Monk Documentation

Release 0.14.0-dev

Andrey Mikhaylenko

April 30, 2015

Contents

An unobtrusive data modeling, manipulation and validation library.

Supports MongoDB out of the box. Can be used for any other DB (or even without one).

CHAPTER 1

Installation

```
$ pip install monk
```

CHAPTER 2

Dependencies

Monk is tested against the following versions of Python:

- CPython 2.6, 2.7, 3.2, 3.4
- PyPy 2.0

Optional dependencies:

- The MongoDB extension requires *pymongo*.

CHAPTER 3

Documentation

See the complete documentation for details.

CHAPTER 4

Examples

4.1 Modeling

The schema is defined as a template using native Python data types:

```
# we will reuse this structure in examples below

spec = {
    'title': 'Untitled',
    'comments': [
        {
            'author': str,
            'date': datetime.datetime.utcnow,
            'text': str
        }
    ],
}
```

You are free to design as complex a document as you need. The *manipulation* and *validation* functions (described below) support arbitrary nested structures.

When this "natural" pythonic approach is not sufficient, you can mix it with a more verbose notation, e.g.:

```
title_spec = IsA(str, default='Untitled') | Equals(None)
```

There are also neat shortcuts:

```
spec = {
    'url': nullable(str),
    'status': one_of(['new', 'in progress', 'closed']),
    'comments': [str],
    'blob': None,
}
```

This could be written a bit more verbosely:

```
spec = {
    'url': IsA(str) | Equals(None),
    'status': Equals('new') | Equals('in progress') | Equals('closed'),
    'comments': ListOf(IsA(str)),
    'blob': Anything(),
}
```

It is even possible to define schemata for dictionary keys:

```
CATEGORIES = ['books', 'films', 'toys']
spec = {
    'title': str,
    opt_key('price'): float,    # key is optional; value is mandatory
    'similar_items': {
        one_of(CATEGORIES): [    # suggestions grouped by category
            {'url': str, 'title': str}
        ],
    }
}

# (what if the categories should be populated dynamically?
#  well, the schema is plain Python data, just copy/update on the fly.)
```

And, yes, you can mix notations. See FAQ.

This very short intro shows that Monk requires almost **zero learning to start** and then provides very **powerful tools when you need them**; you won't have to rewrite the "intuitive" code, only augment complexity exactly in places where it's inevitable.

4.2 Validation

The schema can be used to ensure that the document has correct structure and the values are of correct types.

```
from monk.validation import validate

# correct data: staying silent

>>> validate(spec, data)

# a key is missing

>>> validate(spec, {'title': 'Hello'})
Traceback (most recent call last):
   ...
MissingKeys: must have keys: 'comments'

# a key is missing in a dictionary in a nested list

>>> validate(spec, {'comments': [{'author': 'john'}]}
Traceback (most recent call last):
   ...
DictValueError: 'comments' value item #0: must have keys: 'text', 'date'


# type check; also works with functions and methods (by return value)

>>> validate(spec, {'title': 123, 'comments': []})
Traceback (most recent call last):
    ...
DictValueError: 'title' value must be str
```

Custom validators can be used. Behaviour can be fine-tuned.

The *validate()* function translates the "natural" notation to a validator object under the hood. To improve performance you can "compile" the validator once (using *translate()* function or by creating a validator instance in place) and use it multiple times to validate different values:

```
>>> from monk import *
>>> translate(str) == IsA(str)
True
>>> validator = IsA(str) | IsA(int)
>>> validator('hello')
>>> validator(123)
>>> validator(5.5)
Traceback (most recent call last):
    ...
AllFailed: must be str or must be int
```

4.3 Manipulation

The same schema can be used to create full documents from incomplete data.

```
from monk import merge_defaults

# default values are set for missing keys

>>> merge_defaults(spec, {})
{
    'title': 'Untitled',
    'comments': [],
}

# it's easy to override the defaults

>>> merge_defaults(spec, {'title': 'Hello'})
{
    'title': 'Hello',
    'comments': [],
}

# nested lists of dictionaries can be auto-filled, too.
# by the way, note the date.

>>> merge_defaults(spec, {'comments': [{'author': 'john'}]})
{
    'title': 'Untitled',
    'comments': [
        {
            'author': 'john',
            'date': datetime.datetime(2013, 3, 3, 1, 8, 4, 152113),
            'text': None,
        }
    ]
}
```

CHAPTER 5

Object-Document Mapping

The library can be also viewed as a framework for building ODMs (object-document mappers). See the MongoDB extension and note how it reuses mixins provided by DB-agnostic modules.

