Factory Boy Documentation

Release 2.6.0

Raphaël Barrois, Mark Sandstrom

Mar 08, 2017

Contents

factory_boy is a fixtures replacement based on thoughtbot's factory_girl.

As a fixtures replacement tool, it aims to replace static, hard to maintain fixtures with easy-to-use factories for complex object.

Instead of building an exhaustive test setup with every possible combination of corner cases, `factory_boy` allows you to use objects customized for the current test, while only declaring the test-specific fields:

```
class FooTests(unittest.TestCase):

    def test_with_factory_boy(self):
        # We need a 200€, paid order, shipping to australia, for a VIP customer
        order = OrderFactory(
            amount=200,
            status='PAID',
            customer__is_vip=True,
            address__country='AU',
        )
        # Run the tests here

    def test_without_factory_boy(self):
        address = Address(
            street="42 fubar street",
            zipcode="42Z42",
            city="Sydney",
            country="AU",
        )
        customer = Customer(
            first_name="John",
            last_name="Doe",
            phone="+1234",
            email="john.doe@example.org",
            active=True,
            is_vip=True,
            address=address,
        )
        # etc.
```

factory_boy is designed to work well with various ORMs (Django, Mogo, SQLAlchemy), and can easily be extended for other libraries.

Its main features include:

- Straightforward declarative syntax
- Chaining factory calls while retaining the global context
- Support for multiple build strategies (saved/unsaved instances, stubbed objects)
- Multiple factories per class support, including inheritance

CHAPTER 1

Links

- Documentation: http://factoryboy.readthedocs.org/
- Repository: https://github.com/rbarrois/factory_boy
- Package: https://pypi.python.org/pypi/factory_boy/

factory_boy supports Python 2.6, 2.7, 3.2 and 3.3, as well as PyPy; it requires only the standard Python library.

CHAPTER 2

Download

PyPI: https://pypi.python.org/pypi/factory_boy/

```
$ pip install factory_boy
```

Source: https://github.com/rbarrois/factory_boy/

```
$ git clone git://github.com/rbarrois/factory_boy/
$ python setup.py install
```

CHAPTER 3

Usage

Note: This section provides a quick summary of factory_boy features. A more detailed listing is available in the full documentation.

Defining factories

Factories declare a set of attributes used to instantiate an object. The class of the object must be defined in the `model` field of a `class Meta:` attribute:

```python
import factory
from . import models

class UserFactory(factory.Factory):
    class Meta:
        model = models.User

    first_name = 'John'
    last_name = 'Doe'
    admin = False

# Another, different, factory for the same object
class AdminFactory(factory.Factory):
    class Meta:
        model = models.User

    first_name = 'Admin'
    last_name = 'User'
    admin = True
```

Using factories

factory_boy supports several different build strategies: build, create, and stub:

```
# Returns a User instance that's not saved
user = UserFactory.build()

# Returns a saved User instance
user = UserFactory.create()

# Returns a stub object (just a bunch of attributes)
obj = UserFactory.stub()
```

You can use the Factory class as a shortcut for the default build strategy:

```
# Same as UserFactory.create()
user = UserFactory()
```

No matter which strategy is used, it's possible to override the defined attributes by passing keyword arguments:

```
# Build a User instance and override first_name
>>> user = UserFactory.build(first_name='Joe')
>>> user.first_name
"Joe"
```

It is also possible to create a bunch of objects in a single call:

```
>>> users = UserFactory.build_batch(10, first_name="Joe")
>>> len(users)
10
>>> [user.first_name for user in users]
["Joe", "Joe", "Joe", "Joe", "Joe", "Joe", "Joe", "Joe", "Joe", "Joe"]
```

Realistic, random values

Tests look better with random yet realistic values. For this, factory_boy relies on the excellent fake-factory library:

```
class RandomUserFactory(factory.Factory):
    class Meta:
        model = models.User

    first_name = factory.Faker('first_name')
    last_name = factory.Faker('last_name')
```

```
>>> UserFactory()
<User: Lucy Murray>
```

Lazy Attributes

Most factory attributes can be added using static values that are evaluated when the factory is defined, but some attributes (such as fields whose value is computed from other elements) will need values assigned each time an instance is generated.

These “lazy” attributes can be added as follows:

```
class UserFactory(factory.Factory):
    class Meta:
        model = models.User

    first_name = 'Joe'
    last_name = 'Blow'
    email = factory.LazyAttribute(lambda a: '{0}.{1}@example.com'.format(a.first_name,
↪ a.last_name).lower())
```

```
>>> UserFactory().email
"joe.blow@example.com"
```

Sequences

Unique values in a specific format (for example, e-mail addresses) can be generated using sequences. Sequences are defined by using `Sequence` or the decorator `sequence`:

```
class UserFactory(factory.Factory):
    class Meta:
        model = models.User

    email = factory.Sequence(lambda n: 'person{0}@example.com'.format(n))

>>> UserFactory().email
'person0@example.com'
>>> UserFactory().email
'person1@example.com'
```

Associations

Some objects have a complex field, that should itself be defined from a dedicated factories. This is handled by the `SubFactory` helper:

```
class PostFactory(factory.Factory):
    class Meta:
        model = models.Post

    author = factory.SubFactory(UserFactory)
```

The associated object’s strategy will be used:

```
# Builds and saves a User and a Post
>>> post = PostFactory()
>>> post.id is None  # Post has been 'saved'
False
>>> post.author.id is None  # post.author has been saved
False

# Builds but does not save a User, and then builds but does not save a Post
>>> post = PostFactory.build()
>>> post.id is None
```

```
True
>>> post.author.id is None
True
```

Debugging factory_boy

Debugging factory_boy can be rather complex due to the long chains of calls. Detailed logging is available through the `factory` logger.

A helper, *factory.debug()*, is available to ease debugging:

```
with factory.debug():
    obj = TestModel2Factory()


import logging
logger = logging.getLogger('factory')
logger.addHandler(logging.StreamHandler())
logger.setLevel(logging.DEBUG)
```

This will yield messages similar to those (artificial indentation):

```
BaseFactory: Preparing tests.test_using.TestModel2Factory(extra={})
  LazyStub: Computing values for tests.test_using.TestModel2Factory(two=
↪<OrderedDeclarationWrapper for <factory.declarations.SubFactory object at 0x1e15610>
↪>)
    SubFactory: Instantiating tests.test_using.TestModelFactory(__containers=(
↪<LazyStub for tests.test_using.TestModel2Factory>,), one=4), create=True
    BaseFactory: Preparing tests.test_using.TestModelFactory(extra={'__containers': (
↪<LazyStub for tests.test_using.TestModel2Factory>,), 'one': 4})
      LazyStub: Computing values for tests.test_using.TestModelFactory(one=4)
      LazyStub: Computed values, got tests.test_using.TestModelFactory(one=4)
    BaseFactory: Generating tests.test_using.TestModelFactory(one=4)
  LazyStub: Computed values, got tests.test_using.TestModel2Factory(two=<tests.test_
↪using.TestModel object at 0x1e15410>)
BaseFactory: Generating tests.test_using.TestModel2Factory(two=<tests.test_using.
↪TestModel object at 0x1e15410>)
```

ORM Support

factory_boy has specific support for a few ORMs, through specific `factory.Factory` subclasses:

- Django, with `factory.django.DjangoModelFactory`
- Mogo, with `factory.mogo.MogoFactory`
- MongoEngine, with `factory.mongoengine.MongoEngineFactory`
- SQLAlchemy, with `factory.alchemy.SQLAlchemyModelFactory`

CHAPTER 4

Contributing

factory_boy is distributed under the MIT License.

Issues should be opened through GitHub Issues; whenever possible, a pull request should be included.

All pull request should pass the test suite, which can be launched simply with:

```
$ make test
```

In order to test coverage, please use:

```
$ make coverage
```

To test with a specific framework version, you may use:

```
$ make DJANGO=1.7 test
```

Valid options are:

- `DJANGO` for `Django`
- `MONGOENGINE` for `mongoengine`
- `ALCHEMY` for `SQLAlchemy`

CHAPTER 5

Contents, indices and tables

Introduction

The purpose of factory_boy is to provide a default way of getting a new instance, while still being able to override some fields on a per-call basis.

Note: This section will drive you through an overview of factory_boy's feature. New users are advised to spend a few minutes browsing through this list of useful helpers.

Users looking for quick helpers may take a look at *Common recipes*, while those needing detailed documentation will be interested in the *Reference* section.

Basic usage

Factories declare a set of attributes used to instantiate an object, whose class is defined in the `class Meta`'s `model` attribute:

- Subclass `factory.Factory` (or a more suitable subclass)
- Add a `class Meta:` block
- Set its `model` attribute to the target class
- Add defaults for keyword args to pass to the associated class' `__init__` method

```
import factory
from . import base

class UserFactory(factory.Factory):
    class Meta:
        model = base.User
    firstname = "John"
    lastname = "Doe"
```

You may now get `base.User` instances trivially:

```
>>> john = UserFactory()
<User: John Doe>
```

It is also possible to override the defined attributes by passing keyword arguments to the factory:

```
>>> jack = UserFactory(firstname="Jack")
<User: Jack Doe>
```

A given class may be associated to many *Factory* subclasses:

```
class EnglishUserFactory(factory.Factory):
    class Meta:
        model = base.User

    firstname = "John"
    lastname = "Doe"
    lang = 'en'


class FrenchUserFactory(factory.Factory):
    class Meta:
        model = base.User

    firstname = "Jean"
    lastname = "Dupont"
    lang = 'fr'
```

```
>>> EnglishUserFactory()
<User: John Doe (en)>
>>> FrenchUserFactory()
<User: Jean Dupont (fr)>
```

Sequences

When a field has a unique key, each object generated by the factory should have a different value for that field. This is achieved with the *Sequence* declaration:

```
class UserFactory(factory.Factory):
    class Meta:
        model = models.User

    username = factory.Sequence(lambda n: 'user%d' % n)
```

```
>>> UserFactory()
<User: user1>
>>> UserFactory()
<User: user2>
```

Note: For more complex situations, you may also use the `@sequence()` decorator (note that `self` is not added as first parameter):

```
class UserFactory(factory.Factory):
    class Meta:
        model = models.User

    @factory.sequence
    def username(n):
        return 'user%d' % n
```

LazyAttribute

Some fields may be deduced from others, for instance the email based on the username. The *LazyAttribute* handles such cases: it should receive a function taking the object being built and returning the value for the field:

```
class UserFactory(factory.Factory):
    class Meta:
        model = models.User

    username = factory.Sequence(lambda n: 'user%d' % n)
    email = factory.LazyAttribute(lambda obj: '%s@example.com' % obj.username)
```

```
>>> UserFactory()
<User: user1 (user1@example.com)>

>>> # The LazyAttribute handles overridden fields
>>> UserFactory(username='john')
<User: john (john@example.com)>

>>> # They can be directly overridden as well
>>> UserFactory(email='doe@example.com')
<User: user3 (doe@example.com)>
```

Note: As for *Sequence*, a `@lazy_attribute()` decorator is available:

```
class UserFactory(factory.Factory):
    class Meta:
        model = models.User

    username = factory.Sequence(lambda n: 'user%d' % n)

    @factory.lazy_attribute
    def email(self):
        return '%s@example.com' % self.username
```

Inheritance

Once a "base" factory has been defined for a given class, alternate versions can be easily defined through subclassing.

The subclassed *Factory* will inherit all declarations from its parent, and update them with its own declarations:

```
class UserFactory(factory.Factory):
    class Meta:
        model = base.User

    firstname = "John"
    lastname = "Doe"
    group = 'users'

class AdminFactory(UserFactory):
    admin = True
    group = 'admins'
```

```
>>> user = UserFactory()
>>> user
<User: John Doe>
>>> user.group
'users'

>>> admin = AdminFactory()
>>> admin
<User: John Doe (admin)>
>>> admin.group  # The AdminFactory field has overridden the base field
'admins'
```

Any argument of all factories in the chain can easily be overridden:

```
>>> super_admin = AdminFactory(group='superadmins', lastname="Lennon")
>>> super_admin
<User: John Lennon (admin)>
>>> super_admin.group  # Overridden at call time
'superadmins'
```

Non-kwarg arguments

Some classes take a few, non-kwarg arguments first.

This is handled by the *inline_args* attribute:

```
class MyFactory(factory.Factory):
    class Meta:
        model = MyClass
        inline_args = ('x', 'y')

    x = 1
    y = 2
    z = 3
```

```
>>> MyFactory(y=4)
<MyClass(1, 4, z=3)>
```

Strategies

All factories support two built-in strategies:

- `build` provides a local object
- `create` instantiates a local object, and saves it to the database.

