この記事は BeProud Advent Calendar 2018 7日目の記事ということにします。

公開日を分けるのがめんどくさいので 12/6 の基礎編に合わせて公開してます。事故ではありません

この 記事では 軽い応用から 基礎編だけではカバーしきれなかった少し特殊なケースを取り上げます。

少しだけ長いので、必要な部分だけ参照すれば良いと思います

備考

各セクションごとに プロジェクトの環境を固めた ZIP を用意したので手元で試したい方はご利用ください。

これらの環境を動かすために必要なライブラリが記述された requirements.txt が入っているので `pip install -rrequirements.txt` のようにしてください。 すべて同じライブラリで動作するのでインストールは一回で良いです。(venv推奨)

いずれの環境もマイグレーション適用前のものとなっています。

また、不要ファイルは削除したりしたので環境ごとに入っているファイルが異なります。

既存テーブルがある状態で初期マイグレーションをする

ZIP	apps-existing-table.zip

- 1.7 未満の Django から アップグレードした
- Django 以外のフレームワークから移行してきた

など様々な要因でテーブルの有無とマイグレーションの不整合は発生します。

ここでは、Django 1.6 (マイグレーションがないバージョン) 以前から 2.1 へ移行してきたという気持ちでやってみたいと思います。

1.6 で初期化	スキーマ

1.6 で初期化	スキーマ
$ ./manage.py syncdb Creating tables ... Creating table django_admin_log Creating table auth_permission Creating table auth_group_permissions Creating table auth_group Creating table auth_user_groups Creating table auth_user_user_permission Creating table auth_user Creating table django_content_type Creating table django_session Creating table products_category Creating table products_product Creating table products_price Creating table sales_sales Creating table sales_summary You just installed Django's auth system, Would you like to create one now? (yes/n Installing custom SQL ... Installing indexes ... Installed 0 object(s) from 0 fixture(s)	sqlite> .schema CREATE TABLE IF NOT EXISTS "django_adm "id" integer NOT NULL PRIMARY KEY, "action_time" datetime NOT NULL, "user_id" integer NOT NULL, "content_type_id" integer, "object_id" text, "object_repr" varchar(200) NOT NUL "action_flag" smallint unsigned NO "change_message" text NOT NULL); CREATE TABLE IF NOT EXISTS "auth_permi "id" integer NOT NULL PRIMARY KEY, "name" varchar(50) NOT NULL, "content_type_id" integer NOT NULL "codename" varchar(100) NOT NULL, UNIQUE ("content_type_id", "codena); CREATE TABLE IF NOT EXISTS "auth_group "id" integer NOT NULL PRIMARY KEY, "group_id" integer NOT NULL, "permission_id" integer NOT NULL R UNIQUE ("group_id", "permission_id); CREATE TABLE IF NOT EXISTS "auth_group "id" integer NOT NULL PRIMARY KEY, "name" varchar(80) NOT NULL UNIQUE); CREATE TABLE IF NOT EXISTS "auth_user_

2.1 に移行してきたのでマイグレーションを作って適用してみます。 (用意したzipファイルはここまでの操作が記録されています)

`./manage.py makemigrations products sales`	`./manage.py migrate`

`./manage.py makemigrations products sales`	`./manage.py migrate`

`$ ./manage.py makemigrations products sa`
`Migrations for 'products':`
`  products/migrations/0001_initial.py`
`    - Create model Category`
`    - Create model Price`
`    - Create model Product`
`    - Add field product to price`
`Migrations for 'sales':`
`  sales/migrations/0001_initial.py`
`    - Create model Sales`
`    - Create model Summary`
	`$ ./manage.py migrate`
	`Operations to perform:`
	`  Apply all migrations: admin, auth, c`
	`Running migrations:`
	`  Applying contenttypes.0001_initial..`
	`  File "/venv/lib/python3.7/site-packa`
	`    return self.cursor.execute(sql)`
	`  File "/venv/lib/python3.7/site-packa`
	`    return Database.Cursor.execute(sel`
	`sqlite3.OperationalError: table "djang`
	