Here's an example of the MongoDB ODM bundled with Monk:

```
from monk.mongo import Document

class Item(Document):
    structure = {
        'text': unicode,
        'slug': unicode,
    }
    indexes = {
        'text': None,
        'slug': {'unique': True},
    }

# this involves manipulation (inserting missing fields)
item = Item(text=u'foo', slug=u'bar')

# this involves validation
item.save(db)
```

CHAPTER 6

Links

- Project home page (Github)
- Documentation (Read the Docs)
- Package distribution (PyPI)
- Questions, requests, bug reports, etc.:
  - Issue tracker
  - Direct e-mail (neithere at gmail com)

CHAPTER 7

Author

Originally written by Andrey Mikhaylenko since 2011.

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

http://github.com/neithere/monk/issues/

CHAPTER 8

Licensing

CHAPTER 9

Details

9.1 API reference

9.1.1 Validators

class monk.validators.**All**(*specs*, *default=None*, *first_is_default=False*)
Requires that the value passes all nested validators.

error_class
alias of AtLeastOneFailed

class monk.validators.**Any**(*specs*, *default=None*, *first_is_default=False*)
Requires that the value passes at least one of nested validators.

error_class
alias of AllFailed

class monk.validators.**Anything**
Any values passes validation.

class monk.validators.**Exists**(*default=None*)
Requires that the value exists. Obviously this only makes sense in special cases like dictionary keys; otherwise there's simply nothing to validate. Note that this is *not* a check against *None* or *False*.

class monk.validators.**IsA**(*expected_type*, *default=None*)
Requires that the value is an instance of given type.

class monk.validators.**HasAttr**(*attr_name*)
Requires that the value has given attribute.

class monk.validators.**Equals**(*expected_value*)
Requires that the value equals given expected value.

class monk.validators.**Contains**(*expected_value*)
Requires that the value contains given expected value.

class monk.validators.**InRange**(*min=None*, *max=None*, *default=NotImplemented*)
Requires that the numeric value is in given boundaries.

class monk.validators.**Length**(*min=None*, *max=None*, *default=NotImplemented*)
Requires that the value length is in given boundaries.

monk.validators.**ListOf**
alias of ListOfAll

class monk.validators.**ListOfAll**(*validator*, *default=None*)

Requires that the value is a *list* which items match given validator. Usage:

```
>>> v = ListOfAll(IsA(int) | IsA(str))

>>> v([123, 'hello'])

>>> v([123, 'hello', 5.5])
Traceback (most recent call last):
...
ValidationError: item #2: must be int or must be str
```

error_class

alias of AtLeastOneFailed

class monk.validators.**ListOfAny**(*validator*, *default=None*)

Same as ListOfAll but tolerates invalid items as long as there is at least one valid among them.

error_class

alias of AllFailed

class monk.validators.**DictOf**(*pairs*)

Requires that the value is a *dict* which items match given patterns. Usage:

```
>>> v = DictOf([
...     # key "name" must exist; its value must be a 'str'
...     (Equals('name'), IsA(str)),
...     # key "age" may not exist; its value must be an 'int'
...     (Equals('age') | ~Exists(), IsA(int)),
...     # there may be other 'str' keys with 'str' or 'int' values
...     (IsA(str), IsA(str) | IsA(int)),
... ])

>>> v({'name': 'John'})

>>> v({'name': 'John', 'age': 25})

>>> v({'name': 'John', 'age': 25.5})
Traceback (most recent call last):
...
DictValueError: 'age' value must be int

>>> v({'name': 'John', 'age': 25, 'note': 'custom field'})

>>> v({'name': 'John', 'age': 25, 'note': 5.5})
Traceback (most recent call last):
...
DictValueError: 'note' value must be str or must be int
```

Note that this validator supports Exists to mark keys that can be missing.

monk.validators.**translate**(*value*)

Translates given schema from "pythonic" syntax to a validator.

