norman-doc Documentation

Release 0.7.2

David Townshend

June 04, 2015

Contents

Norman is a framework for advanced data structures in python using an database-like approach. The range of potential applications is wide, for example in-memory databases, multi-keyed dictionaries or node graphs.

CHAPTER 1

Contents

1.1 Introduction

Norman is designed to make it easy and efficient to implement any data structure more complex than a `dict`. The structures are stored entirely in memory, and most operations on them are significantly faster than *O(n)*, often *O(log n)*.

1.1.1 Features

Database-like API

Norman uses a database approach and terminology, allowing it to be used to prototype a formal database. The basic data object is a `Table` which can be instantiated to create records. This is the same approach used by sqlalchemy.

Validation

Data validation is easy to apply on either `fields` or `tables`, and can be implemented using the full power of Python.

Complex structures

Tables can be linked together using a `Join`. This is similar in concept to a typical database join, but is far more flexible as it allows any `Query` to be used as the join definition.

Mutable structure definitions

Data structures are completely mutable, and every aspect of them can be changed at any time. This feature is especially useful for `AutoTable`, which dynamically creates fields as data is added.

Powerful queries

Norman provides a powerful and efficient query mechanism which can be customised to allow rapid, indexed lookups on arbitrary queries (e.g. records where `record.text.endswith('z')`).

Serialisation framework

The `serialise` module provides a framework for easily developing readers and writers for any file format. This allows norman to be used as a file type converter.

1.1.2 Installation

Norman supports Python 2.6 or higher (up to 3.3). The test suite requires nose and mock to run.

Norman in on pypi, so it can be installed using `pip install norman` or can be installed from source. Please user the issue tracker to report bugs and feature requests.

1.1.3 Examples

Norman is designed for working with relatively small amounts of data (i.e. which can fit into memory), but which have complex structures and relationships. A few examples of how Norman can be used are:

1. Extending python data structures, e.g. a multi-keyed dictionary:

```
>>> class MultiDict(Table):
...     key1 = Field(unique=True)
...     key2 = Field(unique=True)
...     key3 = Field(unique=True)
...     value = Field()
...
>>> MultiDict(key1=4, key2='abc', key3=0, value='a')
MultiDict(key1=4, key2='abc', key3=0, value='a')
>>> MultiDict(key1=5, key2='abc', key3=5, value='b')
MultiDict(key1=5, key2='abc', key3=5, value='b')
>>> MultiDict(key1=6, key2='def', key3=0, value='c')
MultiDict(key1=6, key2='def', key3=0, value='c')
>>> MultiDict(key1=4, key2='abc', key3=5, value='d')
MultiDict(key1=4, key2='abc', key3=5, value='d')
>>> query = (MultiDict.key1 == 4) & (MultiDict.key2 == 'abc')
>>> for item in sorted(query, key=lambda r: r.value):
...     print(item)
MultiDict(key1=4, key2='abc', key3=0, value='a')
MultiDict(key1=4, key2='abc', key3=5, value='d')
```

2. A tree, where each node has a parent:

```
>>> class Node(Table):
...     parent = Field()
...     children = Join(parent)
...     node_data = Field()
...
>>> root = Node(node_data='root node')
>>> child1 = Node(node_data='child1', parent=root)
>>> child2 = Node(node_data='child2', parent=root)
>>> subchild1 = Node(node_data='2nd level child', parent=child1)
>>> sorted(n.node_data for n in root.children())
['child1', 'child2']
```

3. A node graph, where nodes are directionally connected by edges:

```
>>> class Edge(Table):
...     from_node = Field(unique=True)
...     to_node = Field(unique=True)
...
>>> class Node(Table):
...     edges_out = Join(Edge.from_node)
...     edges_in = Join(Edge.to_node)
...     all_edges = Join(query=lambda me: \
...                       (Edge.from_node == me) | (Edge.to_node == me))
...
...     def validate_delete(self):
...         # Delete all connecting links if a node is deleted
...         self.edges.delete()
```

3. Even a lightweight database for a personal library:

```
>>> db = Database()
>>>
>>> @db.add
... class Book(Table):
...     name = Field(unique=True, validators=[validate.istype(str)])
...     author = Field()
...
...     def validate(self):
...         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)
```

4. Norman provides a sophisticated serialisation system for writing data to and loading it from virtually any source. This example shows how it can be used as a converter data from CSV files to a sqlite database:

```
>>> db = AutoDatabase()
>>> serialise.CSV().read('source files', db)
>>> serialise.Sqlite().write('output.sqlite', db)
```

1.2 Tutorial

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

Contents

1.2.1 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'])
```

1.2.2 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`*.

1.2.3 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)
```

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.

