout, or on `disconnect()`. This is strictly backend-specific. If a backend does not support the operation, *NotImplementedError* is raised.

class doqu.ext.tokyo_tyrant.query.**QueryAdapter**(*storage*, *doc_class*)

count()
Returns the number of records that match current query. Does not fetch the records.

delete()
Deletes all records that match current query.

order_by(*names*, *reverse=False*)
Returns a query object with same conditions but with results sorted by given field. By default the direction of sorting is ascending.

Parameters

- **names** – list of strings: names of fields by which results should be sorted. Some backends may only support a single field for sorting.
- **reverse** – *bool*: if *True*, the direction of sorting is reversed and becomes descending. Default is *False*.

values(*name*)
Returns a list of unique values for given column name.

where(***conditions*)
Returns Query instance filtered by given conditions.

where_not(***conditions*)
Returns Query instance. Inverted version of `where()`.

6.1.5 MongoDB extension

A storage/query backend for MongoDB.

status beta

database MongoDB

dependencies pymongo

suitable for general purpose (mostly server-side)

Warning: this module is not intended for production. It contains some hacks and should be refactored. However, it is actually used in a real project involving complex queries. Patches, improvements, rewrites are welcome.

class doqu.ext.mongodb.**StorageAdapter**(***kw*)

Parameters

- **host** –
- **port** –
- **database** –
- **collection** –

clear()
Clears the whole storage from data, resets autoincrement counters.

connect()
Connects to the database. Raises RuntimeError if the connection is not closed yet. Use `reconnect()` to explicitly close the connection and open it again.

delete(*key*)
Deletes record with given primary key.

disconnect()
Closes internal store and removes the reference to it. If the backend works with a file, then all pending changes are saved now.

find(*doc_class=<type 'dict'>*, ***conditions*)
Returns instances of given class, optionally filtered by given conditions.

Parameters

- **doc_class** – Document class. Default is *dict*. Normally you will want a more advanced class, such as `Document` or its more concrete subclasses (with explicit structure and validators).
- **conditions** – key/value pairs, same as in `where()`.

Note: By default this returns a tuple of `(key, data_dict)` per item. However, this can be changed if *doc_class* provides the method *from_storage()*. For example, `Document` has the notion of "saved state" so it can store the key within. Thus, only a single *Document* object is returned per item.

---

get(*key*, *doc_class=<type 'dict'>*)
Returns document instance for given document class and primary key. Raises KeyError if there is no item with given key in the database.

Parameters

- **key** – a numeric or string primary key (as supported by the backend).
- **doc_class** – a document class to wrap the data into. Default is *dict*.

get_many(*keys*, *doc_class=<type 'dict'>*)
Returns an iterator of documents with primary keys from given list. Basically this is just a simple wrapper around `get()` but some backends can reimplement the method in a much more efficient way.

get_or_create(*doc_class=<type 'dict'>*, ***conditions*)
Queries the database for records associated with given document class and conforming to given extra conditions. If such records exist, picks the first one (the order may be random depending on the database). If there are no such records, creates one.

Returns the document instance and a boolean value "created".

reconnect()
Gracefully closes current connection (if it's not broken) and connects again to the database (e.g. reopens the file).

save(*key*, *data*)
Saves given data with given primary key into the storage. Returns the primary key.

Parameters

- **key** – the primary key for given object; if *None*, will be generated.
- **data** – a *dict* containing all properties to be saved.

Note that you must provide current primary key for a record which is already in the database in order to update it instead of copying it.

sync()
Synchronizes the storage to disk immediately if the backend supports this operation. Normally the data is synchronized either on `save()`, or on timeout, or on `disconnect()`. This is strictly backend-specific. If a backend does not support the operation, *NotImplementedError* is raised.

class doqu.ext.mongodb.**QueryAdapter**(**args*, ***kw*)

count()
Returns the number of records that match given query. The result of *q.count()* is exactly equivalent to the result of *len(q)* but does not involve fetching of the records.

delete()
Deletes all records that match current query.

order_by(*names*, *reverse=False*)
Returns a query object with same conditions but with results sorted by given field. By default the direction of sorting is ascending.