Usage:

```
>>> translate(str)
IsA(str)

>>> translate('hello')
IsA(str, default='hello')
```

class monk.validators.**MISSING**
Stub for Exists validator to pass if the value is missing (e.g. for dictionary keys).

9.1.2 Shortcuts

monk.shortcuts.**nullable**(*spec*)
Returns a validator which allows the value to be *None*.

```
>>> nullable(str) == IsA(str) | Equals(None)
True
```

monk.shortcuts.**optional**(*spec*)
Returns a validator which allows the value to be missing.

```
>>> optional(str) == IsA(str) | ~Exists()
True
>>> optional('foo') == IsA(str, default='foo') | ~Exists()
True
```

Note that you should normally `opt_key()` to mark dictionary keys as optional.

monk.shortcuts.**opt_key**(*spec*)
Returns a validator which allows the value to be missing. Similar to `optional()` but wraps a string in `Equals` instead of `IsA`. Intended for dictionary keys.

```
>>> opt_key(str) == IsA(str) | ~Exists()
True
>>> opt_key('foo') == Equals('foo') | ~Exists()
True
```

monk.shortcuts.**one_of**(*choices*, *first_is_default=False*, *as_rules=False*)
A wrapper for `Any`.

Parameters **as_rules** – *bool*. If *False* (by default), each element of *choices* is wrapped in the `Equals` validator so they are interpreted as literals.

Deprecated since version 0.13: Use `Any` instead.

9.1.3 Helpers

monk.helpers.**validate**(*spec*, *value*)
Validates given value against given specification. Raises an exception if the value is invalid. Always returns `None`.

In fact, it's just a very thin wrapper around the validators. These three expressions are equal:

```
IsA(str)('foo')
translate(str)('foo')
validate(str, 'foo')
```

Spec a validator instance or any value digestible by `translate()`.

Value any value including complex structures.

Can raise:

MissingKey if a dictionary key is in the spec but not in the value. This applies to root and nested dictionaries.

InvalidKey if a dictionary key is the value but not not in the spec.

StructureSpecificationError if errors were found in spec.

monk.helpers.**walk_dict**(*data*)

Generates pairs `(keys, value)` for each item in given dictionary, including nested dictionaries. Each pair contains:

keys a tuple of 1..n keys, e.g. `('foo',)` for a key on root level or `('foo', 'bar')` for a key in a nested dictionary.

value the value of given key or `None` if it is a nested dictionary and therefore can be further unwrapped.

9.1.4 Data manipulation

monk.manipulation.**merge_defaults**(*spec*, *value*)

Returns a copy of *value* recursively updated to match the *spec*:

- New values are added whenever possible (including nested ones).
- Existing values are never changed or removed.
  - Exception: container values (lists, dictionaries) may be populated; see respective merger functions for details.

The result may not pass validation against the *spec* in the following cases:

1. a required value is missing and the spec does not provide defaults;
2. an existing value is invalid.

The business logic is as follows:

- if *value* is empty, use default value from *spec*;
- if *value* is present or *spec* has no default value:
  - if *spec* datatype is present as a key in *mergers*, use the respective merger function to obtain the value;
  - if no merger is assigned to the datatype, use *fallback* function.

See documentation on concrete merger functions for further details.

Spec A "natural" or "verbose" spec.

Value The value to merge into the *spec*.

Examples:

```
>>> merge_defaults('foo', None)
'foo'
>>> merge_defaults('foo', 'bar')
'bar'
>>> merge_defaults({'a': 'foo'}, {})
{'a': 'foo'}
>>> merge_defaults({'a': [{'b': 123}]},
...                {'a': [{'b': None},
...                       {'x': 0}]})
{'a': [{'b': 123}, {'b': 123, 'x': 0}]}
```

monk.manipulation.**normalize_to_list**(*value*)

Converts given value to a list as follows:

- `[x]` → `[x]`
- `x` → `[x]`

monk.manipulation.**normalize_list_of_dicts**(*value*, *default_key*, *default_value=<class monk.manipulation.UNDEFINED at 0x7f65c0764460>*)

Converts given value to a list of dictionaries as follows:

- `[{...}]` → `[{...}]`
- `{...}` → `[{...}]`
- `'xyz'` → `[{default_key: 'xyz'}]`
- `None` → `[{default_key: default_value}]` (if specified)
- `None` → `[]`

Parameters **default_value** – only Unicode, i.e. *str* in Python 3.x and **only** *unicode* in Python 2.x

9.1.5 Modeling

DB-agnostic helpers to build powerful ODMs.

class monk.modeling.**DotExpandedDictMixin**

Makes the dictionary dot-expandable by exposing dictionary members via `__getattr__` and `__setattr__` in addition to `__getitem__` and `__setitem__`. For example, this is the default API:

```
data = {'foo': {'bar': 0 } }
print data['foo']['bar']
data['foo']['bar'] = 123
```

This mixin adds the following API:

```
print data.foo.bar
data.foo.bar = 123
```

Nested dictionaries are converted to dot-expanded ones on adding.

class monk.modeling.**TypedDictReprMixin**

Makes `repr(self)` depend on `unicode(self)`.

class monk.modeling.**StructuredDictMixin**

A dictionary with structure specification and validation.

structure

The document structure specification. For details see `monk.shortcuts.validate()`.

9.1.6 Exceptions

exception monk.errors.**AllFailed**

Raised when at least one validator was expected to pass but none did.

exception monk.errors.**AtLeastOneFailed**

Raised when all validators were expected to pas but at least one didn't.

exception monk.errors.**CombinedValidationError**

Raised when a combination of specs has failed validation.

exception monk.errors.**DictValueError**

Raised when dictionary value fails validation. Used to detect nested errors in order to format the human-readable messages unambiguously.

exception monk.errors.**InvalidKeys**
Raised whan the value dictionary contains an unexpected key.

exception monk.errors.**MissingKeys**
Raised when a required dictionary key is missing from the value dict.

exception monk.errors.**NoDefaultValue**
Raised when the validator could not produce a default value.

exception monk.errors.**StructureSpecificationError**
Raised when malformed document structure is detected.

exception monk.errors.**ValidationError**
Raised when a document or its part cannot pass validation.

9.1.7 MongoDB integration

This module combines Monk's modeling and validation capabilities with MongoDB.

Declaring indexes

Let's declare a model with indexes:

```
from monk.mongo import Document

class Item(Document):
    structure = dict(text=unicode, slug=unicode)
    indexes = dict(text=None, slug=dict(unique=True))
```

Now create a model instance:

```
item = Item(text=u'foo', slug=u'bar')
```

Save it and make sure the indexes are created:

```
item.save(db)
```

The last line is roughly equivalent to:

```
collection = db[item.collection]
collection.ensure_index('text')
collection.ensure_index('slug', unique=True)
collection.save(dict(item))  # also validation, transformation, etc.
```

class monk.mongo.**Document**(**args*, ***kwargs*)
A structured dictionary that is bound to MongoDB and supports dot notation for access to items.

Inherits features from:

- *dict* (builtin),
- TypedDictReprMixin,
- DotExpandedDictMixin,
- StructuredDictMixin and
- MongoBoundDictMixin.

class monk.mongo.**MongoBoundDictMixin**
Adds MongoDB-specific features to the dictionary.

collection
Collection name.

indexes
(TODO)

classmethod find(*db*, **args*, ***kwargs*)
Returns a `MongoResultSet` object.

Example:

```
items = Item.find(db, {'title': u'Hello'})
```

Note: The arguments are those of pymongo collection's *find* method. A frequent error is to pass query key/value pairs as keyword arguments. This is **wrong**. In most cases you will want to pass a dictionary ("query spec") as the first positional argument.

---

get_id()
Returns object id or `None`.

classmethod get_one(*db*, **args*, ***kwargs*)
Returns an object that corresponds to given query or `None`.

Example:

```
item = Item.get_one(db, {'title': u'Hello'})
```

get_ref()
Returns a *DBRef* for this object or `None`.

id
Returns object id or `None`.

remove(*db*)
Removes the object from given database. Usage:

```
item = Item.get_one(db)
item.remove(db)
```

Collection name is taken from `MongoBoundDictMixin.collection`.

save(*db*)
Saves the object to given database. Usage:

```
item = Item(title=u'Hello')
item.save(db)
```

Collection name is taken from `MongoBoundDictMixin.collection`.

class monk.mongo.**MongoResultSet**(*cursor*, *wrapper*)
A wrapper for pymongo cursor that wraps each item using given function or class.

