PyFactory Documentation

Release 0.4.0

Mitchell Hashimoto

Sep 27, 2017

Contents

CHAPTER 1

Overview

PyFactory is a library for writing and using model factories. Model factories allow you to replace test fixtures or manual model creation in tests with succint, easy to use factories.

The need for factories becomes apparent when you're testing any application of at least average complexity, where models often have a large tree of dependencies. For example, on some website you may want to test the `Comment` model. A `Comment` requires an author, which is a `User`, and a `Post`. A `Post` may further require a `Category`, and so on. So, just to test a simple `Comment`, you're typically forced to either create a brittle network of static fixtures, or manually create many models and glue together their relationships. *Yuck!*

With model factories, you would simply do the following:

```
comment = CommentFactory().create("comment")
```

This handles creating all the dependent models, as well. And the code for this factory is equally simple:

```
from pyfactory import Factory, association, schema
import models

class CommentFactory(Factory):
    _model = models.Comment
    _model_builder = models.ModelBuilder

    @schema()
    def comment(self):
        return {
            "body": "Some text...",
            "author_id": association(UserFactory(), "user", "id"),
            "post_id": association(PostFactory(), "post", "id")
        }

# Imagine the UserFactory and PostFactory here, which are basically
# equivalent to the above.
```

CHAPTER 2

Documentation

Tutorial

A *Tutorial* covering most of the features is available.

Usage Documentation

The following is a list of pages dedicated to specific concepts of **PyFactory**. It is recommended that you read the *Tutorial* before diving into these pages.

Model Builders

PyFactory is model-agnostic, meaning that it can be used to write factories for any kind of model or backing store. That means that **PyFactory** can be used to write factories for Django models, SQLAlchemy, your custom solution, etc.

The key abstraction which makes this possible is the *model builder*. A model builder is a class which implements only a few methods which tells PyFactory how to create models.

Model Builder Interface

The main model builder interface is documented below.

class `ModelBuilder`

`build`(*cls*, *model_cls*, *attributes*)
(classmethod)

This method is called when a model needs to be built. The model should *not* be persisted to the backing store, in this case. The return value of this method should be the instantiated model.

Parameters

- *model_cls* - This is the model class defined by the factory's `_model` class variable.
- *attributes* - This is a dictionary of attributes to initialize the model with.

create (*cls*, *model_cls*, *attributes*)
(classmethod)

This method is called when a model needs to be created. The model should be created and *saved*. The return value should be the new instance of the model.

Parameters

- *model_cls* - This is the model class defined by the factory's `_model` class variable.
- *attributes* - This is a dictionary of attributes to initialize the model with.

Example Model Builder

Below is an example model builder, which simply sets attributes on objects as the "model":

```
class MyModelBuilder(object):
    @classmethod
    def build(cls, model_cls, attrs):
        result = model_cls()
        for key,val in attrs.iteritems():
            setattr(result, key, val)

        return result

    @classmethod
    def create(cls, model_cls, attrs):
        result = self.build(model_cls, attrs)
        result.saved = True
        return result
```

Using Model Builders

To use or enable a model builder, specify it via the `_model_builder` class variable on the factory. For example, to use our model builder above:

```
class MyFactory(Factory):
    _model = object
    _model_builder = MyModelBuilder

    # ...
```

The above creates a new factory `MyFactory` which will use `MyModelBuilder` as the model builder, and `object` will be passed in as the `model_cls` to the model builder methods.

Factories

Factories are the bread and butter of **PyFactory**. Once you have a model builder for your models, you're ready to write factories. The key terms to understand are the following:

- **factory** - The class which can be used to instantiate models.

- **schema** - A type of model to instantiate from a single factory. Schemas define the structure of the model which is built.

Another way to think of it that a factory contains many schemas with which to build models.

Creating a Factory

Creating the Factory Class

To define a factory:

1. Subclass *`Factory`*
2. Define `_model` which is the type of the model to instantiate.
3. Define `_model_builder` to point to the *model builder* you created.
4. Define one or many schemas.

Here is an example factory:

```
from pyfactory import Factory

class MyFactory(Factory):
    _model = MyModel
    _model_builder = MyModelBuilder

    # Define schemas here. This will be explained later.
```

Schemas

Schemas define the structure of a created model. A factory class can contain many schemas and therefore know how to instantiate models with many different attributes.