1.2.4 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()

    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)
```

1.2.5 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)
```

1.2.6 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)
```

1.2.7 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')
```

Since 0.7, all fields are automatically indexed, so queries are fast. Depending on the application, it is possible to change how the data is indexed, allowing for more control over how data can be queried. For example, if we were more concerned about querying books by title length, we could use `len` as the index key function:

```
class Book(Table):
    ...
    name = Field(key=len)
    ...
```

Then we could query all books with a title longer than 10 characters:

```
Book.name > ' '*10
```

Note that the target of the query is also affected by the key, so we need to give it a value such that `len(value)` returns 10.

1.2.8 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.

1.3 What's New

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

1.3.1 Norman-0.7.2

Release Date: 2012-02-25

- Support for Python 2.6 added.
- Fixed Issue #1: Using `uidname=None` in `serialise.Sqlite` does not behave as documented.
- Documentation updated, and installation instructions added.

1.3.2 Norman-0.7.1

Release Date: 2012-02-12

- `Query.table` exposed, resulting in a major implementation change in `Query.add`. This function now exists for all queries, but raises an error if called when it cannot be used.
- `Query.add` can now be used for queries of whole tables.
- Add `AutoDatabase` and `AutoTable` classes.
- Make `Field.readonly` and `Field.unique` mutable.
- Allow `Field` definitions to be copied to another `Table`.
- Add `Database.delete` method.
- Allow a `None` return value from `serialise.Reader.create_record`.
- Fix issue in python2 where uuids cannot be converted to strings.
- Documentation updated.

1.3.3 Norman-0.7.0

Release Date: 2012-12-14

- Many internal changes in the way data is stored and indexed, centred around the introduction of two new classes, `Store` and `Index`.
- All fields are now automatically indexed. As a results the *index* parameter to `Field` objects falls away, and a new *key* argument is introduced.
- `Table` objects have a new attribute, ~‘Table._store‘, which refers to the `Store` used for the table. This may be changed when the

- The `serialise` framework has been completely overhauled and the API simplified. Extensive changes in this module.
- Add a new `CSV` serialiser.
- Add `validate.map` validator to convert values.
- Improved the string representation of `Query` and `Field` instances.
- Deprecated functionality removed

1.3.4 Norman-0.6.2

Release Date: 2012-09-03

- 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.
- Add a return value when calling `Query` objects.
- `Field` level validation added, including some validator factories.
- Add `validate.todatetime`, `validate.todate` and `validate.totime`.
- Deprecated the `tools` module.

1.3.5 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.

1.3.6 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

1.3.7 Norman-0.5.2

Release Date: 2012-04-20

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

1.3.8 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

1.3.9 Norman-0.5.0

Release Date: 2012-04-13

- First public release, repository imported from private project.

1.4 Data Structures

Contents

Norman data structures are build on four objects: `Database`, `Table`, `Field` and `Join`. In overview, a `Database` is a collections of `Table` subclasses. `Table` subclasses represent a tabular data structure where each column is defined by a `Field` and each row is an instance of the subclass. A `Join` is similar to a `Field`, but behaves as a collection of related records:

```
class Branch(Table):

    # Each branch knows its parent branch
    parent = Field(index=True)

    # Children are determined on the fly by searching for matching parents.
    children = Join(parent)
```

`AutoTable` is a special type of `Table` which automatically creates fields dynamically. This is used in conjunction with `AutoDatabase`, is is particularly useful when de-serialising from a source without knowing details of data in the source.

1.4.1 Database

class `norman.`**`Database`**

`Database` instances act as containers of `Table` objects, which are identified by name. `Database` supports the following operations.

Operation	Description
`db[name]`	Return a `Table` by name
`name in db`	Return `True` if a `Table` named *name* is in the database.
`table in db`	Return `True` if a `Table` object is in the database.
`iter(db)`	Return an iterator over `Table` objects in the database.