Parameters

- **names** – list of strings: names of fields by which results should be sorted. Some backends may only support a single field for sorting.
- **reverse** – *bool*: if *True*, the direction of sorting is reversed and becomes descending. Default is *False*.

values(*name*)
Returns a list of unique values for given field name.

Parameters **name** – the field name.

Note: A set is dynamically build on client side if the query contains conditions. If it doesn't, a much more efficient approach is used. It is only available within current **connection**, not query.

---

where(**conditions*)
Returns Query instance filtered by given conditions. The conditions are specified by backend's underlying API.

where_not(**conditions*)
Returns Query instance. Inverted version of *where()*.

6.2 Convenience abstractions

6.2.1 Document Fields

New in version 0.23.

Note: This abstraction is by no means a complete replacement for the normal approach of semantic grouping. Please use it with care. Also note that the API can change. The class can even be removed in future versions of Doqu.

---

class doqu.ext.fields.**Field**(*datatype*, *essential=False*, *required=False*, *default=None*, *choices=None*, *label=None*, *pickled=False*)
Representation of a document property. Syntax sugar for separate definitions of structure, validators, defaults and labels.

Usage:

```
class Book(Document):
    title = Field(unicode, required=True, default=u'Hi', label='Title')
```

this is just another way to type:

```
class Book(Document):
    structure = {
        'title': unicode
    }
    validators = {
        'title': [validators.Required()]
    }
    defaults = {
        'title': u'Hi'
    }
    labels = {
        'title': u'The Title'
    }
```

Nice, eh? But be careful: the *title* definition in the first example barely fits its line. Multiple long definitions will turn your document class into an opaque mess of characters, while the semantically grouped definitions stay short and keep related things aligned together. "Semantic sugar" is sometimes pretty bitter, use it with care.

Complex validators still need to be specified by hand in the relevant dictionary. This can be worked around by creating specialized field classes (e.g. *EmailField*) as it is done e.g. in Django.

Parameters

- **essential** – if True, validator `Exists` is added (i.e. the field may be empty but it must be present in the record).
- **pickled** – if True, the value is preprocessed with pickle's dumps/loads functions. This of course breaks lookups by this field but enables storing arbitrary Python objects.

class doqu.ext.fields.**FileField**(*base_path*, ***kwargs*)

Handles externally stored files.

> **Warning:** This field saves the file when `process_outgoing()` is triggered (see *outgoing_processors* in `DocumentMetadata`).
> Outdated (replaced) files are *not* automatically removed.

Usage:

```
class Doc(Document):
    attachment = FileField(base_path=MEDIA_ROOT+'attachments/')

d = Doc()
d.attachment = open('foo.txt')
d.save(db)

dd = Doc.objects(db)[0]
print dd.attachment.file.read()
```

Parameters **base_path** – A string or callable: the directory where the files should be stored.

file_wrapper_class

alias of `FileWrapper`

class doqu.ext.fields.**ImageField**(*base_path*, ***kwargs*)

A `FileField` that provides extended support for images. The `ImageField.file` is an `ImageWrapper` instance.

Usage:

```
class Photo(Document):
    summary = Field(unicode)
    image = ImageField(base_path='photos/')

p = Photo(summary='Fido', image=open('fido.jpg'))
p.save(db)

# playing with image
print "The photo is {0}×{1}px".format(*p.image.size)
p.image.rotate(90)
p.image.save()
```

file_wrapper_class
alias of ImageWrapper

6.3 Integration with other libraries

6.3.1 WTForms extension

Offers integration with WTForms.

status beta

dependencies wtforms

The extension provides two new field classes: `QuerySetSelectField` and `DocumentSelectField` (inspired by *wtforms.ext.django.**). They connect the forms with the Doqu API for queries. You can manually create forms with these fields.

The easiest way to create a `Document`-compliant form is using the function `document_form_factory()`. It returns a form class based on the document structure:

```
from doqu import Document
from doqu import validators
from doqu.ext.forms import document_form_factory

class Location(Document):
    structure = {'name': unicode}

class Person(Document):
    structure = {'name': unicode, 'age': int, 'location': Location}
    labels = {'name': 'Full name', 'age': 'Age', 'location': 'Location'}
    validators = {'name': [required()]}

PersonForm = document_form_factory(Person)
```

The last line does the same as this code:

```
from wtforms import TextField, IntegerField, validators
from doqu.ext.forms import DocumentSelectField

class PersonForm(wtforms.Form):
    name = TextField('Full name', [validators.Required()])
    age = IntegerField('Age')
    location = DocumentSelectField('Location', [], Location)
```

doqu.ext.forms.**document_form_factory**(*document_class*, *storage=None*)

Expects a Document instance, creates and returns a wtforms.Form class for this model.