A schema is an instance method which returns a dictionary of attributes, and the method must be decorated with *`@schema()`*. This dictionary is then used to instantiate the model.

An example schema, for a hypothetical `User` model:

```
@schema()
def basic(self):
    return {
        "first_name": "John",
        "last_name": "Doe"
    }
```

Note that `self` points to an instance of your *`Factory`*, so you can call any methods on it. The uses of this are shown later for schema inheritance.

Using a Factory

Once the factory class is defined, using it is simple, since it has a very simple API. Here is an example of using the factory we created above:

```
user = MyFactory().create("basic")
```

The basic steps are:

1. Instantiate your factory. This allows you to pass configuration, if needed, to the factory, as well as to hold instante state for the schemas.
2. Call the *factory API* to *realize* a schema. In the above example, we're creating a model with the `basic` schema.

There are three main ways to realize a schema:

- *attributes* - This will return the raw dictionary of the schema. This doesn't invoke your model builder at all.
- *build* - This will return an instance of your model, but will not persist it to any backing store.
- *create* - This will return an instance of your model which is persisted to the backing store after being created.

Note that depending on your model builder and type of models you're creating, `build` and `create` may not be different at all. The two differences are provided for convenience.

Overriding Schema Attributes

While schemas and provide a common skeleton and simple way to quickly build out records, it is very common that you want to override one or more fields of the record. You can do this very easily by passing additional keyword arguments to any of the schema realization methods. For example, for the user above, if we wanted to override the `first_name` field, we can do so easily:

```
user = UserFactory().create("basic", first_name="Bob")
print user.first_name # => "Bob"
print user.last_name  # => "Doe"
```

This can be done with any field and any number of them.

Schema Inheritance

It is often the case that some sort of "schema inheritance" is necessary. For example, a `User` might be the same in every way except for a `type` field noting whether they're an admin, user, guest, etc. In such a case, creating one shared schema and reusing it with subtle differences is the way to go:

```
@schema()
def base(self):
    return {
        "name": "John Doe"
    }

@schema()
def admin(self):
    base = self.schema("base")
    base["type"] = "admin"
    return base

@schema()
def user(self):
    base = self.schema("base")
    base["type"] = "user"
    return base
```

This way, all our shared attributes are in the `base` schema, and the other schemas modify it in a subtle way.

Note: The `schema` method should be used instead of `attributes`. `attributes` will resolve any special fields, and this usually is not the behavior you would like because you want overrides to take effect prior to any special field resolution. `schema` will return the raw schema dictionary.

Factory Inheritance

Although slightly more rare, it is sometimes useful to have factory inheritance, where one factory inherits from another. This is the same as any other class inheritance in Python. The only difference is when you want to create a schema of the same name, but also want to use the attributes of the parent schema of the same name. For example, instead of using schema inheritance above, we could've used factory inheritance. Example:

```
class MySubFactory(MyFactory):
    @schema()
    def base(self):
        base = super(MySubFactory, self).attributes("base")
        base["email"] = "myemail@domain.com"
        return base
```

Hopefully there is nothing surprising here. The only thing is that we can't simply super and call `base` since schemas are treated differently than normal instance methods. Instad, we use the fact that we're a factory to ask for the attributes of the parent's "base" schema, and use that.

Associations

Any models for non-trivial application are going to require associations of some sort. It doesn't matter if you're using a relational database, a key-value store, or a document store. There is always data that is associated in some way.

This is really the killer feature of PyFactory: The ability to create complex graphs of associations for a model on the fly. A real-world example which kicked off the building of this library: An `ApplicationAchievement` requires an `Achievement` and a `Session`, which in turn requires an `Application` which furthermore requires a `User`. So if we were writing unit tests for an `ApplicationAchievement` we could either create all these by hand, or mock them out. With **PyFactory**, you just write the factory for all the models, link them together using associations, and the rest is done for you!