Databases are mainly provided for convenience, as a way to group related tables. Tables may beloong to multiple databases, or no database at all.

`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 in the database.

`delete`(*record*)

Delete a record from the database. This is a convenience function which simply calls record.__class__.delete(record), but also checks that the record does actually belong to the database. If not, a `NormanWarning` is raised, and the record is still deleted.

class `norman.`**`AutoDatabase`**

A subclass of `Database` which automatically creates `AutoTable` subclasses when a table is looked up by name. For example:

```
>>> adb = AutoDatabase()
>>> newtable = adb['NewTable']
>>> issubclass(newtable, AutoTable)
True
```

Apart from this, it behaves exactly the same as `Database`.

1.4.2 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. The class methods shown in `Table` are not visible to instances.

class `norman.`**`Table`**`(`***kwargs*`)`

Records are created by instantiating a `Table` subclass. Tables are defined by subclassing `Table` and adding `fields` to it. For example:

```
>>> class MyTable(Table):
...     field1 = Field()
...     field2 = Field()
```

`Field` names should not start with _, as these names are generally reserved for internal use. `Fields` and `Joins` may also be added to a `Table` after the `Table` is created, but cannot be shared between tables. If a `Field` which already belongs to a table is assigned to another table, a copy of it is created. The same cannot be done with a `Join`, since the behaviour of this would be unclear.

Records are created by simply instantiating the table, optionally with field values as keyword arguments:

```
>>> record = MyTable(field1='value', field2='other value')
```

The following class methods are supported by `Table` objects, but not by instances. Tables also act as a collection of records, and support the following sequence operations:

Operation	Description
`len(t)`	Return the number of records in `t`.
`iter(table)`	Return an iterator over all records in `t`.
`r in t`	Return `True` if the record `r` is an instance of (i.e. contained by) table `t`. This should always return `True` unless the record has been deleted from the table, which usually means that it is a dangling reference which should be deleted.

Boolean operations on tables evaluate to `True` if the table contains any records.

`_store`

A `Store` instance used as a storage backend. This may be overridden when the class is created to use a custom `Store` object. Usually there is no need to use this.

`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. They may be set at any time, but do not affect records already created until the record is next validated.

`delete``([`*records=None*`])`

Delete delete all instances in *records*. If *records* is omitted then all records in the table are deleted.

`fields``()`

Return an iterator over field names in the table

class `norman.`**`AutoTable`**`(`***kwargs*`)`

This is a special type of `Table` which automatically creates a new field whenever a value is assigned to an attribute which does not yet exist. This only occurs for attributes which do not start with `'_'`. This should be subclassed in exactly the same was as `Table`. Attempting to instantiate `AutoTable` directly will result in a `TypeError` being raised.

```
>>> class MyTable(AutoTable): pass
>>> record = MyTable(a=1)
>>> record.a
1
>>> isinstance(MyTable.a, Field)
True
>>> record.b = 2
>>> isinstance(MyTable.b, Field)
True
```

However:

```
>>> record._c = 3
>>> MyTable._c
Traceback (most recent call last):
    ...
AttributeError: '_c'
```

As with other `Table` classes, it is also possible to manually add fields or joins:

```
>>> MyTable.d = Field()
```

Records

Table instances, or records, are created by specifying field values as keyword arguments. Missing fields will use the default value (see `Field`). In addition to the defined fields, records have the following properties and methods.

`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 valid `uuid`. The default value is calculated using `uuid.uuid4` upon its first call. It is not necessary that the value be unique outside the session, unless required by the serialiser.

`Table.`**`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, an exception should be raised. Internal validate (e.g. uniqueness checks) occurs before this method is called, and a failure will result in a `ValidationError` being raised. For convenience, any `AssertionError` which is raised here is considered to indicate invalid data, and is re-raised as a `ValidationError`. 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.

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

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

Notes on Validation and Deletion

Data is validated whenever a record is added or removed, and there is the opportunity to influence this process through validation hooks. When a new record is created, there are three sets of validation criteria which must pass in order for the record to actually be created. The first step is to run the validators specified in `Field.validators`. These can change or verify the value in each field independently of context. The second validation check is applied whenever there are unique fields, and confirms that the combination of values in unique fields in actually unique. The final stage is to run all the validation hooks in `Table.hooks`. These affect the entire record, and may be used to perform changes across multiple fields. If at any stage an Exception is raised, the record will not be created.