The form fields are selected depending on the Python type declared by each property.

Parameters

- **document_class** – the Doqu document class for which the form should be created
- **storage** – a Doqu-compatible storage; we need it to generate lists of choices for references to other models. If not defined, references will not appear in the form.

Caveat: the unicode type can be mapped to TextField and TextAreaField. It is impossible to guess which one should be used unless maximum length is defined for the property. TextAreaField is picked by default. It is a good idea to automatically shrink it with JavaScript so that its size always matches the contents.

class doqu.ext.forms.**QuerySetSelectField**(*label=u''*, *validators=None*, *queryset=None*, *label_attr=''*, *allow_blank=False*, *blank_text=u''*, ***kw*)

Given a QuerySet either at initialization or inside a view, will display a select drop-down field of choices. The *data* property actually will store/keep an ORM model instance, not the ID. Submitting a choice which is not in the queryset will result in a validation error.

Specifying *label_attr* in the constructor will use that property of the model instance for display in the list, else the model object's *__str__* or *__unicode__* will be used.

If *allow_blank* is set to *True*, then a blank choice will be added to the top of the list. Selecting this choice will result in the *data* property being *None*. The label for the blank choice can be set by specifying the *blank_text* parameter.

populate_obj(*obj*, *name*)

Populates *obj.<name>* with the field's data.

Note This is a destructive operation. If *obj.<name>* already exists, it will be overridden. Use with caution.

post_validate(*form*, *validation_stopped*)

Override if you need to run any field-level validation tasks after normal validation. This shouldn't be needed in most cases.

Parameters

- **form** – The form the field belongs to.
- **validation_stopped** – *True* if any validator raised StopValidation.

process(*formdata*, *data=<object object at 0x283e1d0>*)

Process incoming data, calling process_data, process_formdata as needed, and run filters.

If *data* is not provided, process_data will be called on the field's default.

Field subclasses usually won't override this, instead overriding the process_formdata and process_data methods. Only override this for special advanced processing, such as when a field encapsulates many inputs.

process_data(*value*)

Process the Python data applied to this field and store the result.

This will be called during form construction by the form's *kwargs* or *obj* argument.

Parameters **value** – The python object containing the value to process.

validate(*form*, *extra_validators=()*)

Validates the field and returns True or False. *self.errors* will contain any errors raised during validation. This is usually only called by *Form.validate*.

Subfields shouldn't override this, but rather override either *pre_validate*, *post_validate* or both, depending on needs.

Parameters

- **form** – The form the field belongs to.
- **extra_validators** – A list of extra validators to run.

class doqu.ext.forms.**DocumentSelectField**(*label=u''*, *validators=None*, *document_class=None*, *storage=None*, ***kw*)

Like a QuerySetSelectField, except takes a document class instead of a queryset and lists everything in it.

populate_obj(*obj*, *name*)

Populates *obj.<name>* with the field's data.

Note This is a destructive operation. If *obj.<name>* already exists, it will be overridden. Use with caution.

post_validate(*form*, *validation_stopped*)

Override if you need to run any field-level validation tasks after normal validation. This shouldn't be needed in most cases.

Parameters

- **form** – The form the field belongs to.
- **validation_stopped** – *True* if any validator raised StopValidation.

process(*formdata*, *data=<object object at 0x283e1d0>*)

Process incoming data, calling process_data, process_formdata as needed, and run filters.

If *data* is not provided, process_data will be called on the field's default.

Field subclasses usually won't override this, instead overriding the process_formdata and process_data methods. Only override this for special advanced processing, such as when a field encapsulates many inputs.

process_data(*value*)

Process the Python data applied to this field and store the result.

This will be called during form construction by the form's *kwargs* or *obj* argument.

Parameters **value** – The python object containing the value to process.

validate(*form*, *extra_validators=()*)

Validates the field and returns True or False. *self.errors* will contain any errors raised during validation. This is usually only called by *Form.validate*.