Basic Association

Let's look at a basic association: a `Post` has a column which is a foreign key called `author_id` which points to a `User`. Assuming the `UserFactory` is already written, here is the `PostFactory`:

```
from pyfactory import Factory, schema, association

class PostFactory(Factory):
    @schema()
    def basic(self):
        return {
            "title": "My Post",
            "author_id": association(UserFactory(), "basic", "id")
        }
```

The key line is the `association` line. This tells `PyFactory` three important things:

- The association can be built from the `UserFactory`
- The schema to build is `basic`
- The attribute to use is `id`

The result of this is that the `author_id` field will end up being replaced with the "id" attribute for the "basic" schema from the `UserFactory`. Nifty!

The `attribute` parameter can also be ommitted, which will then simply use the entire model as the value of the field. This is particularly useful for document stores where you build up each part of the document piecemeal.

Note: Since schemas return the general structure of your application, rather than direct value, this association doesn't return the model instance right away. Instead, the association is not built until the schema is realized.

For more information on how this works, read the documentation on *special fields*.

Multiple Fields from a Single Association

Sometimes a model may use multiple fields from a single model. This is, for example, common with document stores where instead of using joins for relational data, it is often common to sacrafice some small data normalization for a document structure. As an example, a comment to a blog post may store the name and email of the author with the comment instead of a foreign key to a user document.

In this case, using two `association` calls wouldn't be appropriate since this would cause **PyFactory** to create two *different* records when you really want the value of two different attributes from a *single* record. An example of this exact case is shown below:

```
class CommentFactory(Factory):
    @schema()
    def comment(self):
        user = association(UserFactory(), "basic")

        return {
            "body": "This is my comment.",
            "name": user.attribute("name"),
            "email": user.attribute("email")
        }
```

This allows you to easily get multiple attributes of a single association.

Note: You can **note** assign the return value of an `attribute` call to a variable. The result is not the actual value of the attribute as you would expect. Since schema methods simply return the *structure* of a model, it has to encapsulate associations as such in your schema. To that end, a call to `attribute` actually returns a reference to a *`Field`* instance.

Building a Field from an Association's Data

Another common case is that an attribute of an associated model may not actually be used directly, but instead be used to build the value itself. For example, going back to our comment example above: What if the `User` object had a `first_name` and `last_name` field, but we wanted to store the full name in the comment, in a single field? With what we've seen so far, we would be out of luck.

Associations have another trick up their sleeve: callbacks. You can provide a callback to an association, which will be called with an attributes dictionary you can actually use to build up values.

```
class CommentFactory(Factory):
    @schema()
    def comment(self):
        def get_email(user):
            return "%s %s" % \
                (user["first_name"], user["last_name"])

        user = association(UserFactory(), "basic")

        return {
            "body": "This is my comment.",
            "name": user.callback(get_email)
        }
```

In this case, its hopefully clear to see that the `get_email` callback will be called, passing in a dictionary of attributes for the user. The return value of the callback will be used for the actual value of the field.

Special Fields

Documentation for special fields is coming soon.

In the mean time, you can check read the API reference for *`Field`* and I recommend viewing the source of *`AssociationField`* and *`SequenceField`*.

API Reference

A complete *API reference* is the recommend documentation for in-depth details on various pieces of **PyFactory**.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

API Reference

PyFactory contains one main top-level package, *pyfactory*.

pyfactory – Model Factory Library

PyFactory is a model factory library.

pyfactory.**__version__ = '0.4.0'**
The PyFactory version installed.

pyfactory.**Factory**
Alias for *pyfactory.factory.Factory*

pyfactory.**schema**
Alias for *pyfactory.schema.schema()*

pyfactory.**association**
Alias for *pyfactory.field.association()*

pyfactory.**sequence**
Alias for *pyfactory.field.sequence()*

Sub-modules:

factory – Factory Superclass

Contains the Factory base class and metaclass for all the factories created by PyFactory.

class pyfactory.factory.**Factory**
This is the base class for all created factories. All factories at some point lead back to this superclass.

_model
Subclasses of this class should define this variable to be the class to use as the model for all the schemas.

_model_builder
Subclasses of this class should define this variable to be the value to use as the model builder for all the schemas.

schema(*schema*, ***kwargs*)
This returns the raw schema result for the given name. This will not resolve any special fields.