The following example illustrates how the validation occurs. When a new record is created, the value is first converted to a string by the field validator, then checked for uniqueness, and finally the `validate` method creates the extra *parts* value.

```
>>> class TextTable(Table):
...     'A Table of text values.'
...
...     # A text value stored in the table
...     value = Field(unique=True, validators=[str])
...     # A pre-populated, calculated value.
...     parts = Field()
...
...     def validate(self):
...         self.parts = self.value.split()
...
>>> r = TextTable(value='a string')
>>> r.value
'a string'
>>> r.parts
['a', 'string']
>>> r = TextTable(value=3)
>>> r.value
'3'
>>> r = TextTable(value='3')
Traceback (most recent call last):
    ...
norman._except.ValidationError: Not unique: TextTable(parts=['3'], value='3')
```

When deleting a record, `Table.validate_delete` is first called. This should be used to ensure that any dependent records are dealt with. For example, the following code ensures that all children are deleted when a parent is deleted.

```
>>> class Child(Table):
...     parent = Field()
...
>>> class Parent(Table):
...     children = Join(Child.parent)
...
...     def validate_delete(self):
...         for child in self.children:
...             Child.delete(child)
...
>>> parent = Parent()
>>> child = Child(parent=parent)
>>> Parent.delete(parent)
>>> len(Child)
0
```

1.4.3 Fields

Fields are defined inside a `Table` definition as class attributes, and are used as record properties for instances of a `Table`. If the value of a field has not been set, then the special object `NotSet` is used to indicate this.

`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`**(*unique=False*, *default=NotSet*, *readonly=False*, *validators=None*, *key=None*)
A `Field` is used in tables to define attributes.

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

Fields may be created with a combination of properties as keyword arguments, including *default*, *key*, *readonly*, *unique* and *validators*.

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, provided the data stored supports them: ==, <, >, <=, >==, !=. 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.

Queries For more information of queries in Norman.

default
The value to use when nothing has been set (default: *NotSet*).

key
A key function used for indexing, similar to that used by `sorted`. All values returned by this function should be sortable in the same list. For example, if the field is known to contain a mixture of strings and integers, `str` would be a valid function, but `lambda x: x` would not, since a list of strings and integers cannot be sorted. *key* should raise `TypeError` for any value it cannot handle. These will be indexed separately, so that equality lookups are still optimised, but comparisons will not be supported. As an illustrative example, consider the following case which orders values by length:

```
>>> class T(Table):
...     value = Field(key=len)
...
>>> t1 = T(value='abc')
>>> t2 = T(value='defg')
>>> t3 = T(value=42)
>>> (T.value > 'xxx').one()  # Find values longer than 3 characters
T(value='abc')
>>> (T.value == 42).one()  # Find the numerical value 42
T(value=42)
>>> (T.value() > 42).one()  # len(42) raises TypeError
Traceback (most recent call last)
    ...
TypeError
```

The default implementation orders data by type first, then value, for the following types: `numbers.Real`, `str`, `bytes`. This might lead to unexpected results, since `42 < 'text'` will evaluate True.

NotSet values are handled slightly differently, and are never passed through this function. Comparison queries on *NotSet* will always fail.

name
This is the assigned name of the field and is set when it is added to the *Table*. This attribute is read-only.

owner
: This is the owning *Table* of the field and is set when it is added to the *Table*. This attribute is read-only.

readonly
: If `True`, prohibits setting the variable, unless its value is *NotSet* (default: `False`). This can be used with *default* to simulate a constant. This can be toggled to effectively lock and unlock the field.

unique
: `True` if records should be unique on this field (default: `False`). If more than one field in the table have this set then they are evaluated together as a tuple. If this is set after the field is created, all existing records in the table are evaluated and a *ValidationError* raised if there are duplicates.

validators
: A list of functions which are used as validators for the field. Each function should accept and return a single value (i.e. the value to be set), and should raise an exception if the value is invalid. The validators are called sequentially in the order specified, i.e. `newvalue = validator3(validator2(validator1(oldvalue)))`.