Note: For 1.X versions, the `create` will actually call `AssociatedClass.objects.create`, as for a Django model.

Starting from 2.0, *`factory.Factory.create()`* simply calls `AssociatedClass(**kwargs)`. You should use *`DjangoModelFactory`* for Django models.

When a *`Factory`* includes related fields (*`SubFactory`*, *`RelatedFactory`*), the parent's strategy will be pushed onto related factories.

Calling a *`Factory`* subclass will provide an object through the default strategy:

```
class MyFactory(factory.Factory):
    class Meta:
        model = MyClass
```

```
>>> MyFactory.create()
<MyFactory: X (saved)>

>>> MyFactory.build()
<MyFactory: X (unsaved)>

>>> MyFactory()  # equivalent to MyFactory.create()
<MyClass: X (saved)>
```

The default strategy can be changed by setting the `class Meta` *`strategy`* attribute.

Reference

This section offers an in-depth description of factory_boy features.

For internals and customization points, please refer to the *Internals* section.

The `Factory` class

class `factory.`**`FactoryOptions`**

New in version 2.4.0.

A *`Factory`*'s behaviour can be tuned through a few settings.

For convenience, they are declared in a single `class Meta` attribute:

```
class MyFactory(factory.Factory):
    class Meta:
        model = MyObject
        abstract = False
```

`model`

This optional attribute describes the class of objects to generate.

If unset, it will be inherited from parent *`Factory`* subclasses.

New in version 2.4.0.

abstract

This attribute indicates that the *Factory* subclass should not be used to generate objects, but instead provides some extra defaults.

It will be automatically set to `True` if neither the *Factory* subclass nor its parents define the *model* attribute.

> **Warning:** This flag is reset to `False` when a *Factory* subclasses another one if a *model* is set.

New in version 2.4.0.

inline_args

Some factories require non-keyword arguments to their `__init__()`. They should be listed, in order, in the *inline_args* attribute:

```
class UserFactory(factory.Factory):
    class Meta:
        model = User
        inline_args = ('login', 'email')

    login = 'john'
    email = factory.LazyAttribute(lambda o: '%s@example.com' % o.login)
    firstname = "John"
```

```
>>> UserFactory()
<User: john>
>>> User('john', 'john@example.com', firstname="John")  # actual call
```

New in version 2.4.0.

exclude

While writing a *Factory* for some object, it may be useful to have general fields helping defining others, but that should not be passed to the model class; for instance, a field named 'now' that would hold a reference time used by other objects.

Factory fields whose name are listed in *exclude* will be removed from the set of args/kwargs passed to the underlying class; they can be any valid factory_boy declaration:

```
class OrderFactory(factory.Factory):
    class Meta:
        model = Order
        exclude = ('now',)

    now = factory.LazyAttribute(lambda o: datetime.datetime.utcnow())
    started_at = factory.LazyAttribute(lambda o: o.now - datetime.
↪timedelta(hours=1))
    paid_at = factory.LazyAttribute(lambda o: o.now - datetime.
↪timedelta(minutes=50))
```

```
>>> OrderFactory()    # The value of 'now' isn't passed to Order()
<Order: started 2013-04-01 12:00:00, paid 2013-04-01 12:10:00>

>>> # An alternate value may be passed for 'now'
>>> OrderFactory(now=datetime.datetime(2013, 4, 1, 10))
<Order: started 2013-04-01 09:00:00, paid 2013-04-01 09:10:00>
```

New in version 2.4.0.

rename

Sometimes, a model expect a field with a name already used by one of *Factory*'s methods.

In this case, the *rename* attributes allows to define renaming rules: the keys of the *rename* dict are those used in the *Factory* declarations, and their values the new name:

```
class ImageFactory(factory.Factory):
    # The model expects "attributes"
    form_attributes = ['thumbnail', 'black-and-white']

    class Meta:
        model = Image
        rename = {'form_attributes': 'attributes'}
```

strategy

Use this attribute to change the strategy used by a *Factory*. The default is *CREATE_STRATEGY*.

class factory.**Factory**

Class-level attributes:

_meta

New in version 2.4.0.

The *FactoryOptions* instance attached to a *Factory* class is available as a *_meta* attribute.

_options_class

New in version 2.4.0.

If a *Factory* subclass needs to define additional, extra options, it has to provide a custom *FactoryOptions* subclass.

A pointer to that custom class should be provided as *_options_class* so that the *Factory*-building metaclass can use it instead.

Base functions:

The *Factory* class provides a few methods for getting objects; the usual way being to simply call the class:

```
>>> UserFactory()                  # Calls UserFactory.create()
>>> UserFactory(login='john')      # Calls UserFactory.create(login='john')
```

Under the hood, factory_boy will define the *Factory* __new__() method to call the default *strategy* of the *Factory*.

A specific strategy for getting instance can be selected by calling the adequate method:

classmethod build(*cls*, ***kwargs*)

Provides a new object, using the 'build' strategy.

classmethod build_batch(*cls*, *size*, ***kwargs*)

Provides a list of `size` instances from the *Factory*, through the 'build' strategy.

classmethod create(*cls*, ***kwargs*)

Provides a new object, using the 'create' strategy.

classmethod create_batch(*cls*, *size*, ***kwargs*)

Provides a list of `size` instances from the *Factory*, through the 'create' strategy.

classmethod stub(*cls*, ***kwargs*)

Provides a new stub

classmethod stub_batch(*cls*, *size*, ***kwargs*)

Provides a list of `size` stubs from the *Factory*.

classmethod generate(*cls*, *strategy*, ***kwargs*)
Provide a new instance, with the provided `strategy`.

classmethod generate_batch(*cls*, *strategy*, *size*, ***kwargs*)
Provides a list of `size` instances using the specified strategy.

classmethod simple_generate(*cls*, *create*, ***kwargs*)
Provide a new instance, either built (`create=False`) or created (`create=True`).

classmethod simple_generate_batch(*cls*, *create*, *size*, ***kwargs*)
Provides a list of `size` instances, either built or created according to `create`.

Extension points:

A `Factory` subclass may override a couple of class methods to adapt its behaviour:

classmethod _adjust_kwargs(*cls*, ***kwargs*)
The `_adjust_kwargs()` extension point allows for late fields tuning.

It is called once keyword arguments have been resolved and post-generation items removed, but before the `inline_args` extraction phase.

```
class UserFactory(factory.Factory):

    @classmethod
    def _adjust_kwargs(cls, **kwargs):
        # Ensure ``lastname`` is upper-case.
        kwargs['lastname'] = kwargs['lastname'].upper()
        return kwargs
```

classmethod _setup_next_sequence(*cls*)
This method will compute the first value to use for the sequence counter of this factory.

It is called when the first instance of the factory (or one of its subclasses) is created.

Subclasses may fetch the next free ID from the database, for instance.

classmethod _build(*cls*, *model_class*, **args*, ***kwargs*)
This class method is called whenever a new instance needs to be built. It receives the model class (provided to `model`), and the positional and keyword arguments to use for the class once all has been computed.

Subclasses may override this for custom APIs.

classmethod _create(*cls*, *model_class*, **args*, ***kwargs*)
The `_create()` method is called whenever an instance needs to be created. It receives the same arguments as `_build()`.

Subclasses may override this for specific persistence backends:

```
class BaseBackendFactory(factory.Factory):
    class Meta:
        abstract = True  # Optional

    def _create(cls, model_class, *args, **kwargs):
        obj = model_class(*args, **kwargs)
        obj.save()
        return obj
```

classmethod _after_postgeneration(*cls*, *obj*, *create*, *results=None*)

Parameters

- **obj** (`object`) – The object just generated

- **create** (*bool*) – Whether the object was 'built' or 'created'
- **results** (*dict*) – Map of post-generation declaration name to call result

The `_after_postgeneration()` is called once post-generation declarations have been handled.

Its arguments allow to handle specifically some post-generation return values, for instance.

Advanced functions:

classmethod reset_sequence(*cls*, *value=None*, *force=False*)

Parameters

- **value** (*int*) – The value to reset the sequence to
- **force** (*bool*) – Whether to force-reset the sequence

Allows to reset the sequence counter for a `Factory`. The new value can be passed in as the `value` argument:

```
>>> SomeFactory.reset_sequence(4)
>>> SomeFactory._next_sequence
4
```

Since subclasses of a non-`abstract` `Factory` share the same sequence counter, special care needs to be taken when resetting the counter of such a subclass.

By default, `reset_sequence()` will raise a `ValueError` when called on a subclassed `Factory` subclass. This can be avoided by passing in the `force=True` flag:

```
>>> InheritedFactory.reset_sequence()
Traceback (most recent call last):
  File "factory_boy/tests/test_base.py", line 179, in test_reset_sequence_
↪subclass_parent
    SubTestObjectFactory.reset_sequence()
  File "factory_boy/factory/base.py", line 250, in reset_sequence
    "Cannot reset the sequence of a factory subclass. "
ValueError: Cannot reset the sequence of a factory subclass. Please call␣
↪reset_sequence() on the root factory, or call reset_sequence(forward=True).

>>> InheritedFactory.reset_sequence(force=True)
>>>
```

This is equivalent to calling `reset_sequence()` on the base factory in the chain.

Strategies

factory_boy supports two main strategies for generating instances, plus stubs.

`factory.`**BUILD_STRATEGY**

The 'build' strategy is used when an instance should be created, but not persisted to any datastore.

It is usually a simple call to the `__init__()` method of the `model` class.

`factory.`**CREATE_STRATEGY**

The 'create' strategy builds and saves an instance into its appropriate datastore.

This is the default strategy of factory_boy; it would typically instantiate an object, then save it:

```
>>> obj = self._associated_class(*args, **kwargs)
>>> obj.save()
>>> return obj
```

Warning: For backward compatibility reasons, the default behaviour of factory_boy is to call `MyClass.objects.create(*args, **kwargs)` when using the `create` strategy.

That policy will be used if the *associated class* has an `objects` attribute *and* the *_create()* classmethod of the *Factory* wasn't overridden.

`factory.`**`use_strategy`**(*strategy*)

Decorator

Change the default strategy of the decorated *Factory* to the chosen `strategy`:

```
@use_strategy(factory.BUILD_STRATEGY)
class UserBuildingFactory(UserFactory):
    pass
```

`factory.`**`STUB_STRATEGY`**

The 'stub' strategy is an exception in the factory_boy world: it doesn't return an instance of the *model* class, and actually doesn't require one to be present.

Instead, it returns an instance of *StubObject* whose attributes have been set according to the declarations.

class `factory.`**`StubObject`**(*object*)

A plain, stupid object. No method, no helpers, simply a bunch of attributes.

It is typically instantiated, then has its attributes set:

```
>>> obj = StubObject()
>>> obj.x = 1
>>> obj.y = 2
```

class `factory.`**`StubFactory`**(*Factory*)

An *abstract Factory*, with a default strategy set to *STUB_STRATEGY*.

`factory.`**`debug`**(*logger='factory'*, *stream=None*)

Parameters

- **`logger`** (*str*) – The name of the logger to enable debug for
- **`stream`** (*file*) – The stream to send debug output to, defaults to `sys.stderr`

Context manager to help debugging factory_boy behavior. It will temporarily put the target logger (e.g `'factory'`) in debug mode, sending all output to :obj'~sys.stderr'; upon leaving the context, the logging levels are reset.

A typical use case is to understand what happens during a single factory call:

```
with factory.debug():
    obj = TestModel2Factory()
```

This will yield messages similar to those (artificial indentation):

```
BaseFactory: Preparing tests.test_using.TestModel2Factory(extra={})
  LazyStub: Computing values for tests.test_using.TestModel2Factory(two=
↪<OrderedDeclarationWrapper for <factory.declarations.SubFactory object at␣
↪0x1e15610>>)
```

```
    SubFactory: Instantiating tests.test_using.TestModelFactory(__containers=(
↪<LazyStub for tests.test_using.TestModel2Factory>,), one=4), create=True
    BaseFactory: Preparing tests.test_using.TestModelFactory(extra={'__containers
↪': (<LazyStub for tests.test_using.TestModel2Factory>,), 'one': 4})
      LazyStub: Computing values for tests.test_using.TestModelFactory(one=4)
      LazyStub: Computed values, got tests.test_using.TestModelFactory(one=4)
    BaseFactory: Generating tests.test_using.TestModelFactory(one=4)
  LazyStub: Computed values, got tests.test_using.TestModel2Factory(two=<tests.
↪test_using.TestModel object at 0x1e15410>)
BaseFactory: Generating tests.test_using.TestModel2Factory(two=<tests.test_using.
↪TestModel object at 0x1e15410>)
```

Declarations

Faker

class `factory.`**`Faker`**(*provider*, *locale=None*, ***kwargs*)

In order to easily define realistic-looking factories, use the *`Faker`* attribute declaration.