Parameters

- *schema*: The name of the schema to retrieve.
- ***kwargs* (optional): Any additional keyword arguments given override the attributes returned. This allows for customization of the factory defaults.

attributes(*schema*, *_pyfactory_scope='attributes'*, ***kwargs*)
This returns the attributes for a particular schema. The attributes are returned as a dict instead of a model instance.

Parameters

- *schema*: The name of the schema to retrieve.
- ***kwargs* (optional): Any additional keyword arguments given override the attributes returned. This allows for customization of the factory defaults.

build(*schema*, ***kwargs*)
This builds a model but does not save it. The arguments are the same as *attributes()*.

create(*schema*, ***kwargs*)
This builds a model based on the schema with the given name and saves it, returning the new model. The arguments are the same as *attributes()*.

schema – Schema Decorator

This module contains the schema decorator. Users of PyFactory should instead import *schema()* directly from *pyfactory*.

pyfactory.schema.**schema**(*model=None*)
Decorator to mark a method in a Factory as a schema. This decorator has no effect on functions which aren't part of a subclass of Factory.

field – Contains Special Field Types

Contains the special field types for PyFactory.

pyfactory.field.**association**(**args*, ***kwargs*)
Shortcut method for enabling an association field. See documentation of *AssociationField*.

pyfactory.field.**sequence**(**args*, ***kwargs*)
Shortcut method for enabling a sequence field. See documentation of *SequenceField*.

class pyfactory.field.**Field**

If you wish to implement a custom field which has special behavior, you must inherit and implement the methods in this class. Fields are how things such as *association()* and *sequence()* are implemented.

resolve(*scope*)

This method is called by PyFactory to resolve the value of a special field. The scope argument will be one of attributes, build, or create depending on what method was called on the factory.

The return value of this method is what is put into the actual attributes dict for the model.

class pyfactory.field.**AssociationField**(*factory*, *schema*, *attr=None*)

This marks the field value as the result of creating another model from another factory.

If attr is specified, then that attribute value will be the value of the association. This requires that the model builder understand the getattr class method, otherwise an exception will be raised.

Parameters

- *factory*: An instance of a Factory to get the model from.
- *schema*: The name of the schema to load.
- ***attr* (optional): The name of the attribute to read from the** resulting model to place as the value of this field. If not given, the entire model becomes the value of the field.

attribute(*attr*)

Parameters

- *attr*: The name of the attribute to resolve to.

This will return a new *Field* instance which resolves to the value of the given attr of this association. This method is useful to re-use a single association multiple times for different values. For example, say you have a schema which uses the id and the name field of a user. You could then define the schema like so:

```
@schema()
def example(self):
    user = association(UserFactory(), "basic")

    return {
        "remote_id": user.attribute("id"),
        "remote_name": user.attribute("name")
    }
```

The benefit of the above, instead of using two separate associations, is that the association in this case will resolve to the exact same model, whereas two associations will always resolve to two different models.

callback(*callback*)

Parameters

- *callback*: Callback to be called, with the association as a parameter.

This will return a new *Field* instance which when resolved will call the given callback, passing the association as a parameter. To read an attribute from the association, use the dictionary syntax of association[key]. This will do the right thing depending on whether you're resolving attributes or a model build.

As an example, let's say you want to create a schema which depends on the concatenated first and last name of a user. Here is an example:

```
@schema()
def example(self):
    def get_name(association):
        return "%s %s" % \
            (association["first_name"], association["last_name"])

    user = association(UserFactory(), "basic")

    return {
        "name": user.callback(get_name)
    }
```

class `pyfactory.field.`**`SequenceField`**(*string*, *interpolation_variable='n'*, *unique_key='_default'*)

This causes the value of a field to be interpolated with a sequential value. The `interpolation_variable` will be the variable name used for interpolation in the string.

Parameters

- *string*: The string to perform the sequence interpolation on.
- *interpolation_variable* (optional): The name of the variable to use for interpolation in the `string`.
- *unique_key* (optional): The unique key to base the sequence creation on. By default, every time you call sequence, no matter what model factory or schema you're in, the sequence will increase. By specifying a unique `unique_key`, it isolates the increment to that key.

Tutorial

Welcome to PyFactory!

This is a whirlwind tour of the features of PyFactory and a quick guide on how to use each of them.

What is PyFactory?

