The Axiom Book

Release 0

Laurens Van Houtven

May 26, 2013

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 available before pip tries to install Axiom. To make matters worse, Epsilon, in turn, requires Twisted to be available for `setup.py egg-info`.

In short, you need to do:

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

Unfortunately, due to the way pip works, `pip install Twisted 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
>>> 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
```

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"
```

3.5 Type names

Axiom item classes have a "type name", which is the unique name used to identify instances of it in an Axiom store, and distinguish them from other item classes. If you don't explicitly specify a type name, one is automatically generated based on the name of the class and the module it's in. These are put into lower case and concatenated with underscores. This is done because the type name will be a SQLite table name, and therefore has to be a valid SQL identifier.

```
>>> Bunny.typeName
'bunny_bunny'
```

The module is called `bunny`. In this documentation, there are no modules above it; if your module is in a package, it could be something like `'top_mid_leaf_class'`.

There are some benefits to specifying an explicit type name. For example, you will be able to change the structure of your package, or move items to different, independent packages.

You can specify the type name using the `typeName` class attribute:

```
>>> class MobileBunny(item.Item):
...     typeName = "mobile_bunny"
...     timesPetted = attributes.integer()
>>> MobileBunny.typeName
'mobile_bunny'
```

Note: There's no ubiquitous standard for type names. These will map to SQL tables, so whatever your preference is for those might win out.

CHAPTER

FOUR

STORES

So far, we've created items, but never actually put them in a database. Axiom "databases" are called stores.

To create a store, just call `axiom.store.Store()`. By default, that will create an in-memory store. Obviously, in-memory stores aren't persisted.

```
>>> inMemoryStore = Store()
```

To create a persisted store, pass either a path as a string, or a `twisted.python.filepath.FilePath` to the store constructor:

```
>>> persistedStore = Store("mystore")
```

4.1 Adding objects to stores

You can either set an item's store when you create it, or by setting its store attribute later:

```
>>> store = Store()
>>> alice = Person(store=store, name=u"Alice")
>>> bob = Person(name=u"Bob")
>>> bob.store = store
>>> assert alice.store is bob.store is store
```

4.2 Basic querying

Once we have some objects in a store, we can query them. Simple queries take the item class you want to get instances of, and the usual querying suspects:

- comparisons (filters)
- limits and offsets
- sorting

The result returned by a query is a generator, so we'll call `list` on it to show its contents.

```
>>> people = list(store.query(Person, sort=Person.name.ascending))
>>> assert people == [alice, bob]
>>> person, = store.query(Person, sort=Person.name.ascending, limit=1)
>>> assert person is alice
>>> person, = store.query(Person, sort=Person.name.ascending, limit=1, offset=1)
>>> assert person is bob
```

4.3 Finding unique objects

Since finding exactly one object is such a common pattern, there's an easier shorthand for it called `findUnique`.

```
>>> person = store.findUnique(Person, Person.name == u"Alice")
>>> assert person is alice
```

Since you're expecting to find a single object, `findUnique` will raise an exception if there is more than one:

```
>>> secondAlice = Person(store=store, name=u"Alice")
>>> store.findUnique(Person, Person.name == u"Alice")
Traceback (most recent call last):
...
DuplicateUniqueItem: (person.Person.name = u'Alice', [...])
```

Similarly, it will raise an exception if there are none:

```
>>> store.findUnique(Person, Person.name == u"Nobody")
Traceback (most recent call last):
...
ItemNotFound: person.Person.name = u'Nobody'
```

4.4 Creating items unless they exist already

Sometimes, you want to create an object if it does not exist, or update an object if it does. Axiom calls this `findOrCreate`.

```
>>> newAlice = store.findOrCreate(Person, name=u"Alice")
>>> assert newAlice is alice, "returns the existing object"
>>> charlie = store.findOrCreate(Person, name=u"Charlie")
```

4.5 Getting parts of the data

Often, you only want part of the data instead of the entire stored item using `getColumn`. (This will once again produce a lazy iterator, so we will use `list` to consume it.)

```
>>> bugs = Bunny(store=store, timesPetted=1)
>>> fluffy = Bunny(store=store, timesPetted=2)
>>> thumper = Bunny(store=store, timesPetted=3)
>>> query = store.query(Bunny, sort=Bunny.timesPetted.ascending)
>>> assert list(query.getColumn("timesPetted")) == [1, 2, 3]
```