Subfields shouldn't override this, but rather override either *pre_validate*, *post_validate* or both, depending on needs.

Parameters

- **form** – The form the field belongs to.
- **extra_validators** – A list of extra validators to run.

CHAPTER

SEVEN

API REFERENCE

7.1 Document API

Documents represent database records. Each document is a (in)complete subset of *fields* contained in a *record*. Available data types and query mechanisms are determined by the *storage* in use.

The API was inspired by Django, MongoKit, WTForms, Svarga and several other projects. It was important to KISS (keep it simple, stupid), DRY (do not repeat yourself) and to make the API as abstract as possible so that it did not depend on backends and yet did not get in the way.

class doqu.document_base.**Document**(**kw*)

A document/query object. Dict-like representation of a document stored in a database. Includes schema declaration, bi-directional validation (outgoing and query), handles relations and has the notion of the saved state, i.e. knows the storage and primary key of the corresponding record.

classmethod contribute_to_query(*query*)

Returns given query filtered by schema and validators defined for this document.

delete()

Deletes the object from the associated storage.

classmethod from_storage(*storage*, *key*, *data*)

Returns a document instance filled with given *data* and bound to given *storage* and *key*. The instance can be safely saved back using `save()`. If the concrete subclass defines the structure, then usused fields coming from the storage are hidden from the public API but nevertheless they will be saved back to the database as is.

pk

Returns current primary key (if any) or None.

save(*storage=None*, *keep_key=False*)

Saves instance to given storage.

Parameters

- **storage** – the storage to which the document should be saved. If not specified, default storage is used (the one from which the document was retrieved of to which it this instance was saved before).
- **keep_key** – if *True*, the primary key is preserved even when saving to another storage. This is potentially dangerous because existing unrelated records can be overwritten. You will only *need* this when copying a set of records that reference each other by primary key. Default is *False*.

validate()
Checks if instance data is valid. This involves a) checking whether all values correspond to the declated structure, and b) running all *Validators* against the data dictionary.

Raises `ValidationError` if something is wrong.

Note: if the data dictionary does not contain some items determined by structure or validators, these items are *not* checked.

Note: The document is checked as is. There are no side effects. That is, if some required values are empty, they will be considered invalid even if default values are defined for them. The `save()` method, however, fills in the default values before validating.

---

doqu.document_base.**Many**
alias of `OneToManyRelation`

7.2 Document Fields

New in version 0.23.

Note: This abstraction is by no means a complete replacement for the normal approach of semantic grouping. Please use it with care. Also note that the API can change. The class can even be removed in future versions of Doqu.

---

class doqu.ext.fields.**Field**(*datatype*, *essential=False*, *required=False*, *default=None*, *choices=None*, *label=None*, *pickled=False*)
Representation of a document property. Syntax sugar for separate definitions of structure, validators, defaults and labels.

Usage:

```
class Book(Document):
    title = Field(unicode, required=True, default=u'Hi', label='Title')
```

this is just another way to type:

```
class Book(Document):
    structure = {
        'title': unicode
    }
    validators = {
        'title': [validators.Required()]
    }
    defaults = {
        'title': u'Hi'
    }
    labels = {
        'title': u'The Title'
    }
```

Nice, eh? But be careful: the *title* definition in the first example barely fits its line. Multiple long definitions will turn your document class into an opaque mess of characters, while the semantically grouped definitions stay short and keep related things aligned together. "Semantic sugar" is sometimes pretty bitter, use it with care.

Complex validators still need to be specified by hand in the relevant dictionary. This can be worked around by creating specialized field classes (e.g. *EmailField*) as it is done e.g. in Django.

Parameters

- **essential** – if True, validator `Exists` is added (i.e. the field may be empty but it must be present in the record).
- **pickled** – if True, the value is preprocessed with pickle's dumps/loads functions. This of course breaks lookups by this field but enables storing arbitrary Python objects.

class doqu.ext.fields.**FileField**(*base_path*, ***kwargs*)

Handles externally stored files.

> **Warning:** This field saves the file when `process_outgoing()` is triggered (see *outgoing_processors* in `DocumentMetadata`).
> Outdated (replaced) files are *not* automatically removed.

