norman-doc Documentation

Release 0.6.1

David Townshend

July 20, 2012

Contents

Norman provides a framework for creating complex data structures using an database-like approach. The range of potential application is wide, for example in-memory databases, multi-keyed dictionaries or node graphs. These applications are illustrated in the following examples.

CHAPTER 1

Example Applications

1.1 Database

This is a small database for a personal library:

```
db = Database()

@db.add
class Book(Table):
    name = Field(unique=True)
    author = Field(index=True)

    def validate(self):
        assert isinstance(self.name, str)
        assert isinstance(self.author, Author)

@db.add
class Author(Table):
    surname = Field(unique=True)
    initials = Field(unique=True, default='')
    nationality = Field()
    books = Join(Book.author)
```

1.2 Multi-keyed Dictionary

This table can be used as a dictionary with three keys:

```
class MultiDict(Table):
    key1 = Field(unique=True)
    key2 = Field(unique=True)
    key3 = Field(unique=True)
    value = Field()
```

Values can be added by:

```
MultiDict(key1=4, key2='abc', key3=0, value='efg')
```

And queried by:

```
for m in (MultiDict.key1 == 4 & Multidict.key2 == 'abc'):
    print(m.value)
```

1.3 Node Graph

This is a graph, where each node can have many parent nodes and many children nodes:

```
class Link(Table):
    """
    Directional connections between nodes.
    """
    parent = Field(unique=True)
    child = Field(unique=True)

    def validate(self):
        assert isinstance(self.parent, Node)
        assert isinstance(self.child, Node)


class Node(Table):
    """
    Nodes in the graph.
    """
    parents = Join(query=lambda n: (Link.child == n).field('parent'))
    children = Join(query=lambda n: (Link.parent == n).field('child'))

    def validate_delete(self):
        # Delete all connecting links if a node is deleted
        (Link.parent == self).delete()
        (Link.child == self).delete()
```

Contents

2.1 Tutorial

This tutorial shows how to create a simple library database which manages books and authors using Norman.

Contents

2.1.1 Creating Tables

The first step is to create a `Table` containing all the books in the library. New tables are created by subclassing `Table`, and defining fields as class attributes using `Field`:

```
class Book(Table):
    name = Field()
    author = Field()
```

New books can be added to this table by creating instances of it:

```
Book(name='The Hobbit' author='Tolkien')
```

However, at this stage there are no restrictions on the data that is entered, so it is possible to create something like this:

```
Book(name=42, author=['This', 'is', 'not', 'an', 'author'])
```

2.1.2 Constraints

We want to add some restrictions, such as ensuring that the name is always a unique string. The way to add these constraints is to set the `name` field as unique and to add a `validate` method to the table:

```
class Book(Table):
    name = Field(unique=True)
    author = Field()

    def validate(self):
        assert isinstance(self.refno, int)
        assert isinstance(self.name, str)
```

Now, trying to create an invalid book as in the previous example will raise a `ValueError`.

Validation can also be implemented using `Table.hooks`.

2.1.3 Joined Tables

The next exercise is to add some background information about each author. The best way to do this is to create a new table of authors which can be linked to the books:

```
class Author(Table):
    surname = Field(unique=True)
    initials = Field(unique=True, default='')
    dob = Field()
    nationality = Field()
```

Two new concepts are used here. Default values can be assigned to a `Field` as illustrated by `surname`, and more than one field can be unique. This means that authors cannot have the same surname and initials, so `'A. Adams'` and `'D. Adams'` is ok, but two `'D. Adams'` is not.

We can also add a list of books by the author, by using a `Join`. This is similar to a `Field`, but is created with a reference to foreign field containing the link, and contains an iterable rather than a single value:

```
class Author(Table):
    surname = Field(unique=True)
    initials = Field(unique=True, default='')
    nationality = Field()
    books = Join(Book.author)
```

This tells the `Author` table that its `books` attribute should contain all `Book` instances with a matching `author` field:

```
class Book(Table):
    refno = Field(unique=True)
    name = Field()
    author = Field()

    def validate(self):
        assert isinstance(self.refno, int)
        assert isinstance(self.name, str)
        assert isinstance(self.author, str)
```

This is dynamic link, so every time the `books` attribute is queried, the `Book` table is scanned for matching values. Since each record has to be checked individually this can become very slow, so the `author` field can be indexed to improve performance by adding an *index* argument to its definition. It is worth noting that unique fields are automatically indexed, so `Book.name` already supports fast lookups:

```
class Book(Table):
    ...
    author = Field(index=True)
    ...
```

A `Join` can also point to another `Join`, creating what is termed a many-to-many relationship. These are discussed later, since they rely on a `Database` being used.