4.6 Aggregate data: sums, counts, averages

You can sum over all the attributes in a store:

```
>>> assert store.sum(Bunny.timesPetted) == 6
```

Usually, it's more useful to do it over a query than over all items:

```
>>> query = store.query(Bunny, Bunny.timesPetted > 1).getColumn("timesPetted")
>>> assert query.sum() == 5
```

While you could also do this by using Python's builtin `sum` function to get the same result. The principal difference is that with the query method, the summing is actually done inside the database.

You can also count how many items there are in a query.

```
>>> assert query.count() == 2
```

You can also average values:

```
>>> assert query.average() == 2.5
```

Notice how we're re-using the query object. Queries are lazy: they're only executed when you actually need an item. For example, if an item is created or modified so that it suddenly is affected by the query, you get the appropriate result:

```
>>> bugs.timesPetted += 1
>>> assert query.count() == 3
```

CHAPTER

FIVE

MORE ADVANCED ATTRIBUTE COMPARISONS

Axiom provides a few tools for more advanced querying patterns.

5.1 Checking if something is part of a fixed set

A common pattern is trying to find all items where some attribute is a member (or not a member) of some fixed set. Axiom provides `oneOf` and `notOneOf` for this.

To demonstrate this, let's create a store with some people in it:

```
>>> store = Store()
>>> audrey = Person(store=store, name=u"audreyr")
>>> pydanny = Person(store=store, name=u"pydanny")
>>> lvh = Person(store=store, name=u"lvh")
```

There's a secret club for people who like coffee ice cream:

```
>>> secretClubMemberNames = [u"pydanny", u"lvh"]
```

We can create a comparison, which we'll call `inSecretClub`, that checks if a person's name is in the list of secret club member names. Of course, you could just plug this into the query expression directly.

```
>>> inSecretClub = Person.name.oneOf(secretClubMemberNames)
>>> list(store.query(Person, inSecretClub, sort=Person.name.ascending))
[Person(name=u'lvh', storeID=3)@..., Person(name=u'pydanny', storeID=2)@...]
```

Additionally, there's a super secret club for people who like pistachio flavored ice cream. Only lvh likes pistachio ice cream. If we want to query all the people who *don't*, we'd use `notOneOf`:

```
>>> doesNotLikePistachio = Person.name.notOneOf([u"lvh"])
>>> list(store.query(Person, doesNotLikePistachio, sort=Person.name.ascending))
[Person(name=u'audreyr', storeID=1)@..., Person(name=u'pydanny', storeID=2)@...]
```

And that's all there is to it.

`notOneOf` and `oneOf` are supported by almost all attributes, except `inmemory`. That said, comparing floats (`ieee754_double`) is probably not what you wanted – but that's inherent to floats, not anything specific to Axiom.

CHAPTER

SIX

TRANSACTIONS

Like any database worth its salt, SQLite supports transactions. Axiom exposes these through the store's `transact` method.

The transact method takes a function. That function will be run in a database transaction. The changes in the function happen atomically (that is: either they entirely do or entirely don't).

```
>>> store = Store()
>>> thumper = Bunny(store=store)
>>> bugs = Bunny(store=store)
>>> def petBunny():
...     bugs.timesPetted += 1
...     thumper.timesPetted += 1
>>> store.transact(petBunny)
>>> assert thumper.timesPetted == bugs.timesPetted == 1
```

Any uncaught exception raised by the transacted function will be reraised. When this happens, the changes are reset to what they were before the transaction was initiated.

```
>>> assert thumper.timesPetted == 1
>>> def runIntoProblems():
...     thumper.timesPetted += 1
...     raise RuntimeError("fluffiness overload")
>>> store.transact(runIntoProblems)
Traceback (most recent call last):
...
RuntimeError: fluffiness overload
>>> assert thumper.timesPetted == 1
```

The `transact` method takes any amount of arguments (which are passed verbatim to the transacted function), and will return the return value of the transacted function.

```
>>> def petBunnies(bunnies, times):
...     totalPetsGiven = 0
...     for bunny in bunnies:
...         bunny.timesPetted += times
...         totalPetsGiven += times
...     return totalPetsGiven
>>> store.transact(petBunnies, [thumper, bugs], 2)
4
```

CHAPTER

SEVEN

POWERUPS

Powerups are a fairly unique feature to Axiom. They provide a way of persisting adapters to interfaces.