Warning: This class does not introduce caching. Iterating over results exhausts the cursor.

ids()
Returns a generator with identifiers of objects in set. These expressions are equivalent:

```
ids = (item.id for item in result_set)

ids = result_set.ids()
```

> **Warning:** This method **exhausts** the cursor, so an attempt to iterate over results after calling this method will *fail*. The results are *not* cached.

9.2 Glossary

natural spec Readable and DRY representation of document specification. Must be converted to *detailed spec* before validation.

detailed spec Rule-based verbose representation of document specification.

9.3 Frequently Asked Questions

9.3.1 What are the primary use cases of Monk?

- Entity schema for schema-less data storages (NoSQL, JSON, YAML, CSV, whatever)
- Publish/get data via RESTful API
- Safely implement ETL
- Process user input
- ODMs (object-document wrappers) as syntax sugar upon Monk

9.3.2 Why would I want to use Monk?

Monk allows to quickly prototype schemata using plain Python structures. It is very powerful, flexible, transparent and unobtrusive; all the power is accessible through a few functions and the rule class (which you may not even notice unless your use cases are demanding enough). It is possible to write a simple script, build a large project or a library upon Monk.

If in doubt, I encourage you to use Monk. If it's not enough, read the docs and make sure you squeeze the maximum from the rules.

When *not* to use Monk? Easy: when the case is particularly complex, major additions should be done but a dedicated tool already exists. For instance, it is possible to build an alternative to WTForms upon Monk but why? Well, who knows.

9.3.3 What problems does Monk solve?

- Validation of arbitrary data
- Populating incomplete documents with regard to a schema
- Defining schemata in a universal way for different backends
- Keeping it simple

9.3.4 How does Monk solve these problems?

1. defines two mutually complementary schema conventions:
   - *natural spec* (simple and pythonic)
   - *detailed spec* (more verbose and powerful)
2. validates data against specs;
3. manipulates data with regard to specs;
4. provides an optional ODM for MongoDB based on the above-mentioned features.

9.3.5 Is Monk a standalone tool or a building block for ODMs?

Both.

Monk ships with an integrated MongoDB ODM. It's usable and can serve as an example of how ODMs can be built upon Monk.

In many cases ODM is not needed at all; the validation and manipulation features of Monk are enough even for complex applications.

9.3.6 Is Monk validation usable only for documents?

No. It is possible to validate any value, be it a string, a number, a custom class instance or a full-blown document with multiple nested dictionaries.

9.3.7 Why are there two ways of schema declaration?

The "natural" way is intuitive and requires very little knowledge about how Monk works. It's about using plain Python data types to design a template for the value. The resulting declaration is clear and readable.

The "verbose" way is less readable, involves more boilerplate code and requires additional knowledge. However, this approach enables fine-tuning and inline customization that the "natural" way cannot achieve.

To sum up: a quick start with complexity evolving along with the needs.

9.3.8 Can I mix the "natural" and "verbose" declarations?

Yes. The `validate()` function will convert the "natural" declarations into rules; if it finds a ready-made rule, it just accepts it.

For example:

```
spec = {
    'name': str,
    'age': IsA(int, default=18) | Equals(None),
    'attrs': [IsA(str) | IsA(int) | NotExists()]
}

validate(spec, { ... your data goes here ... })
```

The validators can be combined with other validators or with non-validators; in the latter case the right-hand value is first translated to a validator:

```
>>> full = (InRange(0,9) & IsA(int)) | IsA(str, default='unknown')
>>> brief = InRange(0,9) & int | 'unknown'
>>> full == brief
True
```

Of course the technique only works if the left-hand value is a validator.

9.3.9 Which notation should I prefer?

The one that's more readable in given use case.

Consider these alternative notations:

```
IsA(str, default='foo')
```

```
'foo'
```

The second one is definitely more readable. But if the schema is mostly written in verbose notation, adding bits in the natural one may increase confusion:

```
(IsA(int) | IsA(float)) & InRange(0,5) | IsA(str)
```

```
(IsA(int) | float) & InRange(0,5) | str

# the last one is cleaner and shorter but the mind fails to correctly
# group the items using visual clues
```

When in doubt, stick to the Zen of Python!