2.1.4 Databases

These tables are perfectly usable as they are, but for convenience they can be grouped into a `Database`. This becomes more important when serialising them:

```
db = Database()
db.add(Book)
db.add(Author)
```

`Database.add` can also be used as a class decorator, so the complete code becomes:

```
db = Database()

@db.add
class Book(Table):
    refno = Field(unique=True)
    name = Field()
    author = Field(index=True)

    def validate(self):
        assert isinstance(self.refno, int)
        assert isinstance(self.name, str)
        assert isinstance(self.author, str)

@db.add
class Author(Table):
    surname = Field(unique=True)
    initials = Field(unique=True, default='')
    nationality = Field()
    books = Join(Book.author)
```

2.1.5 Many-to-many Joins

The next step in the library is to allow people to withdraw books from it, tracking both the books a person has, and who has copies of a specific book. This is known as a many-to-many relationship, as `Book.people` contains many people and `Person.books` contains many books, and is implemented in Norman by creating a pair of joins which target each other.

First we need to create another table for people, adding a join to a new field, which we will add to `Book`. However, this causes a slight problem, since we need to reference `Book.people` in order to create `Person.books`, and we need to reference `Person.books` in order to create `Book.people`. Fortunately, Norman allows an alternative method of defining joins when the target `Table` belongs to a database:

```
@db.add
class Person(Table):
    name = Field(unique=True)
    books = Join(db, 'Book.people')

@db.add
class Book(Table):
    ...
    people = Join(db, 'Person.books')
    ...
```

In the background, a new table called `'_BookPerson'` is created and added to the database. This is just a sorted concatenation of the names of the two participating tables, prefixed with an underscore. It is possible to manually set the name used by using the *jointable* keyword argument on one of the joins:

```
@db.add
class Person(Table):
    name = Field(unique=True)
    books = Join(db, 'Book.people', jointable='JoinTable')
```

The newly created join table has two unique fields, *Book* and *Person*, i.e. the participating table names. While records can be added to it directly, it is advisable to add them to the join instead, so for example:

```
mybook.people.add(a_person)
```

2.1.6 Adding records

Now that the database is set up, we can add some records to it:

```
dickens = Author(surname='Dickens', initials='C', nationality='British')
tolkien = Author(surname='Tolkien', initials='JRR', nationality='South African')
pratchett = Author(surname='Pratchett', initials='T', nationality='British')
Book(name='Wyrd Sisters', author=pratchett)
Book(name='The Hobbit', author=tolkien)
Book(name='Lord of the Rings', author=tolkien)
Book(name='Great Expectations', author=dickens)
Book(name='David Copperfield', author=dickens)
Book(name='Guards, guards', author=pratchett)
```

2.1.7 Queries

Queries are constructed by comparing and combining fields. The following examples show how to extract various bit of information from the database.

See Also:

Queries

1. Listing all records in a table is as simple as iterating over it, so generator expressions can be used to extract a list of fields. For example, to get a sorted list of author's surnames:

   ```
   >>> sorted(a.surname for a in Author)
   ['Dickens', 'Pratchett', 'Tolkien']
   ```

2. Records can be queried based on their field values. For example, to list all South African authors:

   ```
   >>> for a in (Author.nationality == 'South African'):
   ...     print(a.surname)
   Tolkien
   ```

3. Queries can be combined and nested, so to get all books by authors who's initials are in the first half of the alphabet:

   ```
   books = Books.authors & (Author.initials <= 'L')
   ```

4. A single result can be obntained using `Query.one`:

   ```
   mybook = (Book.name == 'Wyrd Sisters').one()
   ```

4. Records can be added based on certain queries:

   ```
   (Author.nationality == 'British').add(surname='Adams', intials='D')
   ```

2.1.8 Serialisation

`serialise` provides an extensible framework for serialising databases and a sample implementation for serialising to sqlite. Serialising and de-serialising is as simple as:

```
MySerialiser.dump(mydb, filename)
```

and:

```
MySerialiser.load(mydb, filename)
```

For more detail, see the `serialise` module.

2.2 What's New

This file lists new features and major changes to **Norman**. For a detailed changelog, see the mercurial log.

2.2.1 Norman-0.6.2

Release Date: Not Released

- Add built-in support for many-to-many joins.
- Hooks added to `Table` to allow more control over validation.
- Add `Query.field`, allowing queries to traverse tables.
- Add `Query.add`, allowing records to be created based on query criteria.
- `Field` level validation added, including some validator factories.
- Add `validate.todatetime`, `validate.todate` and `validate.totime`.
- Deprecated the `tools` module.

2.2.2 Norman-0.6.1

Release Date: 2012-07-12

- New serialiser framework added, based on `serialise.Serialiser`. A sample serialiser, `serialise.Sqlite` is included.
- `serialise.Sqlite3` has been deprecated.
- Documentation overhauled introducing major changes to the documentation layout.
- Add boolean comparisons, `Query.delete` and `Query.one` methods to `Query`.
- `Table` now supports inheritance by copying its fields.
- Several changes to implementations, generally to improve performance and consistency.