Powerups are built on top of Zope interfaces. If you don't know what they are, the short version is that they're objects that describe the *API* of other objects, without having any particular implementation.

7.1 Adaptation

Adaptation is a feature that's present in several libraries, such as Twisted. Let's say that you want an object satisfying the `IMailer` interface (an interface for objects that allow you to send e-mail), and you currently have a user object. Perhaps you could get an object that would let you send e-mail to a particular user by doing:

```
mailer = IMailer(user)
```

The basic idea that you can adapt an object to some interface, by calling the interface with that object, to get a (usually different) object that satisfies the interface you want.

Axiom provides adaptation in the form of powerups. Powerups allow you to easily persist objects with pretty much arbitrary behavior and allow you to add them to existing objects, without modifying them.

7.2 Adding powerups

For this example, we'll create transformers that can turn into cars. First, we'll define the interfaces: that is, we define what you can do with robot and car objects:

```
class IRobot(interface.Interface):
    def attack(target):
        """
        Attacks the target.
        """

class ICar(interface.Interface):
    def drive(location):
        """
        Drives to location.
        """
```

We make two sample implementations:

```
class Transformer(item.Item):
    name = attributes.text(allowNone=False)
    damage = attributes.integer(allowNone=False)

    def attack(self, target):
        print "{s} hits {t} for {s.damage} damage!".format(s=self, t=target)
```

```
class Truck(item.Item):
    wheels = attributes.integer(default=18)

    def drive(self, location):
        return defer.succeed(None)
```

When you create just any `Transformer`, it can't automagically turn into a car yet (that is, it's not adaptable to `ICar`). When you try to adapt it, an error is raised:

```
>>> store = Store()
>>> optimus = Transformer(store=store, name=u"Optimus Prime", damage=100)
>>> ICar(optimus)
Traceback (most recent call last):
...
TypeError: ('Could not adapt', ...)
```

The store is required because powerups are persisted.

If you can turn any arbitrary `IRobot` into any `ICar`, you want a regular adapter registry, such as the one offered by Twisted. In this case, we want to make it so that this particular transformer can turn into a car. Additionally, we want that behavior to be persisted. To do this, we'll be installing a powerup on the item:

```
>>> truck = Truck(store=store, wheels=8)
>>> optimus.powerUp(truck, ICar)
```

After that, we can adapt the transformer to be a car:

```
>>> assert ICar(optimus) is truck
```

7.3 In-memory powerups

Sometimes, particularly while experimenting or testing, it may be useful to use in-memory powerups instead of regular powerups. These are not persisted, and die whenever the object dies.

```
>>> inMemory = Store()
>>> rodimus = Transformer(store=inMemory, name=u"Rodimus Prime", damage=50)
>>> hotRod = HotRod(store=inMemory, color=u"Red and yellow")
>>> rodimus.inMemoryPowerUp(hotRod, ICar)
>>> assert ICar(rodimus) is hotRod
```

7.4 Removing powerups

Powerups can be removed by the `powerDown` method.

```
>>> optimus.powerDown(truck, ICar)
>>> ICar(optimus)
Traceback (most recent call last):
...
TypeError: ('Could not adapt', ...)
```

(Since it's a crying shame that Optimus can't turn into a truck, we'll restore his abilities.)

```
>>> optimus.powerUp(truck, ICar)
>>> assert ICar(optimus) is truck
```

CHAPTER

EIGHT

UPGRADING SCHEMATA

Often, you'll want to change the schema of an item. You might want to remove attributes, add attributes, or some combination of the two. Axiom comes with strong built-in support for migrating your data across schema upgrades.

If you're executing these exercises by hand, keep in mind that whenever we re-open a store, you should close the store before continuing. You can close the store by ending your interpreter session.