1.4.4 Joins

A *Join* 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)
>>> set(p.children) == {c1, c2}
True
```

In this example, `Parent.children` 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 function, returning a *Query* for a given instance of the owning table.

class `norman.`**Join**(**args*, ***kwargs*)
: Joins can be created in several ways:

Join(query=queryfactory) Explicitly set the query factory. `queryfactory` is a callable which accepts a single argument (i.e. the owning record) and returns a *Query*.

Join(table.field) This is the most common form, 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[, jointable]) 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, it is the name of the new join table.

Joins have the following attributes, all of which are read-only.

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. If a jointable does not yet exist then it is created, but not added to any database. If the two joins which define it have conflicting information, a *`ConsistencyError`* is raise.

`name`
This is the assigned name of the join and is set when it is added to the *`Table`*.

`owner`
This is the owning *`Table`* of the join and is set when it is added to the *`Table`*.

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

`target`
The target of the join, or `None` if the target cannot be found. This attribute is read only.

1.4.5 Exceptions and Warnings

Exceptions

class `norman.`**`NormanError`**
Base class for all Norman exceptions.

class `norman.`**`ConsistencyError`**
Raised on a fatal inconsistency in the data structure.

class `norman.`**`ValidationError`**
Raised when an operation resulting in table validation failing.

For now this inherits from *`NormanError`*, `ValueError` and `TypeError` to keep it backwardly compatible. This will change in version 0.7.0

Warnings

class `norman.`**`NormanWarning`**
Base class for all Norman warnings.

Currently all warnings use this class. In the future, this behaviour will change, and subclasses will be used.

1.4.6 Advanced API

Two structures, *`Store`* and *`Index`* manage the data internally. These are documented for completeness, but should seldom need to be used directly.

class `norman.`**`Store`**
Stores are designed to hide the implementation details and expose a consistent API, so that they can be switched out without any other changes to the table.

Tables are exposed as an array of cells, where each cell is identified by *`Table`* and *`Field`* instances. Cells are unordered, although implementations may order them internally.

The Store is tolerant of missing values. *`get`* will return defaults if the record requested does not exist. *`set`* will add a new record if the record does not exist.

`add_field`(*field*)
Called whenever a new field is added to the table.

`add_record`(*record*)
Called whenever a new record is created.

`clear`()
Delete all records in the store.

get (*record*, *field*)
: Return the value in a cell specified by *record* and *field*. This should respect any field defaults. If this is called with a record that has not been added, it will be added.

has_record (*record*)
: Return True if the record has an entry in the data store.

iter_field (*field*)
: Iterate over pairs of `(record, value)` for the specified field. This should respect any field defaults. If this is called with a field that has not been added, the behaviour is unspecified.

iter_records ()
: Return an iterator over all records in the data store.

iter_unset (*field*)
: Iterate over records which do not have a value set on *field*, that is, those for which `store.get(record, field)` will return `field.default`. This is used for managing indexes.

record_count ()
: Return the number of records in the table.

remove_field (*field*)
: Remove a field.

remove_record (*record*)
: Remove a record.

set (*record*, *field*, *value*)
: Set the data in a record.

setdefault (*field*, *value*)
: Called when the default value of a field in changed.

class `norman.`**Index** (*field*)
: An index stores records as sorted lists of `(keyvalue, record)` pairs, where *keyvalue* is a key based on the data cell value, determined by the return value of *`Field.key`*, which should always return the same, sortable type. If a return value cannot be sorted, then it is stored separately by its hash, and comparisons (except for equality checks) cannot be used with it. It is is not hashable, then it is stored by `id`, so equality checks will actually return identity matches. Note that *`NotSet`* is handled separately, and is never evaluated with *`Field.key`*. The default *`Field.key`* returns a tuple of `(type, keyvalue)` for recognised types. The implementation is:

```
def key(value):
    if isinstance(value, numbers.Real):
        return '0Real', value
    elif isinstance(value, str):
        return '1str', value
    elif isinstance(value, bytes):
        return '2bytes', value
    else:
        raise TypeError
```

The following examples show a few example of how this can be used:

```
>>> import re
>>> from norman import Table, Field
>>> class MyTable(Table):
...     numbers = Field(key=lambda x: re.findall('\d+', x))
...
>>> r1 = MyTable(numbers='number 1, numbers 2 and 3')
>>> r2 = MyTable(numbers='45 and 46')
>>> r3 = MyTable(numbers='a, b, c = 5, 6, 7')
>>> r4 = MyTable(numbers='no numbers here')
>>> set(MyTable.numbers > 'number 3') == set((r2, r3))
```