2.2.3 Norman-0.6.0

Release Date: 2012-06-12

- Python 2.6 support by Ilya Kutukov
- Move serialisation functions to a new serialise module. This module will be expanded and updated in the near future.
- Add sensible `repr` to `Table` and `NotSet` objects
- `Query` object added, introducing a new method of querying tables, involving `Field` and `Query` comparison operators.
- `Join` class created, which will replace `Group` in 0.7.0.
- `Field.name` and `Field.owner`, which previously existed, have now been formalised and documented.
- `Field.default` is respected when initialising tables
- `Table._uid` property added for Table objects.

- Allow `Table.validate_delete` to make changes.
- Two new `tools` functions added: `tools.dtfromiso` and `tools.reduce2`.
- `Database.add` method added.
- Documentation updated to align with docstrings.
- Fix a bunch of style and PEP8 related issues
- Minor bugfixes

2.2.4 Norman-0.5.2

Release Date: 2012-04-20

- Fixed failing tests
- `Group.add` implemented and documented
- Missing documentation fixed

2.2.5 Norman-0.5.1

Release Date: 2012-04-20

- Exceptions raised by validation errors are now all ValueError
- Group object added to represent sub-collections
- Deletion validation added to tables through `Table.validate_delete`
- Minor documentation updates
- Minor bugfixes

2.2.6 Norman-0.5.0

Release Date: 2012-04-13

- First public release, repository imported from private project.

2.3 Data Structures

Contents

2.3.1 Database

`Database` instances act as containers of `Table` objects, and support `__getitem__`, `__contains__` and `__iter__`. `__getitem__` returns a table given its name (i.e. its class name), `__contains__` returns whether a `Table` object is managed by the database and `__iter__` returns a iterator over the tables.

Tables may be added to the database when they are created by using `Database.add` as a class decorator. For example:

```
>>> db = Database()
>>> @db.add
... class MyTable(Table):
...     name = Field()
>>> MyTable in db
True
```

The database can be written to a file through the `serialise` module. Currently only sqlite3 is supported. If a `Database` instance represents a document state, it can be saved using the following code:

```
>>> serialise.Sqlite.dump(db, 'file.sqlite')
```

And reloaded:

```
>>> serialise.Sqlite.load(db, 'file.sqlite')
```

class `norman.`**Database**
The main database class containing a list of tables.

add(*table*)
Add a `Table` class to the database.

This is the same as including the *database* argument in the class definition. The table is returned so this can be used as a class decorator.

```
>>> db = Database()
>>> @db.add
... class MyTable(Table):
...     name = Field()
```

tablenames:
Return an list of the names of all tables managed by the database.

reset()
Delete all records from all tables.

2.3.2 Tables

Tables are implemented as a class, with records as instances of the class. Accordingly, there are many class-level operations which are only applicable to a `Table`, and others which only apply to records. `Table` operations are defined in `TableMeta`, the metaclass used to create `Table`.

class `norman.`**TableMeta**
Base metaclass for all tables.

Tables support a limited sequence-like interface, with rapid lookup through indexed fields. The sequence operations supported are `__len__`, `__contains__` and `__iter__`, and all act on instances of the table, i.e. records.

hooks
A `dict` containing lists of callables to be run when an event occurs.

Two events are supported: validation on setting a field value and deletion, identified by keys `'validate'` and `'delete'` respectively. When a triggering event occurs, each hook in the list is called in order with the affected table instance as a single argument until an exception occurs. If the exception is an `AssertionError` it is converted to a `ValueError`. If no exception occurs, the event is considered to have passed, otherwise it fails and the table record rolls back to its previous state.

These hooks are called before `Table.validate` and `Table.validate_delete`, and behave in the same way.

contains(***kwargs*)
Return `True` if the table contains any records with field values matching *kwargs*.

delete(*[records=None]*, ***keywords*)
Delete delete all instances in *records* which match *keywords*.

If *records* is omitted then the entire table is searched. For example:

```
>>> class T(Table):
...     id = Field()
...     value = Field()
>>> records = [T(id=1, value='a'),
...            T(id=2, value='b'),
...            T(id=3, value='c'),
...            T(id=4, value='b'),
...            T(id=5, value='b'),
...            T(id=6, value='c'),
...            T(id=7, value='c'),
...            T(id=8, value='b'),
...            T(id=9, value='a')]
>>> sorted(t.id for t in T.get())
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> T.delete(records[:4], value='b')
>>> sorted(t.id for t in T.get())
[1, 3, 5, 6, 7, 8, 9]
```

If no records are specified, then all are used.

```
>>> T.delete(value='a')
>>> sorted(t.id for t in T.get())
[3, 5, 6, 7, 8]
```

If no keywords are given, then all records in in *records* are deleted.

```
>>> T.delete(records[2:5])
>>> sorted(t.id for t in T.get())
[6, 7, 8]
```

If neither records nor keywords are deleted, then the entire table is cleared.

fields()
Return an iterator over field names in the table.

get(***kwargs*)
Return a `set` of for all records with field values matching *kwargs*.

iter(***kwargs*)
Iterate over records with field values matching *kwargs*.

class `norman.`**Table**(***kwargs*)
Each instance of a Table subclass represents a record in that Table.