8.1 Schema versions and upgrader functions

A schema version is an integer describing the current revision of the schema. If you don't specify a schema version, your item's default schema version is `1`:

```
>>> class Lollipop(item.Item):
...     flavor = attributes.text(allowNone=False)
...     yummy = attributes.boolean(default=True)
>>> Lollipop.schemaVersion
1
```

An upgrader is a simple function that receives an old version of the item and is supposed to return a version of the item with a schema version that's one higher.

Note: Upgrader functions, like most migration logic, should never be removed from your code base.

8.1.1 Enabling upgrading behavior

The upgrading behavior is a service present in every Axiom store. To enable it, you should adapt the store to the `IService` interface to get a service object, which you can then start. If that doesn't make any sense to you yet, don't worry – it's pretty easy:

```
>>> from twisted.application.service import IService
>>> theStore = store.Store()
>>> IService(theStore).startService()
```

Of course, generally, you'll only do this on stores that are persisted – otherwise it doesn't make much sense to be upgrading in the first place.

8.2 A simple example: adding a field

A very common schema change would be to add a field. Let's say we have a series of things you can order at a coffee shop, each of which have a description and a price:

```
from axiom import attributes, item


class ShopItem(item.Item):
    typeName = "shop_item"

    description = attributes.text(allowNone=False)
    price = attributes.money(allowNone=False)
```

Let's create a few instances of these:

```
>>> s = store.Store()
>>> from coffeeshop import ShopItem
>>> from decimal import Decimal
>>> fiveBucks = Decimal("5.00")
>>> ShopItem(store=s, description=u"Coffee", price=fiveBucks)
ShopItem(description=u'Coffee', price=Decimal('5.00'), storeID=1)@...
>>> ShopItem(store=s, description=u"Iced Coffee", price=fiveBucks)
ShopItem(description=u'Iced Coffee', price=Decimal('5.00'), storeID=2)@...
>>> ShopItem(store=s, description=u"Muffin", price=fiveBucks)
ShopItem(description=u'Muffin', price=Decimal('5.00'), storeID=3)@...
```

For analytics purposes, we'd like to also store whether or not a thing is a drink or food.

TODO: newcoffeeshop.py

8.3 Writing upgrade logic to change a value

Suppose you have a database with temperature measurements. The American engineer who originally produced the measurements used degrees Fahrenheit.

```
from axiom import attributes, item

class Measurement(item.Item):
    typeName = "measurement"

    temperature = attributes.point4decimal()
    pressure = attributes.point4decimal()

>>> s = store.Store(storePath)
>>> from measurements import Measurement
>>> measurement = Measurement(store=s, temperature=-100, pressure=100)
>>> measurement.schemaVersion
1
```

You decide that it would be better to change the unit from degrees Fahrenheit to Kelvins. Technically, we don't really want to change the database *schema*: the attribute type, `point4decimal`, is entirely appropriate for temperatures in either unit. However, confusing units is potentially disastrous. [1]

By exacting a schema change, Axiom will take care of all the conversions for you, and you know that any item you use will have been converted, so there can be no confusion as to what unit a value is expressed in.

```
from axiom import attributes, item, upgrade
from decimal import Decimal

class Measurement(item.Item):
    typeName = "measurement"
    schemaVersion = 2

    temperature = attributes.point4decimal()
```

[1] Confusing different units for the same quantity was the core problem that caused the loss of the Mars Climate Orbiter.

```
    pressure = attributes.point4decimal()

def _upgradeMeasurementTemperature(old):
    new = old.upgradeVersion("measurement", 1, 2)
    new.temperature = ((old.temperature - 32) * 5 / 9) + Decimal("273.15")
    return new

upgrade.registerUpgrader(_upgradeMeasurementTemperature, "measurement", 1, 2)
```

At this point, we end the process and reload the store.

```
>>> IService(s).startService()
>>> measurement.schemaVersion
2
>>> s.query(Measurement).count()
1
>>> s.findUnique(Measurement).temperature
Decimal('199.81')
```

8.4 Attribute copying upgraders

One common pattern for upgraders is that they'll re-use most of the values the old item has.

TODO: write example

8.5 Deleting upgraders

Another common pattern for upgraders is to delete an item that was previously being stored.

TODO: write example

CHAPTER

NINE

INDICES AND TABLES

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