```
True
>>> set(MyTable.numbers < '1 or 2') == set((r4,))
True
```

clear()
Delete all items from the index.

insert(*value*, *record*)
Insert a new item. If equal keys are found, add to the right.

remove(*value*, *record*)
Remove first occurrence of `(value, record)`.

1.5 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

1.5.1 Examples

The following examples 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`:

```
def isvalid(record):
    return 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 can be iterated over to yield records. The query is not evaluated until a result is requested from it (including `len`). An existing query can be refreshed after the base data has changed by calling it as a function. The return value is the query iteself, so to ensure that the result is up to date, you could call:

```
latest_size = len(q7())
```

1.5.2 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**(*op*, **args*, ***kwargs*)

This object should never be instantiated directly, instead it should be created as the result of a `Field` comparison or by using the `query` function. The interface allows most operations permitted on sets, such as unions and intersections, but returns a new `Query` object instead of any results. 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`.

Queries evaluate to `True` if they contain any results, and `False` if they do not.

Calling a query forces it to be re-evaluated, and the query object is returned.

table

Return the table queried. If no single table is queried, `None` is returned.

add([*arg*, ***kwargs*])

Add a record based on the query criteria, and return the new record. There are two modes of operation for this method, depending on the query. For either mode, the query must be defined by a clear set of field values for a single `Table`. This includes queries such as `(MyTable.field1‘ == 1) & (MyTable.field2‘ == 2)` but not `MyTable.field1‘ > 1`.

The first mode accepts keyword arguments, which are combined with the parameters used to construct the query and passed to the table constructor. For example:

```
        ``((MyTable.a` == 1) & (MyTable.b` == 2)).add(c=3)``
```

evaluates to:

```
        MyTable(a=1, b=2, c=3)
```

The second mode is used when the query has been created by `field`. In this case, a single argument is expected which is the record to apply to the field. For example:

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

is the same as:

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

delete()

Delete all records matching the query from their 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.

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 `IndexError` is raised.

1.6 Serialisation

In addition to supporting the `pickle` protocol, Norman provides a framework for serialising and de-serializing databases to other formats through the *norman.serialise* module. Serialisation classes inherit *Reader*, *Writer* or *Serialiser*, which is a subclass of the first two provided for convenience.

Contents

1.6.1 Serialisation Framework

In addition to the *Reader*, *Writer* and *Serialiser* classes, a convenience function is provided to generate uids.

`norman.serialise.`**uid**()

Create a new uid value. This is useful for files which do not natively provide a uid.

Readers

class `norman.serialise.`**Reader**

An abstract base class providing a framework for readers.

Subclasses are required to implement *iter_source* and may re-implement any other methods to customise behaviour.

The entry point in the *read* method, which iterates of over records yielded by *iter_source*, identifies possible foreign keys by *isuid* and dereferences them by identifying loops and processing them with *create_group*. This method calls *create_record* to actually create the record.

read(*source*, *db*)

Read data from a *source* into *db*.

This converts each value returned by *iter_source* into a record using *create_record*. 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_group*.

iter_source(*source*, *db*)
Iterate over record in the source file, yielding tuples of `(table, data)` or `(table, uid, data)`. *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. If *uid* is omitted, then one is automatically generated using `uuid`.

Parameters

- **db** – The `Database` being read into.
- **source** – The data source, as specified in `read`.

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'`.

create_group(*records*)
Create a group of records. *records* is an iterable containing co-dependant records, i.e. records which cyclically reference each other. In many cases, *records* will contain only a single record.

Each record returned by *records* is a tuples of `(table, uid, data, cycles)`. The first three values are the same as those returned by `iter_source`, except that foreign uids in *data* have been dereferenced. *cycles* is a set of field names which contain the cyclic references.

The default behaviour is to remove the cyclic fields from *data* for each record, create the records using `create_record` and assign the created records to the cyclic fields.

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

create_record(*table*, *uid*, *data*)
Create a single record in *table*, using *uid* and *data*, as given by `iter_source`. This is called by `create_group`, so any foreign uid in *data* should have been dereferenced. The record created should be returned, or, if it cannot be created, `None` should be returned.

The default implementation simply calls `table(**data)` and sets the *uid*.