This class should be subclassed to define the fields in the table. It may also optionally provide `validate` and `validate_delete` methods.

`Field` names should not start with _, as these names are reserved for internal use. Fields may be added to a `Table` after the `Table` is created, provided they do not already belong to another `Table`, and the `Field` name is not already used in the `Table`.

_uid
This contains an id which is unique in the session.

It's primary use is as an identity key during serialisation. Valid values are any integer except 0, or a UUID. The default value is calculated using `uuid.uuid4` upon its first call. It is not necessarily required that it be universally unique.

validate()
Raise an exception if the record contains invalid data.

This is usually re-implemented in subclasses, and checks that all data in the record is valid. If not, and exception should be raised. Internal validate (e.g. uniqueness checks) occurs before this method is called, and a failure will result in a `ValueError` being raised. For convenience, any `AssertionError` which is raised here is considered to indicate invalid data, and is re-raised as a `ValueError`. This allows all validation errors (both from this function and from internal checks) to be captured in a single *except* statement.

Values may also be changed in the method. The default implementation does nothing.

validate_delete()
Raise an exception if the record cannot be deleted.

This is called just before a record is deleted and is usually re-implemented to check for other referring instances. For example, the following structure only allows deletions of *Name* instances not in a *Grouper*.

```
>>> class Name(Table):
...     name = Field()
...     group = Field(default=None)
...
...     def validate_delete(self):
...         assert self.group is None, "Can't delete '{}'".format(self.group)
...
>>> class Grouper(Table):
...     id = Field()
...     names = Group(Name, lambda s: {'group': s})
...
>>> group = Grouper(id=1)
>>> n1 = Name(name='grouped', group=group)
>>> n2 = Name(name='not grouped', group=None)
>>> Name.delete(name='not grouped')
>>> Name.delete(name='grouped')
Traceback (most recent call last):
    ...
ValueError: Can't delete 'grouped'
>>> {name.name for name in Name.get()}
{'grouped'}
```

Exceptions are handled in the same was as for `validate`.

This method can also be used to propogate deletions and can safely modify this or other tables.

2.3.3 Fields

norman.**NotSet**
A sentinel object indicating that the field value has not yet been set.

This evaluates to `False` in conditional statements.

class norman.**Field**
A `Field` is used in tables to define attributes of data.

When a table is created, fields can be identified by using a `Field` object:

```
>>> class MyTable(Table):
...     name = Field()
```

`Field` objects support *get* and *set* operations, similar to *properties*, but also provide additional options. They are intended for use with `Table` subclasses.

Field options are set as keyword arguments when it is initialised

Keyword	Default	Description
unique	False	True if records should be unique on this field. In database terms, this is the same as setting a primary key. If more than one field have this set then records are expected to be unique on all of them. Unique fields are always indexed.
index	False	True if the field should be indexed. Indexed fields are much faster to look up. Setting `unique = True` implies `index = True`
default	None	If missing, `NotSet` is used.
readonly	False	Prohibits setting the variable, unless its value is `NotSet`. This can be used with *default* to simulate a constant.
validate	None	If set, should be a list of functions which are to be used as validators for the field. Each function should accept a and return a single value, and should raise an exception if the value is invalid. The return value is the value passed to the next validator.

Note that *unique* and *index* are table-level controls, and are not used by `Field` directly. It is the responsibility of the table to implement the necessary constraints and indexes.

Fields have read-only properties, *name* and *owner* which are set to the assigned name and the owning table respectively when the table class is created.

Fields can be used with comparison operators to return a `Query` object containing matching records. For example:

```
>>> class MyTable(Table):
...     oid = Field(unique=True)
...     value = Field()
>>> t0 = MyTable(oid=0, value=1)
>>> t1 = MyTable(oid=1, value=2)
>>> t2 = MyTable(oid=2, value=1)
>>> Table.value == 1
Query(MyTable(oid=0, value=1), MyTable(oid=2, value=1))
```

The following comparisons are supported for a `Field` object: ==, <, >, <=, >==, !=. The `&` operator is used to test for containment, e.g. ‘‘ Table.field & mylist‘‘ returns all records where the value of `field` is in `mylist`.

See Also:

`validate` for some pre-build validators.