This is a wrapper around fake-factory; its argument is the name of a `fake-factory` provider:

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    name = factory.Faker('name')
```

```
>>> user = UserFactory()
>>> user.name
'Lucy Cechtelar'
```

`locale`

If a custom locale is required for one specific field, use the `locale` parameter:

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    name = factory.Faker('name', locale='fr_FR')
```

```
>>> user = UserFactory()
>>> user.name
'Jean Valjean'
```

classmethod **`override_default_locale`**(*cls*, *locale*)

If the locale needs to be overridden for a whole test, use *`override_default_locale()`*:

```
>>> with factory.Faker.override_default_locale('de_DE'):
...     UserFactory()
<User: Johannes Brahms>
```

classmethod **`add_provider`**(*cls*, *locale=None*)

Some projects may need to fake fields beyond those provided by `fake-factory`; in such cases, use *`factory.Faker.add_provider()`* to declare additional providers for those fields:

```
factory.Faker.add_provider(SmileyProvider)

class FaceFactory(factory.Factory):
    class Meta:
        model = Face

    smiley = factory.Faker('smiley')
```

LazyAttribute

class factory.**LazyAttribute**(*method_to_call*)

The *LazyAttribute* is a simple yet extremely powerful building brick for extending a *Factory*.

It takes as argument a method to call (usually a lambda); that method should accept the object being built as sole argument, and return a value.

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    username = 'john'
    email = factory.LazyAttribute(lambda o: '%s@example.com' % o.username)
```

```
>>> u = UserFactory()
>>> u.email
'john@example.com'

>>> u = UserFactory(username='leo')
>>> u.email
'leo@example.com'
```

The object passed to *LazyAttribute* is not an instance of the target class, but instead a `LazyStub`: a temporary container that computes the value of all declared fields.

Decorator

factory.**lazy_attribute**()

If a simple lambda isn't enough, you may use the *lazy_attribute()* decorator instead.

This decorates an instance method that should take a single argument, `self`; the name of the method will be used as the name of the attribute to fill with the return value of the method:

```
class UserFactory(factory.Factory)
    class Meta:
        model = User

    name = u"Jean"

    @factory.lazy_attribute
    def email(self):
        # Convert to plain ascii text
        clean_name = (unicodedata.normalize('NFKD', self.name)
                        .encode('ascii', 'ignore')
```

```
                                .decode('utf8'))
        return u'%s@example.com' % clean_name
```

```
>>> joel = UserFactory(name=u"Joël")
>>> joel.email
u'joel@example.com'
```

Sequence

class factory.**Sequence**(*lambda*, *type=int*)

If a field should be unique, and thus different for all built instances, use a *Sequence*.

This declaration takes a single argument, a function accepting a single parameter - the current sequence counter - and returning the related value.

Note: An extra kwarg argument, `type`, may be provided. This feature is deprecated in 1.3.0 and will be removed in 2.0.0.

```
class UserFactory(factory.Factory)
    class Meta:
        model = User

    phone = factory.Sequence(lambda n: '123-555-%04d' % n)
```

```
>>> UserFactory().phone
'123-555-0001'
>>> UserFactory().phone
'123-555-0002'
```

Decorator

factory.**sequence**()

As with *lazy_attribute()*, a decorator is available for complex situations.

sequence() decorates an instance method, whose `self` method will actually be the sequence counter - this might be confusing:

```
class UserFactory(factory.Factory)
    class Meta:
        model = User

    @factory.sequence
    def phone(n):
        a = n // 10000
        b = n % 10000
        return '%03d-555-%04d' % (a, b)
```

```
>>> UserFactory().phone
'000-555-9999'
>>> UserFactory().phone
'001-555-0000'
```

Sharing

The sequence counter is shared across all *Sequence* attributes of the *Factory*:

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    phone = factory.Sequence(lambda n: '%04d' % n)
    office = factory.Sequence(lambda n: 'A23-B%03d' % n)
```

```
>>> u = UserFactory()
>>> u.phone, u.office
'0041', 'A23-B041'
>>> u2 = UserFactory()
>>> u2.phone, u2.office
'0042', 'A23-B042'
```

Inheritance

When a *Factory* inherits from another *Factory*, their sequence counter is shared:

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    phone = factory.Sequence(lambda n: '123-555-%04d' % n)


class EmployeeFactory(UserFactory):
    office_phone = factory.Sequence(lambda n: '%04d' % n)
```

```
>>> u = UserFactory()
>>> u.phone
'123-555-0001'

>>> e = EmployeeFactory()
>>> e.phone, e.office_phone
'123-555-0002', '0002'

>>> u2 = UserFactory()
>>> u2.phone
'123-555-0003'
```

Forcing a sequence counter

If a specific value of the sequence counter is required for one instance, the `__sequence` keyword argument should be passed to the factory method.

This will force the sequence counter during the call, without altering the class-level value.

```
class UserFactory(factory.Factory):
    class Meta:
        model = User
    uid = factory.Sequence(int)
```

```
>>> UserFactory()
<User: 0>
>>> UserFactory()
<User: 1>
>>> UserFactory(__sequence=42)
<User: 42>
```

Warning: The impact of setting `__sequence=n` on a `_batch` call is undefined. Each generated instance may share a same counter, or use incremental values starting from the forced value.

LazyAttributeSequence

class `factory.`**`LazyAttributeSequence`**(*method_to_call*)

The *`LazyAttributeSequence`* declaration merges features of *`Sequence`* and *`LazyAttribute`*.

It takes a single argument, a function whose two parameters are, in order:

- The object being built
- The sequence counter

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    login = 'john'
    email = factory.LazyAttributeSequence(lambda o, n: '%s@s%d.example.com' % (o.
↪login, n))
```

```
>>> UserFactory().email
'john@s1.example.com'
>>> UserFactory(login='jack').email
'jack@s2.example.com'
```

Decorator

`factory.`**`lazy_attribute_sequence`**(*method_to_call*)

As for *`lazy_attribute()`* and *`sequence()`*, the *`lazy_attribute_sequence()`* handles more complex cases:

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    login = 'john'

    @lazy_attribute_sequence
    def email(self, n):
        bucket = n % 10
        return '%s@s%d.example.com' % (self.login, bucket)
```

SubFactory

class `factory.`**`SubFactory`**(*factory*, ***kwargs*)

This attribute declaration calls another *`Factory`* subclass, selecting the same build strategy and collecting extra kwargs in the process.

The *`SubFactory`* attribute should be called with:

- A *`Factory`* subclass as first argument, or the fully qualified import path to that *`Factory`* (see *Circular imports*)
- An optional set of keyword arguments that should be passed when calling that factory

Note: When passing an actual *`Factory`* for the `factory` argument, make sure to pass the class and not instance (i.e no `()` after the class):

```
class FooFactory(factory.Factory):
    class Meta:
        model = Foo

    bar = factory.SubFactory(BarFactory)  # Not BarFactory()
```

Definition

```
# A standard factory
class UserFactory(factory.Factory):
    class Meta:
        model = User

    # Various fields
    first_name = 'John'
    last_name = factory.Sequence(lambda n: 'D%se' % ('o' * n))  # De, Doe, Dooe,␣
↪Doooe, ...
    email = factory.LazyAttribute(lambda o: '%s.%s@example.org' % (o.first_name.
↪lower(), o.last_name.lower()))

# A factory for an object with a 'User' field
class CompanyFactory(factory.Factory):
    class Meta:
        model = Company

    name = factory.Sequence(lambda n: 'FactoryBoyz' + 'z' * n)

    # Let's use our UserFactory to create that user, and override its first name.
    owner = factory.SubFactory(UserFactory, first_name='Jack')
```

Calling

The wrapping factory will call of the inner factory:

```
>>> c = CompanyFactory()
>>> c
<Company: FactoryBoyz>

# Notice that the first_name was overridden
>>> c.owner
<User: Jack De>
>>> c.owner.email
jack.de@example.org
```

Fields of the `SubFactory` may be overridden from the external factory:

```
>>> c = CompanyFactory(owner__first_name='Henry')
>>> c.owner
<User: Henry Doe>

# Notice that the updated first_name was propagated to the email LazyAttribute.
>>> c.owner.email
henry.doe@example.org

# It is also possible to override other fields of the SubFactory
>>> c = CompanyFactory(owner__last_name='Jones')
>>> c.owner
<User: Henry Jones>
>>> c.owner.email
henry.jones@example.org
```

Strategies

The strategy chosen for the external factory will be propagated to all subfactories:

```
>>> c = CompanyFactory()
>>> c.pk            # Saved to the database
3
>>> c.owner.pk      # Saved to the database
8

>>> c = CompanyFactory.build()
>>> c.pk            # Not saved
None
>>> c.owner.pk      # Not saved either
None
```

Circular imports

Some factories may rely on each other in a circular manner. This issue can be handled by passing the absolute import path to the target `Factory` to the `SubFactory`.

New in version 1.3.0.

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    username = 'john'
    main_group = factory.SubFactory('users.factories.GroupFactory')

class GroupFactory(factory.Factory):
    class Meta:
        model = Group

    name = "MyGroup"
    owner = factory.SubFactory(UserFactory)
```

Obviously, such circular relationships require careful handling of loops:

```
>>> owner = UserFactory(main_group=None)
>>> UserFactory(main_group__owner=owner)
<john (group: MyGroup)>
```

SelfAttribute

class `factory.`**`SelfAttribute`**`(`*dotted_path_to_attribute*`)`

Some fields should reference another field of the object being constructed, or an attribute thereof.

This is performed by the *`SelfAttribute`* declaration. That declaration takes a single argument, a dot-delimited path to the attribute to fetch:

```
class UserFactory(factory.Factory)
    class Meta:
        model = User

    birthdate = factory.Sequence(lambda n: datetime.date(2000, 1, 1) + datetime.
↪timedelta(days=n))
    birthmonth = factory.SelfAttribute('birthdate.month')
```

```
>>> u = UserFactory()
>>> u.birthdate
date(2000, 3, 15)
>>> u.birthmonth
3
```

Parents

When used in conjunction with *`SubFactory`*, the *`SelfAttribute`* gains an “upward” semantic through the double-dot notation, as used in Python imports.

`factory.SelfAttribute('..country.language')` means “Select the `language` of the `country` of the *`Factory`* calling me”.

```
class UserFactory(factory.Factory):
    class Meta:
        model = User
    language = 'en'


class CompanyFactory(factory.Factory):
    class Meta:
        model = Company

    country = factory.SubFactory(CountryFactory)
    owner = factory.SubFactory(UserFactory, language=factory.SelfAttribute('..country.
↪language'))
```

```
>>> company = CompanyFactory()
>>> company.country.language
'fr'
>>> company.owner.language
'fr'
```

Obviously, this “follow parents” hability also handles overriding some attributes on call:

```
>>> company = CompanyFactory(country=china)
>>> company.owner.language
'cn'
```

This feature is also available to *`LazyAttribute`* and *`LazyAttributeSequence`*, through the `factory_parent` attribute of the passed-in object:

```
class CompanyFactory(factory.Factory):
    class Meta:
        model = Company
    country = factory.SubFactory(CountryFactory)
    owner = factory.SubFactory(UserFactory,
        language=factory.LazyAttribute(lambda user: user.factory_parent.country.
↪language),
    )
```

Iterator

class `factory.`**`Iterator`**(*iterable*, *cycle=True*, *getter=None*)

The *`Iterator`* declaration takes succesive values from the given iterable. When it is exhausted, it starts again from zero (unless `cycle=False`).

`cycle`

The `cycle` argument is only useful for advanced cases, where the provided iterable has no end (as wishing to cycle it means storing values in memory...).

New in version 1.3.0: The `cycle` argument is available as of v1.3.0; previous versions had a behaviour equivalent to `cycle=False`.

`getter`

A custom function called on each value returned by the iterable. See the *Getter* section for details.

New in version 1.3.0.

`reset`()

Reset the internal iterator used by the attribute, so that the next value will be the first value generated by the iterator.

May be called several times.

Each call to the factory will receive the next value from the iterable:

```
class UserFactory(factory.Factory)
    lang = factory.Iterator(['en', 'fr', 'es', 'it', 'de'])
```

```
>>> UserFactory().lang
'en'
>>> UserFactory().lang
'fr'
```

When a value is passed in for the argument, the iterator will *not* be advanced:

```
>>> UserFactory().lang
'en'
>>> UserFactory(lang='cn').lang
'cn'
>>> UserFactory().lang
'fr'
```

Getter

Some situations may reuse an existing iterable, using only some component. This is handled by the `getter` attribute: this is a function that accepts as sole parameter a value from the iterable, and returns an adequate value.