Writers

class `norman.serialise.`**Writer**
An abstract base class providing a framework for writers.

Subclasses are required to implement `context` and `write_record` and may re-implement any other methods to customise behaviour.

The entry point in the `write` method, which opens the target file with `context` and iterates of over records in the database with `iterdb`. Each record is converted to a simple python structure with `simplify` and written using `write_record`.

write(*targetname*, *db*)
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.

context(*targetname*, *db*)
Return a context manager which opens and closes the file, including and preparation and finalisation needed. A common implementation might be:

```
        def context(self, file):
            return open(file, 'w')
```

This can also be implemented using `contextlib.contextmanager`, which is useful for more complicated examples:

```
        @contextlib.contextmanager
        def context(self, targetname, db):
            fh = open(targetname, 'w')
            fh.write('### Header line ###')
            yield fh
            fh.write('### Footer line ###')
            fh.close()
```

`iterdb`(*db*)

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.

`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 is passed directly to `write_record`, so it can be anything recognised by it. This implementation returns a tuple of `(tablename, record._uid, record_dict)`.

`write_record`(*record*, *target*)

Write *record* to *target*.

This is called by `write` for every record yielded by `iterdb`. *record* is the values returned by `simplify` and *target* is the value returned by `context`.

Serialiser

class `norman.serialise.`**`Serialiser`**

This simply inherits from `Reader` and `Writer` to combine the functionality into one class for interfaces which support both reading and writing.

1.6.2 CSV

class `norman.serialise.`**`CSV`**(*uidname='_uid_'*, ***kwargs*)

This is a `Serialiser` which reads and writes to a collection of csv files.

Each table in the database is written to a separate file, which is managed by `csv.DictReader` and `csv.DictWriter`. Any extra initialisation parameters are passed to these. If this includes *fieldnames*, it should be a mapping of table to fieldnames. This defaults to a sorted list of table fields. This is only used for writing.

An additional field specified by *uidname* is prepended which contains the record's `_uid`. *uidname* may be empty or `None`, in which case uids are ignored and the field is omitted.

Since csv files can only contain text, all values are converted to strings when writing, and it is up to the database to convert them back into other objects when reading. The exception to this is uid keys, which are handled by the `Reader`. `NotSet` values are omitted when writing, and empty field values are converted to `NotSet` when reading.

The target and source specified in `read` and `write` should be a mapping of table name to file name, for example:

```
mapping = {Table1: '/path/table1.csv', Table2: '/path/table2.csv'}
CSV().read(mapping, db)
```

Any missing tables are omitted.

1.6.3 Sqlite

class norman.serialise.**Sqlite**(*uidname='_uid_'*)

This is a *Serialiser* which reads and writes to a sqlite database.

Each table is dumped to a sqlite table with the same field names. An additional field specified by *uidname* is included which contains the record's *_uid*. *uidname* may be empty or None, in which case uids are ignored and the field is omitted.

The sqlite database is created without any constraints. As described in the sqlite3 docs, under Python2, text is always returned as unicode.

1.7 Validators

This module provides some validators and validator factories, intended mainly for use in the *validate* parameter of *Field*s.

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

Return a *Field* validator returning func(value) if *value* is not *NotSet*. If *value* is *NotSet*, then it is returned and func is never called. 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):
    ...
ValidationError: 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, a *ValidationError* 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, a *ValidationError* is raised.

norman.validate.**istype**(*t*[, *t2*[, *t3*[, ...]]])

Return a validator which raises a *ValidationError* on an invalid type.

Parameters t – The expected type, or types.

`norman.validate.`**`map`**(*mapping*)

Return a validator which maps values to new values.

Parameters **`mapping`** – A dict mapping old values to new values.

If a value is passed which has no mapping then it is accepted unchanged. For example:

```
>>> validator = map({1: 'one', 0: NotSet})
>>> validator(1)
'one'
>>> validator(0)
NotSet
>>> validator(2)
2
```

`norman.validate.`**`settype`**(*t*, *default*)

Return a `Field` validator which converts the value to type *t*.

Parameters

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

The following three functions return validators which convert a value to a `datetime` object using a format string. See strftime() and strptime() Behavior for more information of format strings.

`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.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 a `ValidationError` 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.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 a `ValidationError` 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.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 a `ValidationError` is raised.

Python Module Index