2.3.4 Joins

A join is basically an object which dynamically creates queries for a specific record. This is best explained through an example:

```
>>> class Child(Table):
...     parent = Field()
...
>>> class Parent(Table):
...     children = Join(Child.parent)
...
>>> p = Parent()
>>> c1 = Child(parent=p)
>>> c2 = Child(parent=p)
>>> p.children
{c1, c2}
```

Here, `Parent.children` is a factory which returns a `Query` for all `Child` records where `child.parent == parent_instance` for a specific `parent_instance`. Joins have a `query` attribute which is a `Query` factory, returning a `Query` for a given instance of the owning table.

class norman.**Join**(**args*, ***kwargs*)
A join, returning a Query.

Joins can be created with the following arguments:

Join(query=queryfactory) Explicitly set the query factory. queryfactory is a callable which accepts a single argument and returns a Query.

Join(table.field) This is the most common format, since most joins simply involve looking up a field value in another table. This is equivalent to specifying the following query factory:

```
def queryfactory(value):
    return table.field == value
```

Join(db, 'table.field') This has the same affect as the previous example, but is used when the foreign field has not yet been created. In this case, the query factory first locates 'table.field' in the Database db.

Join(other.join) It is possible set the target of a join to another join, creating a *many-to-many* relationship. When used in this way, a join table is automatically created, and can be accessed from Join.jointable. If the optional keyword parameter *jointable* is used, the join table name is set to it.

See Also:

http://en.wikipedia.org/wiki/Many-to-many_(data_model) For more information on *many-to-many* joins.

jointable
The join table in a *many-to-many* join.

This is None if the join is not a *many-to-many* join, and is read only.

name
The name of the Join. This is read only.

owner
The Table containing the Join. This is read only.

query
A function which accepts an instance of owner and returns a Query.

2.4 Queries

Norman features a flexible and extensible query API, the basis of which is the Query class. Queries are constructed by manipulating Field and other Query objects; the result of each operation is another Query.

Contents

2.4.1 Tutorial

The purpose of this short tutorial is to explain the basic concepts behind Norman queries.

Queries are constructed as a series of field comparisons, for example:

```
q1 = MyTable.age > 4
q2 = MyTable.parent.name == 'Bill'
```

These can be joined together with set combination operators:

```
q3 = MyTable.age > 4 | MyTable.parent.name == 'Bill'
```

Containment in an iterable can be checked using the `&` operator. This is the same usage as in `set`:

```
q4 = MyTable.parent.name & ['Bill', 'Bob', 'Bruce']
```

Since queries are themselves iterable, another query can be used as the container:

```
q5 = MyTable.age & OtherTable.age
```

A custom function can be used for filtering records from a `Table` or another `Query`:

```
isvalid = lambda record: record.parrot.endswith('notlob')
q6 = query(isvalid, q5)
```

If the filter function is omitted, then all records are assumed to pass. This is useful for creating a query of a whole table:

```
q7 = query(MyTable)
```

The result of each of these is a `Query` object, which is a set-like iterable of records.

An existing query can be refreshed after the base data has changed by calling it as a function:

```
q7()
```

2.4.2 API

norman.**query**([*func*], *table*)

Return a new `Query` for records in *table* for which *func* is `True`.

table is a `Table` or `Query` object. If *func* is missing, all records are assumed to pass. If it is specified, is should accept a record as its argument and return `True` for passing records.

class norman.**Query**

A set-like object which represents the results of a query.

This object should never be instantiated directly, instead it should be created as the result of a query on a `Table` or `Field`.

This object allows most operations permitted on sets, such as unions and intersections. Comparison operators (such as <) are not supported, except for equality tests.

The following operations are supported:

Operation	Description
`r in q`	Return `True` if record `r` is in the results of query `q`.
`len(q)`	Return the number of results in `q`.
`iter(q)`	Return an iterator over records in `q`.
`q1 == q2`	Return `True` if `q1` and `q2` contain the same records.
`q1 != q2`	Return `True` if `not a == b`
`q1 & q2`	Return a new `Query` object containing records in both `q1` and `q2`.
`q1 | q2`	Return a new `Query` object containing records in either `q1` or `q2`.
`q1 ^ q2`	Return a new `Query` object containing records in either `q1` or `q2`, but not both.
`q1 - q2`	Return a new `Query` object containing records in `q1` which are not in `q2`.

add([***kwargs*])

Add a record based on the query criteria.

This method is only available for queries of the form `field == value`, a `&` combination of them, or a `field` query created from a query of this form. *kwargs* is the same as used for creating a `Table` instance, but is updated to include the query criteria. *arg* is only used for queries created by `field`, and is a record to add to the field. See `field` for more information.

delete()
Delete all records matching the query.

Records are deleted from the table. If no records match, nothing is deleted.

field(*fieldname*)
Return a new `Query` containing records in a single field.