It's also worth noting that natural specs are anyway translated to verbose specs, so if you happen to generate the specs a lot, skip the additional layer. Or, even better, build the schema once (including translation) and only call the resulting validator for every value.

9.3.10 How "Natural" Declarations Map To "Verbose" Style?

In most cases the "natural" style implies providing a class or instance.

A **type or class** means that the value must be an instance of such:

natural	verbose
`int`	`IsA(int)`
`str`	`IsA(str)`
`list`	`IsA(list)`
`dict`	`IsA(dict)`
`MyClass`	`IsA(MyClass)`

An **instance** means that the value must be of the same type (or an instance of the same class) *and* the spec is the default value:

natural	verbose
`5`	`IsA(int, default=5)`
`'hello'`	`IsA(str, default='hello')`
`[]`	`ListOf([])`
`{}`	`DictOf([])`
`MyClass('x')`	`IsA(MyClass, default=MyClass('x'))`

Note that the *dict*, *list* and *MyClass* specs describe containers. It is possible to nest other specs inside of these. Not all containers are handled by Monk as such: only *dict* and *list* are supported at the moment. However, it all depends on

validators and it's possible to write a validator and drop it into any place in the spec. Such validators are the building blocks for complex multi-level schemata. If the "natural" spec is a non-empty container, the `translate()` function wraps it in a relevant validator using its special requirements:

natural	verbose
`[str]`	`ListOf(IsA(str))`
`{str: int}`	`DictOf([ (IsA(str), IsA(int) ])`

Note: On defaults as dictionary keys

WARNING: THIS SECTION APPLIES TO v0.12 BUT IS OUT OF DATE AS OF v0.13

TODO: UPDATE

Normally default values are only used in *manipulation*. In dictionaries they are also important for *validation*. Consider this:

```
spec_a = {str: int}
spec_b = {'a': int}
```

The spec *spec_a* defines a dictionary which may contain any number of keys that must be of type `type('a')` → *str*.

The spec *spec_b* requires that the dictionary contains a single key `'a'` and nothing else. So, *a* in this case is not a default value but rather a precise requirement.

The keys may be marked as optional and be multiple:

```
spec_c = {'a': int, optional('b'): float}
```

It's also possible to allow arbitrary keys of different types:

```
spec_d = {str: int, tuple: float}
```

Of course the key datatype must be hashable.

Note: On optional dictionary keys vs. values

WARNING: THIS SECTION APPLIES TO v0.12 BUT IS OUT OF DATE AS OF v0.13

TODO: UPDATE

Consider this spec:

```
spec_a = {
    'a': int,
    'b': optional(int),
    optional('c'): int,
    optional('d'): optional(int),
}
```

It should not be surprising that the inner specs are interpreted thusly:

a key must be present; value must be of *int* type

b key must be present; value must be of *int* type or may be *None*

c key may exist or not; if yes, the value must be of *int* type

d key may exist or not; value must be of *int* type or may be *None*

9.3.11 Do I need MongoDB to use Monk?

No. Monk comes with a MongoDB extension but since v.0.6 the dependency is optional.

9.3.12 Does Monk support DBRefs and other MongoDB features?

Yes. However, there's room for improvement. Feel free to submit your use cases.

9.3.13 Is Monk stable enough?

It depends on requirements. Feel free to use Monk in personal apps and prototypes. Avoid using it in production until v1.0 is out (or expect minor changes in the API and therefore ensure good coverage of your code).

quality More than 90% of code is covered by tests. The key modules are fully covered.

stability The API is still evolving but the core was considered stable since v0.7. Even serious changes under the hood barely affect the public interface.

Even after v0.13 featured a complete rewrite, the top-level API (the "natural" notation) was almost intact.

speed There are real-life projects where declarative Monk-powered code processes large volumes of data with roughly the same speed as the previous purely imperative version of rather good quality. Monk makes it easy to prepare various operations as validator objects and simply apply them to the value so the overall overhead is minimal; in some cases Monk-based logic performs even better than "classic" implementation. This is mostly related to tasks where you need to make decisions based on rules, tables, etc. being unable to hard-code all the logic once and forever. Of course the speed rather depends on design: one could arrange the "classic" code better and cache a mixture of data and functions to save on lookups, but here you have that mixture with a nice and clean API, "Monk" it is called.