Usage:

```
class Doc(Document):
    attachment = FileField(base_path=MEDIA_ROOT+'attachments/')

d = Doc()
d.attachment = open('foo.txt')
d.save(db)

dd = Doc.objects(db)[0]
print dd.attachment.file.read()
```

Parameters **base_path** – A string or callable: the directory where the files should be stored.

file_wrapper_class

alias of `FileWrapper`

class doqu.ext.fields.**ImageField**(*base_path*, ***kwargs*)

A `FileField` that provides extended support for images. The `ImageField.file` is an `ImageWrapper` instance.

Usage:

```
class Photo(Document):
    summary = Field(unicode)
    image = ImageField(base_path='photos/')

p = Photo(summary='Fido', image=open('fido.jpg'))
p.save(db)

# playing with image
print "The photo is {0}×{1}px".format(*p.image.size)
p.image.rotate(90)
p.image.save()
```

file_wrapper_class

alias of `ImageWrapper`

7.3 Backend API

Abstract classes for unified storage/query API with various backends.

Derivative classes are expected to be either complete implementations or wrappers for external libraries. The latter is assumed to be a better solution as Doqu is only one of the possible layers. It is always a good idea to provide different levels of abstraction and let others combine them as needed.

The backends do not have to subclass `BaseStorageAdapter` and `BaseQueryAdapter`. However, they must closely follow their API.

class `doqu.backend_base.`**`BaseStorageAdapter`**(***kw*)

Abstract adapter class for storage backends.

Note: Backends policy

If a public method *foo()* internally uses a private method *_foo()*, then subclasses should only overload only the private attribute. This ensures that docstring and signature are always correct. However, if the backend introduces some deviations in behaviour or extends the signature, the public method can (and should) be overloaded at least to provide documentation.

`clear`()

Clears the whole storage from data, resets autoincrement counters.

`connect`()

Connects to the database. Raises RuntimeError if the connection is not closed yet. Use `reconnect()` to explicitly close the connection and open it again.

`delete`(*key*)

Deletes record with given primary key.

`disconnect`()

Closes internal store and removes the reference to it. If the backend works with a file, then all pending changes are saved now.

`find`(*doc_class=<type 'dict'>*, ***conditions*)

Returns instances of given class, optionally filtered by given conditions.

Parameters

- **doc_class** – Document class. Default is *dict*. Normally you will want a more advanced class, such as `Document` or its more concrete subclasses (with explicit structure and validators).
- **conditions** – key/value pairs, same as in `where()`.

Note: By default this returns a tuple of `(key, data_dict)` per item. However, this can be changed if *doc_class* provides the method *from_storage()*. For example, `Document` has the notion of "saved state" so it can store the key within. Thus, only a single *Document* object is returned per item.

`get`(*key*, *doc_class=<type 'dict'>*)

Returns document instance for given document class and primary key. Raises KeyError if there is no item with given key in the database.

Parameters

- **key** – a numeric or string primary key (as supported by the backend).
- **doc_class** – a document class to wrap the data into. Default is *dict*.

get_many(*keys*, *doc_class=<type 'dict'>*)
Returns an iterator of documents with primary keys from given list. Basically this is just a simple wrapper around `get()` but some backends can reimplement the method in a much more efficient way.

get_or_create(*doc_class=<type 'dict'>*, ***conditions*)
Queries the database for records associated with given document class and conforming to given extra conditions. If such records exist, picks the first one (the order may be random depending on the database). If there are no such records, creates one.

Returns the document instance and a boolean value "created".

reconnect()
Gracefully closes current connection (if it's not broken) and connects again to the database (e.g. reopens the file).

save(*key*, *data*)
Saves given data with given primary key into the storage. Returns the primary key.

Parameters

- **key** – the primary key for given object; if *None*, will be generated.
- **data** – a *dict* containing all properties to be saved.

Note that you must provide current primary key for a record which is already in the database in order to update it instead of copying it.

sync()
Synchronizes the storage to disk immediately if the backend supports this operation. Normally the data is synchronized either on `save()`, or on timeout, or on `disconnect()`. This is strictly backend-specific. If a backend does not support the operation, *NotImplementedError* is raised.

class doqu.backend_base.**BaseQueryAdapter**(*storage*, *doc_class*)
Query adapter for given backend.

count()
Returns the number of records that match given query. The result of *q.count()* is exactly equivalent to the result of *len(q)*. The implementation details do not differ by default, but it is recommended that the backends stick to the following convention:

- *__len__* executes the query, retrieves all matching records and tests the length of the resulting list;
- *count* executes a special query that only returns a single value: the number of matching records.