	`The above exception was the direct cau`
	
	`Traceback (most recent call last):`
	`  File "./manage.py", line 10, in <mod`
	`    execute_from_command_line(sys.argv`
	`  File "/venv/lib/python3.7/site-packa`
	`    utility.execute()`
	`  File "/venv/lib/python3.7/site-packa`
	`    self.fetch_command(subcommand).run`
	`  File "/venv/lib/python3.7/site-packa`
	`    self.execute(*args, **cmd_options)`
	`  File "/venv/lib/python3.7/site-packa`
	`    output = self.handle(*args, **opti`
	`  File "/venv/lib/python3.7/site-packa`
	`    res = handle_func(*args, **kwargs)`
	`  File "/venv/lib/python3.7/site-packa`
	`    fake_initial=fake_initial,`
	`  File "/venv/lib/python3.7/site-packa`

続いて `--fake-initial` を使って適用してみます

`--fake-initial` 適用	スキーマの状態

--fake-initial 適用	スキーマの状態

$ ./manage.py migrate --fake-initial
Operations to perform:
  Apply all migrations: admin, auth, con
Running migrations:
  Applying contenttypes.0001_initial...
  Applying auth.0001_initial... FAKED
  Applying admin.0001_initial... FAKED
  Applying admin.0002_logentry_remove_au
  Applying admin.0003_logentry_add_actio
  Applying contenttypes.0002_remove_cont
  Applying auth.0002_alter_permission_na
  Applying auth.0003_alter_user_email_ma
  Applying auth.0004_alter_user_username
  Applying auth.0005_alter_user_last_log
  Applying auth.0006_require_contenttype
  Applying auth.0007_alter_validators_ad
  Applying auth.0008_alter_user_username
  Applying auth.0009_alter_user_last_nam
  Applying products.0001_initial... FAKE
  Applying sales.0001_initial... FAKED
  Applying sessions.0001_initial... FAKE
	sqlite> .schema
	CREATE TABLE IF NOT EXISTS "auth_group
	    "id" integer NOT NULL PRIMARY KEY,
	    "group_id" integer NOT NULL,
	    "permission_id" integer NOT NULL R
	    UNIQUE ("group_id", "permission_id
	);
	CREATE TABLE IF NOT EXISTS "auth_group
	    "id" integer NOT NULL PRIMARY KEY,
	    "name" varchar(80) NOT NULL UNIQUE
	);
	CREATE TABLE IF NOT EXISTS "auth_user_
	    "id" integer NOT NULL PRIMARY KEY,
	    "user_id" integer NOT NULL,
	    "group_id" integer NOT NULL REFERE
	    UNIQUE ("user_id", "group_id")
	);
	CREATE TABLE IF NOT EXISTS "auth_user_
	    "id" integer NOT NULL PRIMARY KEY,
	    "user_id" integer NOT NULL,
	    "permission_id" integer NOT NULL R
	    UNIQUE ("user_id", "permission_id"
	);
	CREATE TABLE IF NOT EXISTS "django_ses
	    "session_key" varchar(40) NOT NULL
	    "session_data" text NOT NULL,
	    "expire_date" datetime NOT NULL
	);
	CREATE TABLE IF NOT EXISTS "products_c

各アプリの初期(0001_initial.py) 以外のマイグレーションも無事適用されました。

現時点で auth/migrations/ には 10個のマイグレーションがありますが反映されているようです。 例えば auth_user.last_name の varchar length が 30 -> 150 になっています。

逆に --fake で適用してしまうと..

