The Axiom Book Documentation

Release 0

Laurens Van Houtven

November 15, 2012

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 What is Axiom?

Simply put, Axiom is a Python object database written on top of SQLite.

However, to just call it an "object database" doesn't quite do it justice. Axiom has many other cool features, including but not limited to:

- powerups, allowing you to pretty much tack arbitrary behavior on to stores and items within those stores alike
- scheduling, allowing you to efficiently persist tasks to be executed at some point in the future
- upgrading, allowing you to transparently upgrade items in stores through pretty much any schema change imaginable

Despite all of this, Axiom manages to be small enough to fit in your head, a property often sorely missed in more complex systems. For example, there's an obvious one-to-one mapping from any Axiom item definition to a database schema.

1.2 Why this book?

Because Axiom is not always given the attention it deserves. Also, Axiom's many useful features and properties are not always obvious to the casual observer, and warrant some more explaining.

Additionally, a catastrophic fate has befallen the server that originally hosted all of the code and documentation, meaning documentation is often harder to find than it needs to be. This book aims to alleviate that problem.

CHAPTER

TWO

INSTALLATION

2.1 Using pip

Axiom is reasonably easy to install using pip. However, it does require its dependency, Epsilon, during `setup.py egg-info`. That means that it needs to be avaiable before pip tries to install Axiom. In short, you need to do:

```
$ pip install Epsilon
$ pip install Axiom
```

Unfortunately, due to the way pip works, `pip install Epsilon Axiom` or putting both of them in a requirements file and installing it using `pip -r` does *not* work.

CHAPTER

THREE

ITEMS

An item is a persisted object with one or more attributes that make up a schema.

3.1 Attributes

Axiom comes with the usual suspects of attributes:

- `text` for (Unicode) text
- `bytes` for bytes
- `integer` for integer values
- `pointXdecimal` (X varying from 1 to 10) for decimals of varying precision
- `ieee754_double` for floating point values
- `timestamp` for absolute points in time

It also comes with a few slightly more advanced ones:

- `reference` which allows you to store a reference to any other Axiom item (not just those of a particular type)
- `textlist` which allows you to store lists of (Unicode) text in one field
- `path` which allows you to store both relative and absolute paths
- `inmemory` which allows you to store non-persisted/non-persistable state on the item instance

All of these will be covered throughout the book.

3.2 Creating items and accessing attributes

Creating an item class is done by subclassing `axiom.item.Item`, and assigning at least one attribute from `axiom.attributes` to a name in the class body:

```
from axiom import attributes, item

class Person(item.Item):
    """
    A person.
    """
    name = attributes.text()
```

You can then create instances of that item by calling the `Item` class and specifying the attributes as keyword arguments. You can access the attributes on the item just like regular attributes:

```
>>> alice = Person(name=u"Alice")
>>> assert alice.name == u"Alice"
```

3.3 Static, strong typing

Axiom item attributes are not just statically typed but also strongly typed. For example, you can't assign a bytestring to a `text` attribute:

```
>>> Person(name="a bytestring")
Traceback (most recent call last):
  ...
ConstraintError: attribute [Person.name = text()] must be (unicode string without NULL bytes); not 's
>>> alice = Person(name=u"Alice")
>>> alice.name = "another bytestring"
Traceback (most recent call last):
  ...
ConstraintError: attribute [Person.name = text()] must be (unicode string without NULL bytes); not 's
```

3.4 Defaults and unspecified values

You're allowed to not specify an attribute, which sets it to `None`:

```
>>> anonymous = Person()
>>> assert anonymous.name is None
```

If you don't want to allow `None` as a value, set `allowNone=False` on the attribute:

```
from axiom import attributes, item

class Coin(item.Item):
    """
    A coin.
    """
    # attributes.money is the same thing as point4decimal
    value = attributes.money(allowNone=False)

>>> Coin()
Traceback (most recent call last):
  ...
TypeError: attribute [Coin.value = money()] must not be None
>>> quarter = Coin(value=decimal.Decimal("0.25"))
```

You can also have default values. For example, we could have a petting zoo with bunnies, and bunnies start out having been petted zero times:

```
from axiom import attributes, item

class Bunny(item.Item):
    """
    A bunny in a petting zoo.
    """
    timesPetted = attributes.integer(default=0)
```

```
>>> thumper = Bunny()
>>> assert thumper.timesPetted == 0  # Aww :-(
>>> thumper.timesPetted += 1
>>> assert thumper.timesPetted == 1  # Yay :-)
```

Sometimes a default value isn't enough, and you need a default value factory that gets called when the item gets created. Let's recite the alphabet:

```
import string
from axiom import attributes, item

letters = string.ascii_lowercase.decode("ascii")

class Letter(item.Item):
    """
    A letter in the alphabet being recited.
    """
    value = attributes.text(defaultFactory=iter(letters).next)
    # This creates an iterator over the list, and takes its ``next`` method.
    # Calling this method will produce the letters in sequence.

>>> a, b, c = Letter(), Letter(), Letter()
>>> assert a.value == "a"
>>> assert b.value == "b"
>>> assert c.value == "c"
```

CHAPTER

FOUR

INDICES AND TABLES

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