Thus, *__len__* is more suitable when you are going to iterate the records anyway (and do no extra queries), while *count* is better when you just want to check if the records exist, or to only use a part of matching records (i.e. a slice).

delete()
Deletes all records that match current query.

order_by(*names*, *reverse=False*)
Returns a query object with same conditions but with results sorted by given field. By default the direction of sorting is ascending.

Parameters

- **names** – list of strings: names of fields by which results should be sorted. Some backends may only support a single field for sorting.
- **reverse** – *bool*: if *True*, the direction of sorting is reversed and becomes descending. Default is *False*.

values(*name*)
Returns a list of unique values for given field name.

Parameters **name** – the field name.

where(***conditions*)
Returns Query instance filtered by given conditions. The conditions are specified by backend's underlying API.

where_not(***conditions*)
Returns Query instance. Inverted version of *where()*.

exception doqu.backend_base.**ProcessorDoesNotExist**
This exception is raised when given backend does not have a processor suitable for given value. Usually you will need to catch a subclass of this exception.

class doqu.backend_base.**LookupManager**
Usage:

```
lookup_manager = LookupManager()

@lookup_manager.register('equals', default=True)  # only one lookup can be default
def exact_match(name, value):
    '''
    Returns native Tokyo Cabinet lookup triplets for given
    backend-agnostic lookup triplet.
    '''
    if isinstance(value, basestring):
        return (
            (name, proto.RDBQCSTREQ, value),
        )
    if isinstance(value, (int, float)):
        return (
            (name, proto.RDBQCNUMEQ, value),
        )
    raise ValueError
```

Now if you call `lookup_manager.resolve('age', 'equals', 99)`, the returned value will be `(('age', proto.RDBCNUMEQ, 99),)`.

A single generic lookup may yield multiple native lookups because some backends do not support certain lookups directly and therefore must translate them to a combination of elementary conditions. In most cases `resolve()` will yield a single condition. Its format is determined by the query adapter.

exception_class
alias of `LookupProcessorDoesNotExist`

resolve(*name*, *operation*, *value*)
Returns a set of backend-specific conditions for given backend-agnostic triplet, e.g.:

```
('age', 'gt', 90)
```

will be translated by the Tokyo Cabinet backend to:

```
('age', 9, '90')
```

or by the MongoDB backend to:

```
{'age': {'$gt': 90}}
```

exception doqu.backend_base.**LookupProcessorDoesNotExist**
This exception is raised when given backend does not support the requested lookup.

class doqu.backend_base.**ConverterManager**

An instance of this class can manage property processors for given backend. Processor classes must be registered against Python types or classes. The processor manager allows encoding and decoding data between a document class instance and a database record. Each backend supports only a certain subset of Python datatypes and has its own rules in regard to how *None* values are interpreted, how complex data structures are serialized and so on. Moreover, there's no way to guess how a custom class should be processed. Therefore, each combination of data type + backend has to be explicitly defined as a set of processing methods (to and from).

exception_class

alias of DataProcessorDoesNotExist

from_db(*datatype*, *value*)

Converts given value to given Python datatype. The value must be correctly pre-encoded by the symmetrical PropertyManager.to_db() method before saving it to the database.

Raises DataProcessorDoesNotExist if no suitable processor is defined by the backend.

to_db(*value*, *storage*)

Prepares given value and returns it in a form ready for storing in the database.

Raises DataProcessorDoesNotExist if no suitable processor is defined by the backend.

exception doqu.backend_base.**DataProcessorDoesNotExist**

This exception is raised when given backend does not have a datatype processor suitable for given value.

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

CHAPTER

NINE

AUTHOR

Originally written by Andrey Mikhaylenko since 2009.

See the file AUTHORS for a complete authors list of this application.

Please feel free to submit patches, report bugs or request features:

http://bitbucket.org/neithere/doqu/issues/

CHAPTER

TEN

LICENSING

Doqu is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Doqu is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Doqu. If not, see <http://gnu.org/licenses/>.

PYTHON MODULE INDEX

d