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    # CATEGORY_CHOICES is a list of (key, title) tuples
    category = factory.Iterator(User.CATEGORY_CHOICES, getter=lambda c: c[0])
```

Decorator

`factory.`**`iterator`**(*func*)

When generating items of the iterator gets too complex for a simple list comprehension, use the `iterator()` decorator:

Warning: The decorated function takes **no** argument, notably no `self` parameter.

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    @factory.iterator
    def name():
        with open('test/data/names.dat', 'r') as f:
            for line in f:
                yield line
```

Resetting

In order to start back at the first value in an *Iterator*, simply call the *reset()* method of that attribute (accessing it from the bare *Factory* subclass):

```
>>> UserFactory().lang
'en'
>>> UserFactory().lang
'fr'
>>> UserFactory.lang.reset()
>>> UserFactory().lang
'en'
```

Dict and List

When a factory expects lists or dicts as arguments, such values can be generated through the whole range of factory_boy declarations, with the *Dict* and *List* attributes:

class `factory.`**`Dict`**(*params*[, *dict_factory=factory.DictFactory*])

The *Dict* class is used for dict-like attributes. It receives as non-keyword argument a dictionary of fields to define, whose value may be any factory-enabled declarations:

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    is_superuser = False
    roles = factory.Dict({
        'role1': True,
        'role2': False,
        'role3': factory.Iterator([True, False]),
        'admin': factory.SelfAttribute('..is_superuser'),
    })
```

Note: Declarations used as a *Dict* values are evaluated within that *Dict*'s context; this means that you must use the `..foo` syntax to access fields defined at the factory level.

On the other hand, the *Sequence* counter is aligned on the containing factory's one.

The *Dict* behaviour can be tuned through the following parameters:

`dict_factory`

The actual factory to use for generating the dict can be set as a keyword argument, if an exotic dictionary-like object (SortedDict, ...) is required.

class `factory.`**`List`**(*items*[, *list_factory=factory.ListFactory*])

The *List* can be used for list-like attributes.

Internally, the fields are converted into a `index=value` dict, which makes it possible to override some values at use time:

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    flags = factory.List([
        'user',
        'active',
        'admin',
    ])
```

```
>>> u = UserFactory(flags__2='superadmin')
>>> u.flags
['user', 'active', 'superadmin']
```

The *List* behaviour can be tuned through the following parameters:

list_factory
The actual factory to use for generating the list can be set as a keyword argument, if another type (tuple, set, ...) is required.

Post-generation hooks

Some objects expect additional method calls or complex processing for proper definition. For instance, a `User` may need to have a related `Profile`, where the `Profile` is built from the `User` object.

To support this pattern, factory_boy provides the following tools:

- *PostGenerationMethodCall*: allows you to hook a particular attribute to a function call
- *PostGeneration*: this class allows calling a given function with the generated object as argument
- *post_generation()*: decorator performing the same functions as *PostGeneration*
- *RelatedFactory*: this builds or creates a given factory *after* building/creating the first Factory.

Extracting parameters

All post-building hooks share a common base for picking parameters from the set of attributes passed to the *Factory*.

For instance, a *PostGeneration* hook is declared as `post`:

```
class SomeFactory(factory.Factory):
    class Meta:
        model = SomeObject

    @post_generation
    def post(obj, create, extracted, **kwargs):
        obj.set_origin(create)
```

When calling the factory, some arguments will be extracted for this method:

- If a `post` argument is passed, it will be passed as the `extracted` field
- Any argument starting with `post__XYZ` will be extracted, its `post__` prefix removed, and added to the kwargs passed to the post-generation hook.

Extracted arguments won't be passed to the *model* class.

Thus, in the following call:

```
>>> SomeFactory(
    post=1,
    post_x=2,
    post__y=3,
    post__z__t=42,
)
```

The `post` hook will receive `1` as `extracted` and `{'y': 3, 'z__t': 42}` as keyword arguments; `{'post_x': 2}` will be passed to `SomeFactory._meta.model`.

RelatedFactory

class `factory.`**`RelatedFactory`**(*factory*, *factory_related_name=''*, ***kwargs*)

A `RelatedFactory` behaves mostly like a `SubFactory`, with the main difference that the related `Factory` will be generated *after* the base `Factory`.

`factory`

As for `SubFactory`, the `factory` argument can be:

- A `Factory` subclass
- Or the fully qualified path to a `Factory` subclass (see *Circular imports* for details)

`name`

The generated object (where the `RelatedFactory` attribute will set) may be passed to the related factory if the `factory_related_name` parameter is set.

It will be passed as a keyword argument, using the `name` value as keyword:

Note: When passing an actual `Factory` for the `factory` argument, make sure to pass the class and not instance (i.e no `()` after the class):

```
class FooFactory(factory.Factory):
    class Meta:
        model = Foo

    bar = factory.RelatedFactory(BarFactory)  # Not BarFactory()
```

```
class CityFactory(factory.Factory):
    class Meta:
        model = City

    capital_of = None
    name = "Toronto"

class CountryFactory(factory.Factory):
    class Meta:
        model = Country

    lang = 'fr'
    capital_city = factory.RelatedFactory(CityFactory, 'capital_of', name="Paris")
```

```
>>> france = CountryFactory()
>>> City.objects.get(capital_of=france)
<City: Paris>
```

Extra kwargs may be passed to the related factory, through the usual `ATTR__SUBATTR` syntax:

```
>>> england = CountryFactory(lang='en', capital_city__name="London")
>>> City.objects.get(capital_of=england)
<City: London>
```

If a value if passed for the *RelatedFactory* attribute, this disables *RelatedFactory* generation:

```
>>> france = CountryFactory()
>>> paris = City.objects.get()
>>> paris
<City: Paris>
>>> reunion = CountryFactory(capital_city=paris)
>>> City.objects.count()  # No new capital_city generated
1
>>> guyane = CountryFactory(capital_city=paris, capital_city__name='Kourou')
>>> City.objects.count()  # No new capital_city generated, ``name`` ignored.
1
```

PostGeneration

class `factory.`**`PostGeneration`**`(`*callable*`)`

The *PostGeneration* declaration performs actions once the model object has been generated.

Its sole argument is a callable, that will be called once the base object has been generated.

Once the base object has been generated, the provided callable will be called as `callable(obj, create, extracted, **kwargs)`, where:

- `obj` is the base object previously generated
- `create` is a boolean indicating which strategy was used
- `extracted` is `None` unless a value was passed in for the *PostGeneration* declaration at *Factory* declaration time
- `kwargs` are any extra parameters passed as `attr__key=value` when calling the *Factory*:

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    login = 'john'
    make_mbox = factory.PostGeneration(
            lambda obj, create, extracted, **kwargs: os.makedirs(obj.login))
```

Decorator

`factory.`**`post_generation`**`()`

A decorator is also provided, decorating a single method accepting the same `obj`, `created`, `extracted` and keyword arguments as *PostGeneration*.

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    login = 'john'
    @factory.post_generation
    def mbox(self, create, extracted, **kwargs):
        if not create:
            return
        path = extracted or os.path.join('/tmp/mbox/', self.login)
        os.path.makedirs(path)
        return path
```

```
>>> UserFactory.build()                     # Nothing was created
>>> UserFactory.create()                    # Creates dir /tmp/mbox/john
>>> UserFactory.create(login='jack')        # Creates dir /tmp/mbox/jack
>>> UserFactory.create(mbox='/tmp/alt')     # Creates dir /tmp/alt
```

PostGenerationMethodCall

class factory.**PostGenerationMethodCall**(*method_name*, **args*, ***kwargs*)

The *PostGenerationMethodCall* declaration will call a method on the generated object just after instantiation. This declaration class provides a friendly means of generating attributes of a factory instance during initialization. The declaration is created using the following arguments:

method_name
The name of the method to call on the *model* object

args
The default set of unnamed arguments to pass to the method given in *method_name*

kwargs
The default set of keyword arguments to pass to the method given in *method_name*

Once the factory instance has been generated, the method specified in *method_name* will be called on the generated object with any arguments specified in the *PostGenerationMethodCall* declaration, by default.

For example, to set a default password on a generated User instance during instantiation, we could make a declaration for a `password` attribute like below:

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    username = 'user'
    password = factory.PostGenerationMethodCall('set_password',
                                                'defaultpassword')
```

When we instantiate a user from the `UserFactory`, the factory will create a password attribute by calling `User.set_password('defaultpassword')`. Thus, by default, our users will have a password set to `'defaultpassword'`.

```
>>> u = UserFactory()                             # Calls user.set_password(
↪'defaultpassword')
>>> u.check_password('defaultpassword')
True
```

If the *PostGenerationMethodCall* declaration contained no arguments or one argument, an overriding the value can be passed directly to the method through a keyword argument matching the attribute name. For example we can override the default password specified in the declaration above by simply passing in the desired password as a keyword argument to the factory during instantiation.

```
>>> other_u = UserFactory(password='different')    # Calls user.set_password('different
↪')
>>> other_u.check_password('defaultpassword')
False
>>> other_u.check_password('different')
True
```

Note: For Django models, unless the object method called by *PostGenerationMethodCall* saves the object back to the database, we will have to explicitly remember to save the object back if we performed a `create()`.

```
>>> u = UserFactory.create()  # u.password has not been saved back to the database
>>> u.save()                  # we must remember to do it ourselves
```

We can avoid this by subclassing from `DjangoModelFactory`, instead, e.g.,

```
class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = User

    username = 'user'
    password = factory.PostGenerationMethodCall('set_password',
                                                'defaultpassword')
```

If instead the *PostGenerationMethodCall* declaration uses two or more positional arguments, the overriding value must be an iterable. For example, if we declared the `password` attribute like the following,

```
class UserFactory(factory.Factory):
    class Meta:
        model = User

    username = 'user'
    password = factory.PostGenerationMethodCall('set_password', '', 'sha1')
```

then we must be cautious to pass in an iterable for the `password` keyword argument when creating an instance from the factory:

```
>>> UserFactory()                           # Calls user.set_password('', 'sha1')
>>> UserFactory(password=('test', 'md5'))   # Calls user.set_password('test', 'md5')

>>> # Always pass in a good iterable:
>>> UserFactory(password=('test',))         # Calls user.set_password('test')
>>> UserFactory(password='test')            # Calls user.set_password('t', 'e', 's',
↪'t')
```

Note: While this setup provides sane and intuitive defaults for most users, it prevents passing more than one argument when the declaration used zero or one.

In such cases, users are advised to either resort to the more powerful *PostGeneration* or to add the second expected argument default value to the *PostGenerationMethodCall* declaration (`PostGenerationMethodCall('method', 'x', 'y_that_is_the_default')`)

Keywords extracted from the factory arguments are merged into the defaults present in the *PostGenerationMethodCall* declaration.