--fake 適用	スキーマの状態

--fake 適用	スキーマの状態

```
$ ./manage.py migrate --fake
Operations to perform:
  Apply all migrations: admin, auth, con
Running migrations:
  Applying contenttypes.0001_initial...
  Applying auth.0001_initial... FAKED
  Applying admin.0001_initial... FAKED
  Applying admin.0002_logentry_remove_au
  Applying admin.0003_logentry_add_actio
  Applying contenttypes.0002_remove_cont
  Applying auth.0002_alter_permission_na
  Applying auth.0003_alter_user_email_ma
  Applying auth.0004_alter_user_username
  Applying auth.0005_alter_user_last_log
  Applying auth.0006_require_contenttype
  Applying auth.0007_alter_validators_ad
  Applying auth.0008_alter_user_username
  Applying auth.0009_alter_user_last_nam
  Applying products.0001_initial... FAKE
  Applying sales.0001_initial... FAKED
  Applying sessions.0001_initial... FAKE
```

```
sqlite> .schema
CREATE TABLE IF NOT EXISTS "django_adm
    "id" integer NOT NULL PRIMARY KEY,
    "action_time" datetime NOT NULL,
    "user_id" integer NOT NULL,
    "content_type_id" integer,
    "object_id" text,
    "object_repr" varchar(200) NOT NUL
    "action_flag" smallint unsigned NO
    "change_message" text NOT NULL
);
CREATE TABLE IF NOT EXISTS "auth_permi
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(50) NOT NULL,
    "content_type_id" integer NOT NULL
    "codename" varchar(100) NOT NULL,
    UNIQUE ("content_type_id", "codena
);
CREATE TABLE IF NOT EXISTS "auth_group
    "id" integer NOT NULL PRIMARY KEY,
    "group_id" integer NOT NULL,
    "permission_id" integer NOT NULL R
    UNIQUE ("group_id", "permission_id
);
CREATE TABLE IF NOT EXISTS "auth_group
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(80) NOT NULL UNIQUE
);
CREATE TABLE IF NOT EXISTS "auth_user_
```

適用されるべきマイグレーションまで フェイク適用してしまったようです。

--fake-initial と --fake の使い分けには注意しましょうね。

備考

今回はシステムで使っているテーブルだけに恩恵がありましたが 今後 products , sales アプリにもマイグレーションが追加され、 自分以外のチームメンバーがそのマイグレーションを適用するときも、 --fake-initial は役に立ちます。

😇 NULLが許可されていないフィールドを追加

ZIP	apps-non-nullable-field.zip

null が許可されていないフィールドを追加しようとすると以下のような入力を求められることがあります。

```
You are trying to add a non-nullable field 'code' to category without a default; we can't d
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for t
 2) Quit, and let me add a default in models.py
Select an option:
```

これは null が許可されていない上、デフォルト値も指定されていないため、すでにレコード (rows) がある場合に 補完する値を Django が判断できないので指示を求められています。

それぞれ

- 1. 既存レコードに割り当てるデフォルト値を対話的に指定する
- 2. マイグレーションファイルを作るのを中断する (models.py に直接デフォルト値を指定してやりなおしてね)

と言った感じですね。

1 を選ぶ場合、 フィールドの方に沿った 値で適切なものを指定すればよいです。 また、値としてコーラブルなオブジェクト (例えば `django.utils.timezone`) を指定すると、 実行時の返却値がデフォルト値となります。

警告

これらのデフォルト値はレコードごとに設定することはできず、画一的に同じ値が設定されます。

別カラムの値を元にフィールドの値を設定したい場合は 一度 nullable なフィールドとして定義した後、データマイグレーション(後述) を行い、 nullable (`null=False`) を指定すればよいでしょう。

では、実際に 以下のようなカテゴリレコードがあるという状況でやってみます。

```
insert into products_category(id, name) values
  (1, 'alpaca'),
  (2, 'dog')
;

select * from products_category;
1|alpaca
2|dog
```