The set of records returned by this is similar to:

```
set(getattr(r, fieldname) for r in query)
```

However, the returned object is another `Query` instead of a set. Only instances of a `Table` subclass are contained in the results, other values are dropped. This is functionally similar to a SQL query on a foreign key. If the target field is a `Join`, then all the results of each join are concatenated.

If this query supports addition, then the resultant query will too, but with slightly different parameters. For example:

```
(Table1.id == 4).field('tble2').add(table2_instance)
```

is the same as:

```
(Table1.id == 4).add(table2=table2_instance)
```

one([*default*])
Return a single value from the query results.

If the query is empty and *default* is specified, then it is returned instead. Otherwise an exception is raised.

2.4.3 Groups

Deprecated since version 0.6.

class norman.**Group**(*table*[, *matcher=None*], ***kwargs*)
This is a collection class which represents a collection of records.

Parameters

- **table** – The table which contains records returned by this `Group`.
- **matcher** – A callable which returns a dict. This can be used instead of *kwargs* if it needs to be created dynamically.
- **kwargs** – Keyword arguments used to filter records.

If *matcher* is specified, it is called with a single argument to update *kwargs*. The argument passed to it is the instance of the owning table, so this can only be used where `Group` is in a class.

`Group` is a set-like container, closely resembling a `Table` and supports `__len__`, `__contains__` and `__iter__`.

This is typically used as a field type in a `Table`, but may be used anywhere where a dynamic subset of a `Table` is needed.

The easiest way to demonstrating usage is through an example. This represents a collection of *Child* objects contained in a *Parent*.

```
>>> class Child(Table):
...     name = Field()
...     parent = Field()
...
...     def __repr__(self):
...         return "Child('{}')".format(self.name)
...
>>> class Parent(Table):
...     children = Group(Child, lambda self: {'parent': self})
...
>>> parent = Parent()
>>> a = Child(name='a', parent=parent)
>>> b = Child(name='b', parent=parent)
>>> len(parent.children)
2
>>> parent.children.get(name='a')
{Child('a')}
>>> parent.children.iter(name='b')
<set_iterator object at ...>
>>> parent.children.add(name='c')
Child('c')
```

table
Read-only property containing the `Table` object referred to.

add(***kwargs*)
Create a new record of the reference `table`.

kwargs is updated with the keyword arguments defining this `Group` and the resulting dict used as the initialisation parameters of `table`.

contains(***kwargs*)
Return `True` if the `Group` contains records matching *kwargs*.

delete([*records=None*], ***keywords*)
Delete delete all instances in *records* which match *keywords*.

This only deletes instances in the `Group`, but it completely deletes them. If *records* is omitted then the entire `Group` is searched.

See Also:

Table.delete

get(***kwargs*)
Return a set of all records in the `Group` matching *kwargs*.

iter(***kwargs*)
Iterate over records in the `Group` matching *kwargs*.

2.5 Serialisation

In addition to supporting the `pickle` protocol, Norman provides a framework for serialising and deserializing databases to other formats through the `norman.serialise` module. Serialisation classes inherit `Serialiser`, and should reimplement at least `iterfile` and `write_record`. `Serialiser` has the following methods, grouped by functionality:

- General
  - `open`
  - `close`
- Loading (Reading)

  - load (Class method)
  - create_records
  - finalise_read
  - initialise_read
  - isuid
  - iterfile
  - read
  - run_read
- Dumping (Writing)
  - dump (Class method)
  - finalise_write
  - initialise_write
  - iterdb
  - run_write
  - simplify
  - write
  - write_record

Contents

2.5.1 Serialiser framework

class norman.serialise.**Serialiser**(*db*)

An abstract base class providing a framework for serialisers.

Subclasses are instantiated with a Database object, and serialisation and de-serialisation is done through the write and read methods. Class methods dump and load may also be used.

Subclasses are required to implement iterfile and its counterpart, write_record, but may re-implement any other methods to customise behaviour.

db

The database handled by the serialiser.

fh

An open file (or database connection), or None.

This is set to the result of open. If a file is not currently open, then this is None.

mode

Indicates the current operation.

This is set to 'w' during *dump* operations and 'r' during *load*. At other times it is None.

classmethod dump(*db*, *filename*)

This is a convenience method for calling write.

This is equivalent to Serialise(db).write(filename) and is provided for compatibility with the pickle API.

classmethod load(*db*, *filename*)
This is a convenience method for calling `read`.

This is equivalent to `Serialise(db).read(filename)` and is provided for compatibility with the `pickle` API.

close()
Close the currently opened file.

The default behaviour is to call the file object's `close` method. This method is always called once a file has been opened, even if an exception occurs during writing.

create_records(*records*)
Create one or more new records.