```
>>> UserFactory(password__disabled=True)    # Calls user.set_password('', 'sha1',␣
↪disabled=True)
```

Module-level functions

Beyond the *Factory* class and the various *Declarations* classes and methods, factory_boy exposes a few module-level functions, mostly useful for lightweight factory generation.

Lightweight factory declaration

factory.**make_factory**(*klass*, ***kwargs*)

The *make_factory()* function takes a class, declarations as keyword arguments, and generates a new *Factory* for that class accordingly:

```
UserFactory = make_factory(User,
    login='john',
    email=factory.LazyAttribute(lambda u: '%s@example.com' % u.login),
)

# This is equivalent to:

class UserFactory(factory.Factory):
    class Meta:
        model = User

    login = 'john'
    email = factory.LazyAttribute(lambda u: '%s@example.com' % u.login)
```

An alternate base class to *Factory* can be specified in the `FACTORY_CLASS` argument:

```
UserFactory = make_factory(models.User,
    login='john',
    email=factory.LazyAttribute(lambda u: '%s@example.com' % u.login),
    FACTORY_CLASS=factory.django.DjangoModelFactory,
)

# This is equivalent to:

class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.User

    login = 'john'
    email = factory.LazyAttribute(lambda u: '%s@example.com' % u.login)
```

New in version 2.0.0: The `FACTORY_CLASS` kwarg was added in 2.0.0.

Instance building

The `factory` module provides a bunch of shortcuts for creating a factory and extracting instances from them:

factory.**build**(*klass*, *FACTORY_CLASS=None*, ***kwargs*) `factory.`**`build_batch`**(*klass*, *size*, *FACTORY_CLASS=None*, ***kwargs*)

Create a factory for `klass` using declarations passed in kwargs; return an instance built from that factory, or a list of `size` instances (for *`build_batch()`*).

Parameters

- **`klass`** (*`class`*) – Class of the instance to build
- **`size`** (*`int`*) – Number of instances to build
- **`kwargs`** – Declarations to use for the generated factory
- **`FACTORY_CLASS`** – Alternate base class (instead of *`Factory`*)

`factory.`**`create`**(*klass*, *FACTORY_CLASS=None*, ***kwargs*)

`factory.`**`create_batch`**(*klass*, *size*, *FACTORY_CLASS=None*, ***kwargs*)

Create a factory for `klass` using declarations passed in kwargs; return an instance created from that factory, or a list of `size` instances (for *`create_batch()`*).

Parameters

- **`klass`** (*`class`*) – Class of the instance to create
- **`size`** (*`int`*) – Number of instances to create
- **`kwargs`** – Declarations to use for the generated factory
- **`FACTORY_CLASS`** – Alternate base class (instead of *`Factory`*)

`factory.`**`stub`**(*klass*, *FACTORY_CLASS=None*, ***kwargs*)

`factory.`**`stub_batch`**(*klass*, *size*, *FACTORY_CLASS=None*, ***kwargs*)

Create a factory for `klass` using declarations passed in kwargs; return an instance stubbed from that factory, or a list of `size` instances (for *`stub_batch()`*).

Parameters

- **`klass`** (*`class`*) – Class of the instance to stub
- **`size`** (*`int`*) – Number of instances to stub
- **`kwargs`** – Declarations to use for the generated factory
- **`FACTORY_CLASS`** – Alternate base class (instead of *`Factory`*)

`factory.`**`generate`**(*klass*, *strategy*, *FACTORY_CLASS=None*, ***kwargs*)

`factory.`**`generate_batch`**(*klass*, *strategy*, *size*, *FACTORY_CLASS=None*, ***kwargs*)

Create a factory for `klass` using declarations passed in kwargs; return an instance generated from that factory with the `strategy` strategy, or a list of `size` instances (for *`generate_batch()`*).

Parameters

- **`klass`** (*`class`*) – Class of the instance to generate
- **`strategy`** (*`str`*) – The strategy to use
- **`size`** (*`int`*) – Number of instances to generate
- **`kwargs`** – Declarations to use for the generated factory
- **`FACTORY_CLASS`** – Alternate base class (instead of *`Factory`*)

`factory.`**`simple_generate`**(*klass*, *create*, *FACTORY_CLASS=None*, ***kwargs*) `factory.`**`simple_generate_batch`**(*klass*, *create*, *size*, *FACTORY_CLASS=None*, ***kwargs*)

Create a factory for `klass` using declarations passed in kwargs; return an instance generated from that factory according to the *`create`* flag, or a list of `size` instances (for *`simple_generate_batch()`*).

Parameters

- **`klass`** (*`class`*) – Class of the instance to generate
- **`create`** (*`bool`*) – Whether to build (`False`) or create (`True`) instances
- **`size`** (*`int`*) – Number of instances to generate
- **`kwargs`** – Declarations to use for the generated factory
- **`FACTORY_CLASS`** – Alternate base class (instead of *`Factory`*)

Using factory_boy with ORMs

factory_boy provides custom *`Factory`* subclasses for various ORMs, adding dedicated features.

Django

The first versions of factory_boy were designed specifically for Django, but the library has now evolved to be framework-independant.

Most features should thus feel quite familiar to Django users.

The `DjangoModelFactory` subclass

All factories for a Django `Model` should use the *`DjangoModelFactory`* base class.

class `factory.django.`**`DjangoModelFactory`**(*factory.Factory*)

Dedicated class for Django `Model` factories.

This class provides the following features:

- The *`model`* attribute also supports the `'app.Model'` syntax
- *`create()`* uses `Model.objects.create()`
- When using *`RelatedFactory`* or *`PostGeneration`* attributes, the base object will be `saved` once all post-generation hooks have run.

Note: With Django versions 1.8.0 to 1.8.3, it was no longer possible to call `.build()` on a factory if this factory used a *`SubFactory`* pointing to another model: Django refused to set a `ForeignKey` to an unsaved `Model` instance.

See https://code.djangoproject.com/ticket/10811 and https://code.djangoproject.com/ticket/25160 for details.

class `factory.django.`**`DjangoOptions`**(*factory.base.FactoryOptions*)

The `class Meta` on a *`DjangoModelFactory`* supports extra parameters:

`database`

New in version 2.5.0.

All queries to the related model will be routed to the given database. It defaults to `'default'`.

django_get_or_create

New in version 2.4.0.

Fields whose name are passed in this list will be used to perform a `Model.objects.get_or_create()` instead of the usual `Model.objects.create()`:

```
class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = 'myapp.User'  # Equivalent to ``model = myapp.models.User``
        django_get_or_create = ('username',)

    username = 'john'
```

```
>>> User.objects.all()
[]
>>> UserFactory()                   # Creates a new user
<User: john>
>>> User.objects.all()
[<User: john>]

>>> UserFactory()                   # Fetches the existing user
<User: john>
>>> User.objects.all()              # No new user!
[<User: john>]

>>> UserFactory(username='jack')    # Creates another user
<User: jack>
>>> User.objects.all()
[<User: john>, <User: jack>]
```

Note: If a *DjangoModelFactory* relates to an `abstract` model, be sure to declare the *DjangoModelFactory* as abstract:

```
class MyAbstractModelFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.MyAbstractModel
        abstract = True

class MyConcreteModelFactory(MyAbstractModelFactory):
    class Meta:
        model = models.MyConcreteModel
```

Otherwise, factory_boy will try to get the 'next PK' counter from the abstract model.

Extra fields

class `factory.django.`**`FileField`**

Custom declarations for `django.db.models.FileField`

__init__(*self*, *from_path=''*, *from_file=''*, *data=b''*, *filename='example.dat'*)

Parameters

- **from_path** (*str*) – Use data from the file located at `from_path`, and keep its filename
- **from_file** (*file*) – Use the contents of the provided file object; use its filename if available

- **data** (*bytes*) – Use the provided bytes as file contents
- **filename** (*str*) – The filename for the FileField

Note: If the value `None` was passed for the *FileField* field, this will disable field generation:

```
class MyFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.MyModel

    the_file = factory.django.FileField(filename='the_file.dat')
```

```
>>> MyFactory(the_file__data=b'uhuh').the_file.read()
b'uhuh'
>>> MyFactory(the_file=None).the_file
None
```

class `factory.django.`**`ImageField`**

Custom declarations for `django.db.models.ImageField`

`__init__`(*self*, *from_path=''*, *from_file=''*, *filename='example.jpg'*, *width=100*, *height=100*, *color='green'*, *format='JPEG'*)

Parameters

- **from_path** (*str*) – Use data from the file located at `from_path`, and keep its filename
- **from_file** (*file*) – Use the contents of the provided file object; use its filename if available
- **filename** (*str*) – The filename for the ImageField
- **width** (*int*) – The width of the generated image (default: `100`)
- **height** (*int*) – The height of the generated image (default: `100`)
- **color** (*str*) – The color of the generated image (default: `'green'`)
- **format** (*str*) – The image format (as supported by PIL) (default: `'JPEG'`)

Note: If the value `None` was passed for the *FileField* field, this will disable field generation:

Note: Just as Django's `django.db.models.ImageField` requires the Python Imaging Library, this *ImageField* requires it too.

```
class MyFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.MyModel

    the_image = factory.django.ImageField(color='blue')
```

```
>>> MyFactory(the_image__width=42).the_image.width
42
>>> MyFactory(the_image=None).the_image
None
```

Disabling signals

Signals are often used to plug some custom code into external components code; for instance to create `Profile` objects on-the-fly when a new `User` object is saved.

This may interfere with finely tuned *`factories`*, which would create both using *`RelatedFactory`*.

To work around this problem, use the *`mute_signals()`* decorator/context manager:

`factory.django.`**`mute_signals`**`(`*signal1, ...*`)`

Disable the list of selected signals when calling the factory, and reactivate them upon leaving.

```
# foo/factories.py

import factory
import factory.django

from . import models
from . import signals

@factory.django.mute_signals(signals.pre_save, signals.post_save)
class FooFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.Foo

    # ...

def make_chain():
    with factory.django.mute_signals(signals.pre_save, signals.post_save):
        # pre_save/post_save won't be called here.
        return SomeFactory(), SomeOtherFactory()
```

Mogo

factory_boy supports Mogo-style models, through the *`MogoFactory`* class.

Mogo is a wrapper around the `pymongo` library for MongoDB.

class `factory.mogo.`**`MogoFactory`**`(`*factory.Factory*`)`

Dedicated class for Mogo models.

This class provides the following features:

- *`build()`* calls a model's `new()` method
- *`create()`* builds an instance through `new()` then saves it.

MongoEngine

factory_boy supports MongoEngine-style models, through the *`MongoEngineFactory`* class.

mongoengine is a wrapper around the `pymongo` library for MongoDB.

class `factory.mongoengine.`**`MongoEngineFactory`**`(`*factory.Factory*`)`

Dedicated class for MongoEngine models.

This class provides the following features:

- *`build()`* calls a model's `__init__` method

- *`create()`* builds an instance through `__init__` then saves it.

Note: If the `associated class <factory.FactoryOptions.model` is a `mongoengine.EmbeddedDocument`, the `create()` function won't "save" it, since this wouldn't make sense.

This feature makes it possible to use *`SubFactory`* to create embedded document.

A minimalist example:

```
import mongoengine

class Address(mongoengine.EmbeddedDocument):
    street = mongoengine.StringField()

class Person(mongoengine.Document):
    name = mongoengine.StringField()
    address = mongoengine.EmbeddedDocumentField(Address)

import factory

class AddressFactory(factory.mongoengine.MongoEngineFactory):
    class Meta:
        model = Address

    street = factory.Sequence(lambda n: 'street%d' % n)

class PersonFactory(factory.mongoengine.MongoEngineFactory):
    class Meta:
        model = Person

    name = factory.Sequence(lambda n: 'name%d' % n)
    address = factory.SubFactory(AddressFactory)
```

SQLAlchemy

Factoy_boy also supports SQLAlchemy models through the *`SQLAlchemyModelFactory`* class.

To work, this class needs an SQLAlchemy session object affected to the *`Meta.sqlalchemy_session`* attribute.

class `factory.alchemy.`**`SQLAlchemyModelFactory`**(*factory.Factory*)

Dedicated class for SQLAlchemy models.

This class provides the following features:

- *`create()`* uses `sqlalchemy.orm.session.Session.add()`

class `factory.alchemy.`**`SQLAlchemyOptions`**(*factory.base.FactoryOptions*)

In addition to the usual parameters available in `class Meta`, a *`SQLAlchemyModelFactory`* also supports the following settings:

`sqlalchemy_session`

SQLAlchemy session to use to communicate with the database when creating an object through this *`SQLAlchemyModelFactory`*.

A (very) simple example:

```
from sqlalchemy import Column, Integer, Unicode, create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import scoped_session, sessionmaker

engine = create_engine('sqlite://')
session = scoped_session(sessionmaker(bind=engine))
Base = declarative_base()


class User(Base):
    """ A SQLAlchemy simple model class who represents a user """
    __tablename__ = 'UserTable'

    id = Column(Integer(), primary_key=True)
    name = Column(Unicode(20))

Base.metadata.create_all(engine)

import factory

class UserFactory(factory.alchemy.SQLAlchemyModelFactory):
    class Meta:
        model = User
        sqlalchemy_session = session   # the SQLAlchemy session object

    id = factory.Sequence(lambda n: n)
    name = factory.Sequence(lambda n: u'User %d' % n)
```

```
>>> session.query(User).all()
[]
>>> UserFactory()
<User: User 1>
>>> session.query(User).all()
[<User: User 1>]
```

Managing sessions

Since SQLAlchemy is a general purpose library, there is no "global" session management system.

The most common pattern when working with unit tests and `factory_boy` is to use SQLAlchemy's `sqlalchemy.orm.scoping.scoped_session`:

- The test runner configures some project-wide `scoped_session`
- Each *SQLAlchemyModelFactory* subclass uses this `scoped_session` as its *sqlalchemy_session*
- The `tearDown()` method of tests calls `Session.remove` to reset the session.

Note: See the excellent SQLAlchemy guide on scoped_session for details of `scoped_session`'s usage.

The basic idea is that declarative parts of the code (including factories) need a simple way to access the "current session", but that session will only be created and configured at a later point.

The `scoped_session` handles this, by virtue of only creating the session when a query is sent to the database.

Here is an example layout:

- A global (test-only?) file holds the `scoped_session`:

```
# myprojet/test/common.py

from sqlalchemy import orm
Session = orm.scoped_session(orm.sessionmaker())
```

- All factory access it:

```
# myproject/factories.py

import factory
import factory.alchemy

from . import models
from .test import common

class UserFactory(factory.alchemy.SQLAlchemyModelFactory):
    class Meta:
        model = models.User

        # Use the not-so-global scoped_session
        # Warning: DO NOT USE common.Session()!
        sqlalchemy_session = common.Session

    name = factory.Sequence(lambda n: "User %d" % n)
```

- The test runner configures the `scoped_session` when it starts:

```
# myproject/test/runtests.py

import sqlalchemy

from . import common

def runtests():
    engine = sqlalchemy.create_engine('sqlite://')

    # It's a scoped_session, and now is the time to configure it.
    common.Session.configure(bind=engine)

    run_the_tests
```

- `test cases` use this `scoped_session`, and clear it after each test (for isolation):

```
# myproject/test/test_stuff.py

import unittest

from . import common

class MyTest(unittest.TestCase):

    def setUp(self):
        # Prepare a new, clean session
        self.session = common.Session()

    def test_something(self):
        u = factories.UserFactory()
        self.assertEqual([u], self.session.query(User).all())
    def tearDown(self):
        # Rollback the session => no changes to the database
        self.session.rollback()
        # Remove it, so that the next test gets a new Session()
        common.Session.remove()
```

Common recipes

Note: Most recipes below take on Django model examples, but can also be used on their own.

Dependent objects (ForeignKey)

When one attribute is actually a complex field (e.g a `ForeignKey` to another `Model`), use the *`SubFactory`* declaration:

```
# models.py
class User(models.Model):
    first_name = models.CharField()
    group = models.ForeignKey(Group)


# factories.py
import factory
from . import models

class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.User

    first_name = factory.Sequence(lambda n: "Agent %03d" % n)
    group = factory.SubFactory(GroupFactory)
```

Choosing from a populated table

If the target of the `ForeignKey` should be chosen from a pre-populated table (e.g `django.contrib.contenttypes.models.ContentType`), simply use a *`factory.Iterator`* on the chosen queryset:

```
import factory, factory.django
from . import models

class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.User

    language = factory.Iterator(models.Language.objects.all())
```

Here, `models.Language.objects.all()` won't be evaluated until the first call to `UserFactory`; thus avoiding DB queries at import time.

Reverse dependencies (reverse ForeignKey)

When a related object should be created upon object creation (e.g a reverse `ForeignKey` from another `Model`), use a *`RelatedFactory`* declaration:

```
# models.py
class User(models.Model):
    pass

class UserLog(models.Model):
    user = models.ForeignKey(User)
    action = models.CharField()


# factories.py
class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.User

    log = factory.RelatedFactory(UserLogFactory, 'user', action=models.UserLog.ACTION_
↪CREATE)
```

When a `UserFactory` is instantiated, factory_boy will call `UserLogFactory(user=that_user, action=...)` just before returning the created `User`.

Django (<1.5) provided a mechanism to attach a `Profile` to a `User` instance, using a `OneToOneField` from the `Profile` to the `User`.

A typical way to create those profiles was to hook a post-save signal to the `User` model.

factory_boy allows to define attributes of such profiles dynamically when creating a `User`:

```
class ProfileFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = my_models.Profile

    title = 'Dr'
    # We pass in profile=None to prevent UserFactory from creating another profile
    # (this disables the RelatedFactory)
    user = factory.SubFactory('app.factories.UserFactory', profile=None)

class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = auth_models.User

    username = factory.Sequence(lambda n: "user_%d" % n)

    # We pass in 'user' to link the generated Profile to our just-generated User
    # This will call ProfileFactory(user=our_new_user), thus skipping the SubFactory.
    profile = factory.RelatedFactory(ProfileFactory, 'user')

    @classmethod
    def _generate(cls, create, attrs):
        """Override the default _generate() to disable the post-save signal."""

        # Note: If the signal was defined with a dispatch_uid, include that in both␣
↪calls.
        post_save.disconnect(handler_create_user_profile, auth_models.User)
        user = super(UserFactory, cls)._generate(create, attrs)
        post_save.connect(handler_create_user_profile, auth_models.User)
        return user
```

```
>>> u = UserFactory(profile__title=u"Lord")
>>> u.get_profile().title
u"Lord"
```

Such behaviour can be extended to other situations where a signal interferes with factory_boy related factories.

Note: When any *RelatedFactory* or *post_generation* attribute is defined on the *DjangoModelFactory* subclass, a second `save()` is performed *after* the call to `_create()`.

Code working with signals should thus override the `_generate()` method.

Simple Many-to-many relationship

Building the adequate link between two models depends heavily on the use case; factory_boy doesn't provide a "all in one tools" as for *SubFactory* or *RelatedFactory*, users will have to craft their own depending on the model.

The base building block for this feature is the *post_generation* hook:

```
# models.py
class Group(models.Model):
    name = models.CharField()

class User(models.Model):
    name = models.CharField()
    groups = models.ManyToManyField(Group)


# factories.py
class GroupFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.Group

    name = factory.Sequence(lambda n: "Group #%s" % n)

class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.User

    name = "John Doe"

    @factory.post_generation
    def groups(self, create, extracted, **kwargs):
        if not create:
            # Simple build, do nothing.
            return

        if extracted:
            # A list of groups were passed in, use them
            for group in extracted:
                self.groups.add(group)
```

When calling `UserFactory()` or `UserFactory.build()`, no group binding will be created.

But when `UserFactory.create(groups=(group1, group2, group3))` is called, the `groups` declaration will add passed in groups to the set of groups for the user.

Many-to-many relation with a 'through'

If only one link is required, this can be simply performed with a `RelatedFactory`. If more links are needed, simply add more `RelatedFactory` declarations:

```
# models.py
class User(models.Model):
    name = models.CharField()

class Group(models.Model):
    name = models.CharField()
    members = models.ManyToManyField(User, through='GroupLevel')

class GroupLevel(models.Model):
    user = models.ForeignKey(User)
    group = models.ForeignKey(Group)
    rank = models.IntegerField()


# factories.py
class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.User

    name = "John Doe"

class GroupFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.Group

    name = "Admins"

class GroupLevelFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.GroupLevel

    user = factory.SubFactory(UserFactory)
    group = factory.SubFactory(GroupFactory)
    rank = 1

class UserWithGroupFactory(UserFactory):
    membership = factory.RelatedFactory(GroupLevelFactory, 'user')

class UserWith2GroupsFactory(UserFactory):
    membership1 = factory.RelatedFactory(GroupLevelFactory, 'user', group__name=
↪'Group1')
    membership2 = factory.RelatedFactory(GroupLevelFactory, 'user', group__name=
↪'Group2')
```

Whenever the `UserWithGroupFactory` is called, it will, as a post-generation hook, call the `GroupLevelFactory`, passing the generated user as a `user` field:

1. `UserWithGroupFactory()` generates a `User` instance, `obj`
2. It calls `GroupLevelFactory(user=obj)`

3. It returns `obj`

When using the `UserWith2GroupsFactory`, that behavior becomes:

1. `UserWith2GroupsFactory()` generates a `User` instance, `obj`
2. It calls `GroupLevelFactory(user=obj, group__name='Group1')`
3. It calls `GroupLevelFactory(user=obj, group__name='Group2')`
4. It returns `obj`

Copying fields to a SubFactory

When a field of a related class should match one of the container:

```
# models.py
class Country(models.Model):
    name = models.CharField()
    lang = models.CharField()

class User(models.Model):
    name = models.CharField()
    lang = models.CharField()
    country = models.ForeignKey(Country)

class Company(models.Model):
    name = models.CharField()
    owner = models.ForeignKey(User)
    country = models.ForeignKey(Country)
```

Here, we want:

- The User to have the lang of its country (`factory.SelfAttribute('country.lang')`)
- The Company owner to live in the country of the company (`factory.SelfAttribute('..country')`)

```
# factories.py
class CountryFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.Country

    name = factory.Iterator(["France", "Italy", "Spain"])
    lang = factory.Iterator(['fr', 'it', 'es'])

class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.User

    name = "John"
    lang = factory.SelfAttribute('country.lang')
    country = factory.SubFactory(CountryFactory)

class CompanyFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.Company

    name = "ACME, Inc."
    country = factory.SubFactory(CountryFactory)
    owner = factory.SubFactory(UserFactory, country=factory.SelfAttribute('..country
↪'))
```

Custom manager methods

Sometimes you need a factory to call a specific manager method other then the default `Model.objects.create()` method:

```
class UserFactory(factory.DjangoModelFactory):
    class Meta:
        model = UserenaSignup

    username = "l7d8s"
    email = "my_name@example.com"
    password = "my_password"

    @classmethod
    def _create(cls, model_class, *args, **kwargs):
        """Override the default ``_create`` with our custom call."""
        manager = cls._get_manager(model_class)
        # The default would use ``manager.create(*args, **kwargs)``
        return manager.create_user(*args, **kwargs)
```

Forcing the sequence counter

A common pattern with factory_boy is to use a *factory.Sequence* declaration to provide varying values to attributes declared as unique.

However, it is sometimes useful to force a given value to the counter, for instance to ensure that tests are properly reproductible.

factory_boy provides a few hooks for this:

Forcing the value on a per-call basis In order to force the counter for a specific *Factory* instantiation, just pass the value in the `__sequence=42` parameter:

```
class AccountFactory(factory.Factory):
    class Meta:
        model = Account
    uid = factory.Sequence(lambda n: n)
    name = "Test"
```

```
>>> obj1 = AccountFactory(name="John Doe", __sequence=10)
>>> obj1.uid  # Taken from the __sequence counter
10
>>> obj2 = AccountFactory(name="Jane Doe")
>>> obj2.uid  # The base sequence counter hasn't changed
1
```

Resetting the counter globally If all calls for a factory must start from a deterministic number, use *factory.Factory.reset_sequence()*; this will reset the counter to its initial value (as defined by *factory.Factory._setup_next_sequence()*).

```
>>> AccountFactory().uid
1
>>> AccountFactory().uid
2
```

```
>>> AccountFactory.reset_sequence()
>>> AccountFactory().uid  # Reset to the initial value
1
>>> AccountFactory().uid
2
```

It is also possible to reset the counter to a specific value:

```
>>> AccountFactory.reset_sequence(10)
>>> AccountFactory().uid
10
>>> AccountFactory().uid
11
```

This recipe is most useful in a `TestCase`'s `setUp()` method.

Forcing the initial value for all projects The sequence counter of a *`Factory`* can also be set automatically upon the first call through the *`_setup_next_sequence()`* method; this helps when the objects's attributes mustn't conflict with pre-existing data.

A typical example is to ensure that running a Python script twice will create non-conflicting objects, by setting up the counter to "max used value plus one":

```
class AccountFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = models.Account

    @classmethod
    def _setup_next_sequence(cls):
        try:
            return models.Accounts.objects.latest('uid').uid + 1
        except models.Account.DoesNotExist:
            return 1
```

```
>>> Account.objects.create(uid=42, name="Blah")
>>> AccountFactory.create()  # Sets up the account number based on the latest uid
<Account uid=43, name=Test>
```

Fuzzy attributes

Some tests may be interested in testing with fuzzy, random values.

This is handled by the *`factory.fuzzy`* module, which provides a few random declarations.

Note: Use `import factory.fuzzy` to load this module.

FuzzyAttribute

class `factory.fuzzy.`**`FuzzyAttribute`**

The *`FuzzyAttribute`* uses an arbitrary callable as fuzzer. It is expected that successive calls of that function return various values.

fuzzer
The callable that generates random values

FuzzyText

class factory.fuzzy.**FuzzyText**(*length=12*, *chars=string.ascii_letters*, *prefix='‘*)
The *FuzzyText* fuzzer yields random strings beginning with the given *prefix*, followed by *length* charactes chosen from the *chars* character set, and ending with the given *suffix*.

length
int, the length of the random part

prefix
text, an optional prefix to prepend to the random part

suffix
text, an optional suffix to append to the random part

chars

char iterable, the chars to choose from; defaults to the list of ascii letters and numbers.

FuzzyChoice

class factory.fuzzy.**FuzzyChoice**(*choices*)
The *FuzzyChoice* fuzzer yields random choices from the given iterable.

Note: The passed in *choices* will be converted into a list upon first use, not at declaration time.

This allows passing in, for instance, a Django queryset that will only hit the database during the database, not at import time.

choices
The list of choices to select randomly

FuzzyInteger

class factory.fuzzy.**FuzzyInteger**(*low*[, *high*[, *step*]])
The *FuzzyInteger* fuzzer generates random integers within a given inclusive range.

The *low* bound may be omitted, in which case it defaults to 0:

```
>>> fi = FuzzyInteger(0, 42)
>>> fi.low, fi.high
0, 42

>>> fi = FuzzyInteger(42)
>>> fi.low, fi.high
0, 42
```

low
int, the inclusive lower bound of generated integers

high
int, the inclusive higher bound of generated integers

step
int, the step between values in the range; for instance, a `FuzzyInteger(0, 42, step=3)` might only yield values from `[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42]`.

FuzzyDecimal

class `factory.fuzzy.`**`FuzzyDecimal`**(*low*[, *high*[, *precision=2*]])
The *`FuzzyDecimal`* fuzzer generates random `decimals` within a given inclusive range.

The *`low`* bound may be omitted, in which case it defaults to 0:

```
>>> FuzzyDecimal(0.5, 42.7)
>>> fi.low, fi.high
0.5, 42.7

>>> fi = FuzzyDecimal(42.7)
>>> fi.low, fi.high
0.0, 42.7

>>> fi = FuzzyDecimal(0.5, 42.7, 3)
>>> fi.low, fi.high, fi.precision
0.5, 42.7, 3
```

low
decimal, the inclusive lower bound of generated decimals

high
decimal, the inclusive higher bound of generated decimals

precision
`int, the number of digits to generate after the dot. The default is 2 digits.`

FuzzyFloat

class `factory.fuzzy.`**`FuzzyFloat`**(*low*[, *high*])
The *`FuzzyFloat`* fuzzer provides random `float` objects within a given inclusive range.