9.3.14 What are the alternatives?

See *Similar Projects*.

9.4 Similar Projects

Below is a list of projects that share one or more major goals with Monk.

The descriptions may be a bit too critical, but that's only because of the inevitable competition between Monk and these projects; Monk aims to be better and the list shows how it is better. I mean, what's the point of creating a project if not to make a solution that would incorporate the strong points and address the weaknesses of its predecessors? Oh well.

Note: Spotted an error?

Please excuse me for possible false assumptions about the projects being described; if you find an error, please don't hesitate to poke me via e-mail or the issue tracker (as this would be a proper documentation issue).

9.4.1 Schema Definition

Monk See `monk.schema`.

MongoKit Simple and pythonic, very similar to Monk (actually, Monk's "natural" DSL was inspired by that of MongoKit). However, everything is tightly bound to MongoDB (not to mention the lacking possibility to work with plain data without ODMs); the required flag, default values and custom validators are defined on the root level, duplicating the structure in each case.

MongoKit example:

```
class Spec(Document):
    structure = {
        'foo': int,
        'bar': str,
        'baz': {
            'quux': str
        }
    }
    default_values = {
        'foo': 5,
        'quux': 'flux'
    }
    required = ['foo', 'baz.quux']

Spec(**data).validate()
```

Semantically equivalent schema in Monk (without classes):

```
spec = {
    'foo': 5,
    'bar': nullable(str),
    'baz': {
        'quux': 'flux'
    }
}

validate(spec, data)
```

Very similar support (and notation) for nested lists and dicts; also supports nested tuples.

MongoEngine Very verbose Django-like syntax, traditional for ORMs.

MongoEngine example:

```
class User(Document):
    name = StringField(required=True)

class Comment(EmbeddedDocument):
    author = ReferenceField(User, required=True)
    content = StringField(required=True, max_length=30)
    added = DateTimeField(required=True, default=datetime.datetime.utcnow)

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

Semantically equivalent schema in Monk (without classes):

```
user_schema = {'name': str}

comment_schema = {
    'author': ObjectId,    # see monk.modeling; still needs work
    'content': IsA(str) & Length(max=30),
    'added': datetime.datetime.utcnow,
}

post_schema = {
    'title': str,
    'author': ObjectId,
    'tag': [ optional(str) ],
    'comments': [ optional(comment_schema) ]
}
```

The *FooField* layer can be added on top of the normal Monk syntax if needed.

MongoEngine is tightly bound to MongoDB and provides many database-specific features which are not present in Monk (e.g. defining deletion policy of referred documents).

Colander Declarative and imperative schema declaration (for "static" and dynamically generated data models). Very verbose, class-based. Similar to traditional ORMs but more flexible and generalized: there are tuple/mapping/sequence schemata with nested "schema nodes" and/or other schemata. Supports inheritance.

Colander example (from tutorial):

```
import colander

class Friend(colander.TupleSchema):
    rank = colander.SchemaNode(colander.Int(),
                               validator=colander.Range(0, 9999))
    name = colander.SchemaNode(colander.String())

class Phone(colander.MappingSchema):
    location = colander.SchemaNode(colander.String(),
                                   validator=colander.OneOf(['home', 'work']))
    number = colander.SchemaNode(colander.String())

class Friends(colander.SequenceSchema):
    friend = Friend()

class Phones(colander.SequenceSchema):
    phone = Phone()

class Person(colander.MappingSchema):
    name = colander.SchemaNode(colander.String())
    age = colander.SchemaNode(colander.Int(),
                              validator=colander.Range(0, 200))
    friends = Friends()
    phones = Phones()
```

Semantically equivalent schema in Monk (without classes):