This is called for every group of cyclic records. For example, if records *a* references record *b*, which references record *c*, and record *c* references record *a*, then records *a*, *b*, and *c* form a cycle. If record *d* references record *e* but record *e* doesn't reference any other record, each of them are considered to be isolated.

records is an iterator yielding tuples of `(table, uid, data, cycles)` for each record in the cycle, or only one record if there is no cycle. The first three values are the same as those returned by `iterfile`, except that foreign uids in data have been dereferenced. *cycles* is a set of field names which contain the cyclic data.

The default behaviour is to remove the cyclic fields from *data* for each record, create the records using `table(**data)` and assign the created records to the cyclic fields. The *uid* of each record is also assigned to its *_uid* attribute.

The return value is an iterator over `(uid, record)` pairs.

finalise_read()
Finalise the file after reading data.

This is called after `run_read` but before `close`, and can be re-implemented to for implementation-specific finalisation.

The default implementation does nothing.

finalise_write()
Finalise the file after writing data.

This is called after `run_write` but before `close`, and can be re-implemented to for implementation-specific finalisation.

The default implementation does nothing.

initialise_read()
Prepare the file for reading data.

This is called before `run_read` but after `open`, and can be re-implemented to for implementation-specific setup.

The default implementation does nothing.

initialise_write()
Prepare the file for writing data.

This is called before `run_write` but after `open`, and can be re-implemented to for implementation-specific setup.

The default implementation does nothing.

isuid(*field*, *value*)
Return `True` if *value*, for the specified *field*, could be a *uid*.

field is a `Field` object.

This only needs to check whether the value could possibly represent another field. It is only actually considered a *uid* if there is another record which matches it.

By default, this returns `True` for all strings which match a UUID regular expression, e.g. `'a8098c1a-f86e-11da-bd1a-00112444be1e'`.

iterdb()
Return an iterator over records in the database.

Records should be returned in the order they are to be written. The default implementation is a generator which iterates over records in each table.

iterfile()
Return an iterator over records read from the file.

Each item returned by the iterator should be a tuple of `(table, uid, data)` where *table* is the `Table` containing the record, *uid* is a globally unique value identifying the record and *data* is a dict of field values for the record, possibly containing other uids.

This is commonly implemented as a generator.

read(*filename*)
Load data into `db` from *filename*.

fieldname is used only to open the file using `open`, so, depending on the implementation could be anything (e.g. a URL) which `open` recognises. It could even be omitted entirely if, for example, the serialiser reads from stdin.

open(*filename*)
Open *filename* for the current `mode`.

The return value should be a handle to the open file. The default behaviour is to open the file as binary using the builtin *open* function.

run_read()
Read data from the currently opened file.

This is called between `initialise_read` and `finalise_read`, and converts each value returned by `iterfile` into a record using `create_records`. It also attempts to re-map nested records by searching for matching uids.

Cycles in the data are detected, and all records involved in in a cycle are created in `create_records`.

run_write()
Called by `dump` to write data.

This is called after `initialise_write` and before `finalise_write`, and simply calls `write_record` for each value yielded by `iterdb`.

simplify(*record*)
Convert a record to a simple python structure.

The default implementation converts *record* to a `dict` of field values, omitting `NotSet` values and replacing other records with their *_uid* properties. The return value of this implementation is a tuple of `(tablename, record._uid, record_dict)`.

write(*filename*)
Write the database to *filename*.

fieldname is used only to open the file using `open`, so, depending on the implementation could be anything (e.g. a URL) which `open` recognises. It could even be omitted entirely if, for example, the serialiser dumps the database as formatted text to stdout.

write_record(*record*)
Write *record* to the current file.

This is called by `run_write` for every record yielded by `iterdb`. *record* is the values returned by `simplify`.

2.5.2 Sqlite

class norman.serialise.**Sqlite**
This is a Serialiser which reads and writes to a sqlite database.

Each table in db is dumped to a sqlite table with the same field names. An additional field, *_uid_* is included which contains the record's *_uid*. The sqlite database does not have any constraints, not even primary key constraints, as it is intended to be used purely for storage.

The following methods are re-implemented from Serialiser:

- finalise_write commits changes to the database.
- initialise_write starts a database transaction and create tables.
- initialise_read sets the sqlite3 row factory.
- iterfile yield records from each valid table in the file which matches a table in db.
- open returns an open database connection to *filename*.
- write_record adds a record to the sqlite database.

class norman.serialise.**Sqlite3**
Deprecated since version 0.6.1.

dump(*db*, *filename*)
Dump the database to a sqlite database.

Each table is dumped to a sqlite table, without any constraints. All values in the table are converted to strings and foreign objects are stored as an integer id (referring to another record). Each record has an additional field, '_oid_', which contains a unique integer.

load(*db*, *filename*)
The database supplied is read as follows:

1. Tables are searched for by name, if they are missing then they are ignored.
2. If a table is found, but does not have an "oid" field, it is ignored
3. Values in "oid" should be unique within the database, e.g. a record in "units" cannot have the same "oid" as a record in "cycles".
4. Records which cannot be added, for any reason, are ignored and a message logged.