```
>>> FuzzyFloat(0.5, 42.7)
>>> fi.low, fi.high
0.5, 42.7

>>> fi = FuzzyFloat(42.7)
>>> fi.low, fi.high
0.0, 42.7
```

low
decimal, the inclusive lower bound of generated floats

high
decimal, the inclusive higher bound of generated floats

FuzzyDate

class `factory.fuzzy.`**`FuzzyDate`**(*start_date*[, *end_date*])

The *FuzzyDate* fuzzer generates random dates within a given inclusive range.

The *end_date* bound may be omitted, in which case it defaults to the current date:

```
>>> fd = FuzzyDate(datetime.date(2008, 1, 1))
>>> fd.start_date, fd.end_date
datetime.date(2008, 1, 1), datetime.date(2013, 4, 16)
```

`start_date`

`datetime.date`, the inclusive lower bound of generated dates

`end_date`

`datetime.date`, the inclusive higher bound of generated dates

FuzzyDateTime

class `factory.fuzzy.`**`FuzzyDateTime`**(*start_dt*[, *end_dt*], *tz=UTC*, *force_year=None*, *force_month=None*, *force_day=None*, *force_hour=None*, *force_minute=None*, *force_second=None*, *force_microsecond=None*)

The *FuzzyDateTime* fuzzer generates random timezone-aware datetime within a given inclusive range.

The *end_dt* bound may be omitted, in which case it defaults to `datetime.datetime.now()` localized into the UTC timezone.

```
>>> fdt = FuzzyDateTime(datetime.datetime(2008, 1, 1, tzinfo=UTC))
>>> fdt.start_dt, fdt.end_dt
datetime.datetime(2008, 1, 1, tzinfo=UTC), datetime.datetime(2013, 4, 21, 19, 13,␣
↪32, 458487, tzinfo=UTC)
```

The `force_XXX` keyword arguments force the related value of generated datetimes:

```
>>> fdt = FuzzyDateTime(datetime.datetime(2008, 1, 1, tzinfo=UTC), datetime.
↪datetime(2009, 1, 1, tzinfo=UTC),
...     force_day=3, force_second=42)
>>> fdt.evaluate(2, None, False)  # Actual code used by ``SomeFactory.build()``
datetime.datetime(2008, 5, 3, 12, 13, 42, 124848, tzinfo=UTC)
```

`start_dt`

`datetime.datetime`, the inclusive lower bound of generated datetimes

`end_dt`

`datetime.datetime`, the inclusive upper bound of generated datetimes

`force_year`

int or None; if set, forces the `year` of generated datetime.

`force_month`

int or None; if set, forces the `month` of generated datetime.

`force_day`

int or None; if set, forces the `day` of generated datetime.

`force_hour`

int or None; if set, forces the `hour` of generated datetime.

force_minute
int or None; if set, forces the `minute` of generated datetime.

force_second
int or None; if set, forces the `second` of generated datetime.

force_microsecond
int or None; if set, forces the `microsecond` of generated datetime.

FuzzyNaiveDateTime

class `factory.fuzzy.`**`FuzzyNaiveDateTime`**(*start_dt*[, *end_dt*], *force_year=None*, *force_month=None*, *force_day=None*, *force_hour=None*, *force_minute=None*, *force_second=None*, *force_microsecond=None*)

The *`FuzzyNaiveDateTime`* fuzzer generates random naive datetime within a given inclusive range.

The *`end_dt`* bound may be omitted, in which case it defaults to `datetime.datetime.now()`:

```
>>> fdt = FuzzyNaiveDateTime(datetime.datetime(2008, 1, 1))
>>> fdt.start_dt, fdt.end_dt
datetime.datetime(2008, 1, 1), datetime.datetime(2013, 4, 21, 19, 13, 32, 458487)
```

The `force_XXX` keyword arguments force the related value of generated datetimes:

```
>>> fdt = FuzzyNaiveDateTime(datetime.datetime(2008, 1, 1), datetime.
↪datetime(2009, 1, 1),
...     force_day=3, force_second=42)
>>> fdt.evaluate(2, None, False)  # Actual code used by ``SomeFactory.build()``
datetime.datetime(2008, 5, 3, 12, 13, 42, 124848)
```

start_dt
`datetime.datetime`, the inclusive lower bound of generated datetimes

end_dt
`datetime.datetime`, the inclusive upper bound of generated datetimes

force_year
int or None; if set, forces the `year` of generated datetime.

force_month
int or None; if set, forces the `month` of generated datetime.

force_day
int or None; if set, forces the `day` of generated datetime.

force_hour
int or None; if set, forces the `hour` of generated datetime.

force_minute
int or None; if set, forces the `minute` of generated datetime.

force_second
int or None; if set, forces the `second` of generated datetime.

force_microsecond
int or None; if set, forces the `microsecond` of generated datetime.

Custom fuzzy fields

Alternate fuzzy fields may be defined. They should inherit from the `BaseFuzzyAttribute` class, and override its `fuzz()` method.

class `factory.fuzzy.`**`BaseFuzzyAttribute`**

Base class for all fuzzy attributes.

`fuzz`(*self*)

The method responsible for generating random values. *Must* be overridden in subclasses.

Managing randomness

Using `random` in factories allows to "fuzz" a program efficiently. However, it's sometimes required to *reproduce* a failing test.

`factory.fuzzy` uses a separate instance of `random.Random`, and provides a few helpers for this:

`factory.fuzzy.`**`get_random_state`**()

Call `get_random_state()` to retrieve the random generator's current state.

`factory.fuzzy.`**`set_random_state`**(*state*)

Use `set_random_state()` to set a custom state into the random generator (fetched from `get_random_state()` in a previous run, for instance)

`factory.fuzzy.`**`reseed_random`**(*seed*)

The `reseed_random()` function allows to load a chosen seed into the random generator.

Custom `BaseFuzzyAttribute` subclasses **SHOULD** use `factory.fuzzy._random` as a randomness source; this ensures that data they generate can be regenerated using the simple state from `get_random_state()`.

Examples

Here are some real-world examples of using FactoryBoy.

Objects

First, let's define a couple of objects:

```python
class Account(object):
    def __init__(self, username, email):
        self.username = username
        self.email = email

    def __str__(self):
        return '%s (%s)' % (self.username, self.email)


class Profile(object):

    GENDER_MALE = 'm'
    GENDER_FEMALE = 'f'
    GENDER_UNKNOWN = 'u'  # If the user refused to give it

    def __init__(self, account, gender, firstname, lastname, planet='Earth'):
        self.account = account
        self.gender = gender
        self.firstname = firstname
        self.lastname = lastname
        self.planet = planet

    def __unicode__(self):
        return u'%s %s (%s)' % (
            unicode(self.firstname),
            unicode(self.lastname),
            unicode(self.account.accountname),
        )
```

Factories

And now, we'll define the related factories:

```
import factory
import random

from . import objects


class AccountFactory(factory.Factory):
    class Meta:
        model = objects.Account

    username = factory.Sequence(lambda n: 'john%s' % n)
    email = factory.LazyAttribute(lambda o: '%s@example.org' % o.username)


class ProfileFactory(factory.Factory):
    class Meta:
        model = objects.Profile

    account = factory.SubFactory(AccountFactory)
    gender = factory.Iterator([objects.Profile.GENDER_MALE, objects.Profile.GENDER_
↪FEMALE])
    firstname = u'John'
    lastname = u'Doe'
```

We have now defined basic factories for our `Account` and `Profile` classes.

If we commonly use a specific variant of our objects, we can refine a factory accordingly:

```
class FemaleProfileFactory(ProfileFactory):
    gender = objects.Profile.GENDER_FEMALE
    firstname = u'Jane'
    user__username = factory.Sequence(lambda n: 'jane%s' % n)
```

Using the factories

We can now use our factories, for tests:

```python
import unittest

from . import business_logic
from . import factories
from . import objects


class MyTestCase(unittest.TestCase):

    def test_send_mail(self):
        account = factories.AccountFactory()
        email = business_logic.prepare_email(account, subject='Foo', text='Bar')

        self.assertEqual(email.to, account.email)

    def test_get_profile_stats(self):
        profiles = []

        profiles.extend(factories.ProfileFactory.create_batch(4))
        profiles.extend(factories.FemaleProfileFactory.create_batch(2))
        profiles.extend(factories.ProfileFactory.create_batch(2, planet="Tatooine"))

        stats = business_logic.profile_stats(profiles)
        self.assertEqual({'Earth': 6, 'Mars': 2}, stats.planets)
        self.assertLess(stats.genders[objects.Profile.GENDER_FEMALE], 2)
```

Or for fixtures:

```python
from . import factories

def make_objects():
    factories.ProfileFactory.create_batch(size=50)

    # Let's create a few, known objects.
    factories.ProfileFactory(
        gender=objects.Profile.GENDER_MALE,
        firstname='Luke',
        lastname='Skywalker',
        planet='Tatooine',
    )

    factories.ProfileFactory(
        gender=objects.Profile.GENDER_FEMALE,
        firstname='Leia',
        lastname='Organa',
        planet='Alderaan',
    )
```

Internals

ChangeLog

2.6.0 (XXXX-XX-XX)

New:

- Add `factory.FactoryOptions.rename` to help handle conflicting names (issue #206)
- Add support for random-yet-realistic values through fake-factory, through the `factory.Faker` class.
- `factory.Iterator` no longer begins iteration of its argument at import time, thus allowing to pass in a lazy iterator such as a Django queryset (i.e `factory.Iterator(models.MyThingy.objects.all())`).
- Simplify imports for ORM layers, now available through a simple `factory` import, at `factory.alchemy.SQLAlchemyModelFactory` / `factory.django.DjangoModelFactory` / `factory.mongoengine.MongoEngineFactory`.

Bugfix:

- issue #201: Properly handle custom Django managers when dealing with abstract Django models.
- issue #212: Fix `factory.django.mute_signals()` to handle Django's signal caching
- issue #228: Don't load `django.apps.apps.get_model()` until required
- issue #219: Stop using `mogo.model.Model.new()`, deprecated 4 years ago.

2.5.2 (2015-04-21)

Bugfix:

- Add support for Django 1.7/1.8
- Add support for mongoengine>=0.9.0 / pymongo>=2.1

2.5.1 (2015-03-27)

Bugfix:

- Respect custom managers in `DjangoModelFactory` (see issue #192)
- Allow passing declarations (e.g `Sequence`) as parameters to `FileField` and `ImageField`.

2.5.0 (2015-03-26)

New:

- Add support for getting/setting `factory.fuzzy`'s random state (see issue #175, issue #185).
- Support lazy evaluation of iterables in `factory.fuzzy.FuzzyChoice` (see issue #184).
- Support non-default databases at the factory level (see issue #171)
- Make `factory.django.FileField` and `factory.django.ImageField` non-post_generation, i.e normal fields also available in `save()` (see issue #141).

Bugfix:

- Avoid issues when using *factory.django.mute_signals()* on a base factory class (see issue #183).
- Fix limitations of *factory.StubFactory*, that can now use *factory.SubFactory* and co (see issue #131).

Deprecation:

- Remove deprecated features from *2.4.0 (2014-06-21)*
- Remove the auto-magical sequence setup (based on the latest primary key value in the database) for Django and SQLAlchemy; this relates to issues issue #170, issue #153, issue #111, issue #103, issue #92, issue #78. See https://github.com/rbarrois/factory_boy/commit/13d310f for technical details.

Warning: Version 2.5.0 removes the 'auto-magical sequence setup' bug-and-feature. This could trigger some bugs when tests expected a non-zero sequence reference.

Upgrading

Warning: Version 2.5.0 removes features that were marked as deprecated in *v2.4.0*.

All `FACTORY_*`-style attributes are now declared in a `class Meta:` section:

```python
# Old-style, deprecated
class MyFactory(factory.Factory):
    FACTORY_FOR = models.MyModel
    FACTORY_HIDDEN_ARGS = ['a', 'b', 'c']

# New-style
class MyFactory(factory.Factory):
    class Meta:
        model = models.MyModel
        exclude = ['a', 'b', 'c']
```

A simple shell command to upgrade the code would be:

```sh
# sed -i: inplace update
# grep -l: only file names, not matching lines
sed -i 's/FACTORY_FOR =/class Meta:\n        model =/' $(grep -l FACTORY_FOR $(find .␣
↪-name '*.py'))
```

This takes care of all `FACTORY_FOR` occurences; the files containing other attributes to rename can be found with `grep -R FACTORY .`

2.4.1 (2014-06-23)

Bugfix:

- Fix overriding deeply inherited attributes (set in one factory, overridden in a subclass, used in a sub-sub-class).

2.4.0 (2014-06-21)

New:

- Add support for *factory.fuzzy.FuzzyInteger.step*, thanks to ilya-pirogov (issue #120)
- Add *mute_signals()* decorator to temporarily disable some signals, thanks to ilya-pirogov (issue #122)
- Add *FuzzyFloat* (issue #124)
- Declare target model and other non-declaration fields in a `class Meta` section.

Deprecation:

- Use of `FACTORY_FOR` and other `FACTORY` class-level attributes is deprecated and will be removed in 2.5. Those attributes should now declared within the *class Meta* attribute:

  For *factory.Factory*:

  – Rename `FACTORY_FOR` to *model*

  – Rename `FACTORY_FOR` to *model*

  – Rename `ABSTRACT_FACTORY` to *abstract*

  – Rename `FACTORY_STRATEGY` to *strategy*

  – Rename `FACTORY_ARG_PARAMETERS` to *inline_args*

  – Rename `FACTORY_HIDDEN_ARGS` to *exclude*

  For *factory.django.DjangoModelFactory*:

  – Rename `FACTORY_DJANGO_GET_OR_CREATE` to *django_get_or_create*

  For *factory.alchemy.SQLAlchemyModelFactory*:

  – Rename `FACTORY_SESSION` to *sqlalchemy_session*

2.3.1 (2014-01-22)

Bugfix:

- Fix badly written assert containing state-changing code, spotted by chsigi (issue #126)
- Don't crash when handling objects whose __repr__ is non-pure-ascii bytes on Py2, discovered by mbertheau (issue #123) and strycore (issue #127)

2.3.0 (2013-12-25)

New:

- Add *FuzzyText*, thanks to jdufresne (issue #97)
- Add *FuzzyDecimal*, thanks to thedrow (issue #94)
- Add support for `EmbeddedDocument`, thanks to imiric (issue #100)

2.2.1 (2013-09-24)

Bugfix:

- Fixed sequence counter for *DjangoModelFactory* when a factory inherits from another factory relating to an abstract model.

2.2.0 (2013-09-24)

Bugfix:

- Removed duplicated *SQLAlchemyModelFactory* lurking in `factory` (issue #83)
- Properly handle sequences within object inheritance chains. If FactoryA inherits from FactoryB, and their associated classes share the same link, sequence counters will be shared (issue #93)
- Properly handle nested *SubFactory* overrides

New:

- The *DjangoModelFactory* now supports the `FACTORY_FOR = 'myapp.MyModel'` syntax, making it easier to shove all factories in a single module (issue #66).
- Add *factory.debug()* helper for easier backtrace analysis
- Adding factory support for mongoengine with *MongoEngineFactory*.

2.1.2 (2013-08-14)

New:

- The `ABSTRACT_FACTORY` keyword is now optional, and automatically set to `True` if neither the *Factory* subclass nor its parent declare the `FACTORY_FOR` attribute (issue #74)

2.1.1 (2013-07-02)

Bugfix:

- Properly retrieve the `color` keyword argument passed to *ImageField*

2.1.0 (2013-06-26)

New:

- Add *FuzzyDate* thanks to saulshanabrook
- Add *FuzzyDateTime* and *FuzzyNaiveDateTime*.
- Add a `factory_parent` attribute to the `LazyStub` passed to *LazyAttribute*, in order to access fields defined in wrapping factories.
- Move *DjangoModelFactory* and *MogoFactory* to their own modules (`factory.django` and `factory.mogo`)
- Add the *reset_sequence()* classmethod to *Factory* to ease resetting the sequence counter for a given factory.
- Add debug messages to `factory` logger.
- Add a *reset()* method to *Iterator* (issue #63)
- Add support for the SQLAlchemy ORM through *SQLAlchemyModelFactory* (issue #64, thanks to Romain Commandé)
- Add *factory.django.FileField* and *factory.django.ImageField* hooks for related Django model fields (issue #52)

Bugfix

- Properly handle non-integer pks in *DjangoModelFactory* (issue #57).
- Disable *RelatedFactory* generation when a specific value was passed (issue #62, thanks to Gabe Koscky)

Deprecation:

- Rename *RelatedFactory*'s `name` argument to `factory_related_name` (See issue #58)

2.0.2 (2013-04-16)

New:

- When `FACTORY_DJANGO_GET_OR_CREATE` is empty, use `Model.objects.create()` instead of `Model.objects.get_or_create`.

2.0.1 (2013-04-16)

New:

- Don't push `defaults` to `get_or_create` when `FACTORY_DJANGO_GET_OR_CREATE` is not set.

2.0.0 (2013-04-15)

New:

- Allow overriding the base factory class for *make_factory()* and friends.
- Add support for Python3 (Thanks to kmike and nkryptic)
- The default `type` for *Sequence* is now `int`
- Fields listed in `FACTORY_HIDDEN_ARGS` won't be passed to the associated class' constructor
- Add support for `get_or_create` in *DjangoModelFactory*, through `FACTORY_DJANGO_GET_OR_CREATE`.
- Add support for *fuzzy* attribute definitions.
- The `Sequence` counter can be overridden when calling a generating function
- Add *Dict* and *List* declarations (Closes issue #18).

Removed:

- Remove associated class discovery
- Remove `InfiniteIterator` and `infinite_iterator()`
- Remove `CircularSubFactory`
- Remove `extract_prefix` kwarg to post-generation hooks.
- Stop defaulting to Django's `Foo.objects.create()` when "creating" instances
- Remove STRATEGY_*
- Remove `set_building_function()` / `set_creation_function()`

1.3.0 (2013-03-11)

Warning: This version deprecates many magic or unexplicit features that will be removed in v2.0.0.

Please read the *Upgrading* section, then run your tests with `python -W default` to see all remaining warnings.

New

- **Global:**
  - Rewrite the whole documentation
  - Provide a dedicated `MogoFactory` subclass of `Factory`
- **The Factory class:**
  - Better creation/building customization hooks at `factory.Factory._build()` and `factory.Factory.create()`
  - Add support for passing non-kwarg parameters to a `Factory` wrapped class through `FACTORY_ARG_PARAMETERS`.
  - Keep the `FACTORY_FOR` attribute in `Factory` classes
- **Declarations:**
  - Allow `SubFactory` to solve circular dependencies between factories
  - Enhance `SelfAttribute` to handle "container" attribute fetching
  - Add a `getter` to `Iterator` declarations
  - A `Iterator` may be prevented from cycling by setting its `cycle` argument to `False`
  - Allow overriding default arguments in a `PostGenerationMethodCall` when generating an instance of the factory
  - An object created by a `DjangoModelFactory` will be saved again after `PostGeneration` hooks execution

Pending deprecation

The following features have been deprecated and will be removed in an upcoming release.

- **Declarations:**
  - `InfiniteIterator` is deprecated in favor of `Iterator`
  - `CircularSubFactory` is deprecated in favor of `SubFactory`
  - The `extract_prefix` argument to `post_generation()` is now deprecated
- **Factory:**
  - Usage of `set_creation_function()` and `set_building_function()` are now deprecated
  - Implicit associated class discovery is no longer supported, you must set the `FACTORY_FOR` attribute on all `Factory` subclasses

Upgrading

This version deprecates a few magic or undocumented features. All warnings will turn into errors starting from v2.0.0.

In order to upgrade client code, apply the following rules:

- Add a `FACTORY_FOR` attribute pointing to the target class to each *`Factory`*, instead of relying on automagic associated class discovery
- When using factory_boy for Django models, have each factory inherit from *`DjangoModelFactory`*
- Replace `factory.CircularSubFactory('some.module', 'Symbol')` with `factory.SubFactory('some.module.Symbol')`
- Replace `factory.InfiniteIterator(iterable)` with `factory.Iterator(iterable)`
- Replace `@factory.post_generation()` with `@factory.post_generation`
- Replace `factory.set_building_function(SomeFactory, building_function)` with an override of the *`_build()`* method of `SomeFactory`
- Replace `factory.set_creation_function(SomeFactory, creation_function)` with an override of the *`_create()`* method of `SomeFactory`

1.2.0 (2012-09-08)

New:

- Add `CircularSubFactory` to solve circular dependencies between factories

1.1.5 (2012-07-09)

Bugfix:

- Fix `PostGenerationDeclaration` and derived classes.

1.1.4 (2012-06-19)

New:

- Add *`use_strategy()`* decorator to override a *`Factory`*‘s default strategy
- Improve test running (tox, python2.6/2.7)
- Introduce *`PostGeneration`* and *`RelatedFactory`*

1.1.3 (2012-03-09)

Bugfix:

- Fix packaging rules

1.1.2 (2012-02-25)

New:

- Add *Iterator* and InfiniteIterator for *Factory* attribute declarations.
- Provide *generate()* and *simple_generate()*, that allow specifying the instantiation strategy directly. Also provides *generate_batch()* and *simple_generate_batch()*.

1.1.1 (2012-02-24)

New:

- Add *build_batch()*, *create_batch()* and *stub_batch()*, to instantiate factories in batch

1.1.0 (2012-02-24)

New:

- Improve the *SelfAttribute* syntax to fetch sub-attributes using the foo.bar syntax;
- Add ContainerAttribute to fetch attributes from the container of a *SubFactory*.
- Provide the *make_factory()* helper: MyClassFactory = make_factory(MyClass, x=3, y=4)
- Add *build()*, *create()*, *stub()* helpers

Bugfix:

- Allow classmethod/staticmethod on factories

Deprecation:

- Auto-discovery of FACTORY_FOR based on class name is now deprecated

1.0.4 (2011-12-21)

New:

- Improve the algorithm for populating a *Factory* attributes dict
- Add python setup.py test command to run the test suite
- Allow custom build functions
- Introduce MOGO_BUILD build function
- Add support for inheriting from multiple *Factory*
- Base *Factory* classes can now be declared abstract.
- Provide *DjangoModelFactory*, whose *Sequence* counter starts at the next free database id
- Introduce *SelfAttribute*, a shortcut for factory.LazyAttribute(lambda o: o.foo.bar.baz.

Bugfix:

- Handle nested *SubFactory*
- Share sequence counter between parent and subclasses

- Fix *SubFactory* / *Sequence* interferences

1.0.2 (2011-05-16)

New:

- Introduce *SubFactory*

1.0.1 (2011-05-13)

New:

- Allow *Factory* inheritance
- Improve handling of custom build/create functions

Bugfix:

- Fix concurrency between *LazyAttribute* and *Sequence*

1.0.0 (2010-08-22)

New:

- First version of factory_boy

Credits

- Initial version by Mark Sandstrom (2010)
- Developed by Raphaël Barrois since 2011

Ideas

This is a list of future features that may be incorporated into factory_boy:

- When a `Factory` is built or created, pass the calling context throughout the calling chain instead of custom solutions everywhere
- Define a proper set of rules for the support of third-party ORMs
- Properly evaluate nested declarations (e.g `factory.fuzzy.FuzzyDate(start_date=factory.SelfAttribute('since'))`)
- genindex
- modindex
- search

Python Module Index

Index