```
from monk import Rule
from monk import validators

friend_schema = {
    'rank': IsA(int) & InRange(0, 9999),
    'name': str
}
```

```
phone_schema = {
    'location': IsA(str) & one_of(['home', 'work']),
    'number': str,
}
person_schema = {
    'name': str,
    'age': IsA(int) & InRange(0, 200),
    'friends': [ friend_schema ],
    'phones': [ phone_schema ],
}
```

Note: Tuples

Monk does not support fixed-size tuples with named arguments out of the box. However, it's easy to write a validator for this specific use case.

9.4.2 Validation

Monk See `monk.validators`.

MongoKit Type validation (extensible with custom types). All validators beyond types belong in a separate dictionary which mostly duplicates the schema dictionary. The list of required fields (with names in a MongoDB-ish dot notation, i.e. `foo.$unicode.bar`) must be defined in yet another place. This approach implies noticeable redundancy for relatively complex documents.

The Document class also has an overloadable `validate()` method which makes sense for simultaneous multi-field validation. In Monk you would simply call the normal and a custom validation functions one after another (or overload the method in a similar way if using modeling).

MongoEngine Validation is integrated into *FooField* classes and triggered on save. Only very basic validators (required, unique, choices) are tunable. Custom validation implies custom field classes. For each field. Ouch.

Colander A *SchemaNode* instance validates a value by a) the *SchemaType* bound to its class, and b) by an optional validator passed to the constructor (a selection of common validators is bundled in the *colander* module).

It takes time to even grasp the terminology, not to mention the code (which is very clean and well-documented but presented as a 2K+ LOC module that handles all flavours of schema declaration + validation + serialization).

9.4.3 Manipulation

Monk See `monk.manipulation`.

MongoKit Data manipulation mostly embraces conversion between Python types and MongoDB internal representation (via PyMongo). This can be tuned with "Custom Types" that handle both manipulation and validation.

It is unknown whether the list of default values supports callables.

MongoEngine Mostly embraces conversion between Python types and MongoDB. This is always implemented by *FooField* classes that handle both manipulation and validation.

Supports callable defaults.

Colander Focused on (de)serialization (which is closer to normalization):

```
>>> class MySchema(colander.MappingSchema):
...     age = colander.SchemaNode(colander.Int())
...
>>> schema = MySchema()
```

```
>>> schema.deserialize({'age': '20'})
{'age': 20}
```

Supports optional preparer functions per node to prepare deserialized data for validation (e.g. strip whitespace, etc.).

In general, this functionality is very useful (and not bound to a concrete storage backend). Not sure if Monk should embrace it, though.

SchemaNode also contains utility functions to manipulate an *appstruct* or a *cstruct*:

- (un)flattening a data structure:

  ```
  >>> schema.flatten({'a': [{'b': 123}]})
  {'a.0.b': 123}
  ```

- accessing and mutating nodes in a data structure:

  ```
  rank = schema.get_value(appstruct, 'friends.2.rank')
  schema.set_value(appstruct, 'friends.2.rank', rank + 5000)
  ```

  (which resembles the MongoDB document updating API)

9.4.4 Modeling

Monk See `monk.modeling`.

lightweight schema Yes. The schema is not bound to any kind of storage or form. It can be — just add another layer on top.

reusable parts Yes. The Document class can be used right away, subclassed or be built anew from the components that were designed to be reusable.

This makes Monk a good building block for custom ODMs.

dot-expanded dictionary behaviour Yes.

polymorphism (document inheritance) Not yet.

MongoKit

lightweight schema No. The Document class is bound to a MongoDB collection.

reusable parts No. The underlying functions are not intended to be used separately.

dot-expanded dictionary behaviour Yes.

polymorphism (document inheritance) Yes.

MongoEngine

lightweight schema No. The Document class is bound to a MongoDB collection.

reusable parts No. The underlying functions are not intended to be used separately.

dot-expanded object behaviour Yes.

polymorphism (document inheritance) Yes.

Colander No modeling as such.

9.4.5 MongoDB extension

Monk See `monk.mongo`.

MongoKit Tightly bound to MongoDB on all levels. The document class is bound to a collection (which I found problematic in the past but generally this may be good design). Very good integration. PyMongo is accessible when needed (like in Monk). Keeps the data clean from tool-specific metadata (like Monk). In general, MongoDB support is superior compared to that of Monk but both use PyMongo so the basic functionality is exactly the same. The choice depends on given project's use cases.

MongoEngine Seems to be on par with MongoKit.

CHAPTER 10

Indices and tables

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

m

A

C

D

E

F

G

H

I

L

M

N

O

R

S

T

V

W