2.6 Tools

Deprecated since version 0.6.2. Some useful tools for use with Norman are provided in norman.tools.

norman.tools.**dtfromiso**(*iso*)
Return a datetime object from a string representation in ISO format.

The database serialisation procedures store datetime objects as strings, in ISO format. This provides an easy way to reverse this. datetime, date and time objects are all supported.

Note that this assumes naive datetimes.

```
>>> import datetime
>>> dt = datetime.date(2001, 12, 23)
>>> isodt = str(dt)
>>> dtfromiso(isodt)
datetime.date(2001, 12, 23)
```

norman.tools.**float2**(*s*[, *default=0.0*])
Convert *s* to a float, returning *default* if it cannot be converted.

```
>>> float2('33.4', 42.5)
33.4
>>> float2('cannot convert this', 42.5)
42.5
>>> float2(None, 0)
0
>>> print(float2('default does not have to be a float', None))
None
```

norman.tools.**int2**(*s*[, *default=0*])

Convert *s* to an int, returning *default* if it cannot be converted.

```
>>> int2('33', 42)
33
>>> int2('cannot convert this', 42)
42
>>> print(int2('default does not have to be an int', None))
None
```

norman.tools.**reduce2**(*func*, *seq*, *default*)

Similar to `functools.reduce`, but return *default* if *seq* is empty.

The third argument to `functools.reduce` is an *initializer*, which essentially acts as the first item in *seq*. In this function, *default* is returned if *seq* is empty, otherwise it is ignored.

```
>>> reduce2(lambda a, b: a + b, [1, 2, 3], 4)
6
>>> reduce2(lambda a, b: a + b, [], 'default')
'default'
```

2.7 Validators

norman.validate.**ifset**(*func*)

Return `func(value)` if *value* is not `NotSet, otherwise return 'NotSet`.

This is normally used as a wrapper around another validator to permit `NotSet` values to pass. For example:

```
>>> validator = ifset(istype(float))
>>> validator(4.3)
4.3
>>> validator(NotSet)
NotSet
>>> validator(None)
Traceback (most recent call last):
    ...
TypeError: None
```

norman.validate.**isfalse**(*func*[, *default*])

Return a `Field` validator which passes if *func* returns `False`.

Parameters

- **func** – A callable which returns `False` if the value passes.
- **default** – The value to return if *func* returns `True`. If this is omitted, an exception is raised.

norman.validate.**istrue**(*func*[, *default*])

Return a `Field` validator which passes if *func* returns `True`.

Parameters

- **func** – A callable which returns `True` if the value passes.

- **default** – The value to return if *func* returns `False`. If this is omitted, an exception is raised.

`norman.validate.`**`istype`**(*t*[, *t2*[, *t3*[, ...]]])
Return a `Field` validator which raises an exception on an invalid type.

Parameters **t** – The expected type, or types.

`norman.validate.`**`settype`**(*t*, *default*)
Return a `Field` validator which converts the value to a type

Parameters

- **t** – The required type.
- **default** – If the value cannot be converted, then use this value instead.

`norman.validate.`**`todate`**([*fmt*])
Return a validator which converts a string to a `datetime.date`.

If *fmt* is omitted, the ISO representation used by `datetime.date.__str__` is used, otherwise it should be a format string for `datetime.strptime`.

If the value passed to the validator is a `datetime.datetime`, the *date* component is returned. If it is a `datetime.date` it is returned unchanged.

The return value is always a `datetime.date` object. If the value cannot be converted an exception is raised.

`norman.validate.`**`todatetime`**([*fmt*])
Return a validator which converts a string to a `datetime.datetime`.

If *fmt* is omitted, the ISO representation used by `datetime.datetime.__str__` is used, otherwise it should be a format string for `datetime.strptime`.

If the value passed to the validator is a `datetime.datetime` it is returned unchanged. If it is a `datetime.date` or `datetime.time`, it is converted to a `datetime.datetime`, replacing missing the missing information with `1900-1-1` or `00:00:00`.

The return value is always a `datetime.datetime` object. If the value cannot be converted an exception is raised.

`norman.validate.`**`totime`**([*fmt*])
Return a validator which converts a string to a `datetime.time`.

If *fmt* is omitted, the ISO representation used by `datetime.time.__str__` is used, otherwise it should be a format string for `datetime.strptime`.

If the value passed to the validator is a `datetime.datetime`, the *time* component is returned. If it is a `datetime.time` it is returned unchanged.

The return value is always a `datetime.time` object. If the value cannot be converted an exception is raised.

Python Module Index

n