PyFactory is a library for writing and using model factories. Model factories allow you to replace test fixtures or manual model creation in tests with succint, easy to use factories.

PyFactory is also model-agnostic, meaning it can generate any sort of models for any ORM or backing store.

The specific feature list of **PyFactory**:

- Model-agnostic. Generate models for any ORM or backing store.
- Generate attributes. Instead of getting a full-blown model, you can request just the attributes of the model as a dictionary.
- Build model objects (and optionally ask for them saved to the backing store as well).
- Model complex associations in your schemas to generate all dependencies along with the model.

Your First Factory

Let's create your first factory. Before doing that, however, we need to define what our models are. In your case, this may be a model for Django, SQLAlchemy, or something custom. For the purposes of our examples here, we're going to simply set attributes on an object as our model.

The Model Builder

First, we need to create a *model builder*. This is the class which knows how to build our models given a dictionary of attributse. This is the core piece of **PyFactory** which enables model-agnosticism. Model builders are extremely easy to write, especially in our case:

```
class MyModelBuilder(object):
    @classmethod
    def build(cls, model_cls, attrs):
        result = model_cls()
        for key,val in attrs.iteritems():
            setattr(result, key, val)

        return result

    @classmethod
    def create(cls, model_cls, attrs):
        result = self.build(model_cls, attrs)
        result.saved = True
        return result
```

As you can see, given a model class and attributes, the model builder knows how to build them. In our case, this is simply using `setattr` on an object.

In-depth documentation is available on *model builders*.

The Factory

Now that we have a model builder, let's go forward and build a simple factory. Let's pretend we have a `User` model we want to create a factory for:

```
from pyfactory import Factory, schema

class UserFactory(Factory):
    _model = object
    _model_builder = MyModelBuilder

    @schema()
    def basic(self):
        return {
            "first_name": "John",
            "last_name": "Doe"
        }
```

Given the above factory, we can now generate users!

```
user = UserFactory().create("basic")
print user.first_name, user.last_name # "John Doe"
```

Using Your Factory

In the example above we used the `create` method to use our factory. There are other methods available as well:

- `attributes` - This method will return the attributes of the model it would create as a `dict`.
- `build` - This method will build the model but will not save it to the backing store. This is useful if you want a model to manipulate more before saving it.

- `create` - This method will build the model and save it to the backing store.

Using these methods is straightforward:

```
factory = UserFactory()
factory.attributes("basic")
factory.build("basic")
factory.create("basic")
```

Associations

Most models, especially those in relational databases, have associations with other models. In the past, if you had a model which had many associations, you would have to manually create all these associations and maintain the brittle relationships. With **PyFactory**, these associations are managed for you.

Let's create another factory, for a hypothetical `Post` which has an author, which is a `User` that we created a factory for moments before.

```
from pyfactory import Factory, association, schema

class PostFactory(Factory):
    _model = object
    _model_builder = MyModelBuilder

    @schema()
    def basic(self):
        return {
            "title": "My Post",
            "author": association(UserFactory(), "basic")
        }
```

The new piece of the above factory, of course, is the `association` function call. This means that the `author` field is an association to a `UserFactory()` factory and the `basic` schema of that factory, which is the schema we created earlier.

Given the above example, we can now create a `Post` which already has a `User` dependency created for us!

```
post = PostFactory().create("basic")
print post.title # => "My Post"
print post.author.first_name, post.author.last_name # => "John Doe"
```

Sequences

Sometimes, in order to help generate information that looks different from other information, it is useful to incorporate sequences of data. For example, instead of having ever user's first name to "User" it would be nice if it could be "User #1," "User #2," etc. Sequences make this easy.

```
from pyfactory import Factory, schema, sequence

class UserFactory(Factory):
    _model = object
    _model_builder = MyModelBuilder

    @schema()
    def basic(self):
        return {
```

```
            "first_name": sequence("User %(n)d")
        }
```

Given the above example, we can now create `User` objects which have sequential first names:

```
user1 = UserFactory().create("basic")
user2 = UserFactory().create("basic")

print user1.first_name # => "User 1"
print user2.first_name # => "User 2"
```

Python Module Index

p

Index

Symbols

A

B

C

F

M

P

R

S