ではマイグレーションを作って適用してみます。 デフォルト値は timezone.now にします。

```
$ ./manage.py makemigrations products
You are trying to add a non-nullable field 'created_at' to category without a default; we c
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for t
 2) Quit, and let me add a default in models.py
Select an option: 1
Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so you can do e.g. timezone.n
```

```
Type 'exit' to exit this prompt
>>> timezone.now
Migrations for 'products':
  products/migrations/0002_category_created_at.py
    - Add field created_at to category

$ ./manage.py migrate products 0002
Operations to perform:
  Target specific migration: 0002_category_created_at, from products
Running migrations:
  Applying products.0002_category_created_at... OK
```

フィールドが追加され、デフォルト値も格納されたようです。

```
sqlite> .schema products_category
CREATE TABLE "products_category" (
  "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
  "name" varchar(30) NOT NULL,
  "created_at" datetime NOT NULL
);

sqlite> select * from products_category ;
1|alpaca|2018-12-05 09:47:37.577821
2|dog|2018-12-05 09:47:37.577821
```

😩 マイグレーションを取り消す

ZIP	apps-revert.zip

基礎編#migrate でも説明しましたが、 適用済マイグレーションを指定するとそれより一つ後のマイグレーションまで逆適用 つまり、取り消しが行われます。

0001 (先頭) まで取り消したい場合は zero を指定します。

--fake と併用すると履歴だけを消せます。

products/migrations/ 配下のマイグレーションをすべて適用した状態で

```
sqlite> select id, app, name from django_migrations;
1|products|0001_initial
2|products|0002_product_deleted_at

sqlite> .table
django_migrations  products_category  products_price     products_product
```

逆適用した結果を見てみましょう

`./manage.py migrate products zero`	`./manage.py migrate products zero --fake`

`./manage.py migrate products zero`	`./manage.py migrate products zero --fake`

Operations to perform:
  Unapply all migrations: products
Running migrations:
  Rendering model states... DONE
  Unapplying products.0002_product_delet
  Unapplying products.0001_initial... OK
	Operations to perform:
	  Unapply all migrations: products
	Running migrations:
	  Rendering model states... DONE
	  Unapplying products.0002_product_delet
	  Unapplying products.0001_initial... FA

sqlite> select id, app, name from django -- products のテーブルは全部消えてる sqlite> .table django_migrations	sqlite> select id, app, name from django -- テーブルは消えない！ sqlite> .table django_migrations products_category pr

😎 データマイグレーション

ZIP	apps-data-migration.zip

既存レコードの整理、初期レコードの追加、フィールドの変更によって スキーマではなくレコードを操作したくなることもあります。

これを行うのが前述した RunSQL や RunPython です。

RunSQL

RunSQL は単純に 第一引数で与えた 生SQL (DML, 例えば INSERT, UPDATE, DELETE 文) を実行するだけです。

第二引数は逆適用の SQLを指定します。省略できます。 形式は 第一引数と同じです。

まず、空のマイグレーションファイルを用意します.

```
$ ./manage.py makemigrations products --empty --name='manual'
Migrations for 'products':
  products/migrations/0002_manual.py
```

マイグレーションファイルはこんな感じになっております

```
from django.db import migrations

# カテゴリ追加
INSERT_CATEGORIES_SQL = '''
INSERT INTO products_category (name)
VALUES
```

```
  (%s),
  (%s),
  (%s)
;
'''

# すべての値を削除
DELETE_CATEGORIES_SQL = '''
DELETE FROM products_category;
'''

class Migration(migrations.Migration):

    dependencies = [
        ('products', '0001_initial'),
    ]

    operations = [
        migrations.RunSQL(
            [
                (INSERT_CATEGORIES_SQL, ('a', 'b', 'c',)),
                (INSERT_CATEGORIES_SQL, ('d', 'e', 'f',)),
            ],
```

第一引数 と 第二引数 の形式は同じと言いましたが、いきなり違っていますね..

実は RunSQL は 引数として文字列を受け取るとそのまま解釈して実行し、 タプルのリストとして受け取ると、タプルの中をパラメータとして SQL を組み立て、リストの要素数回実行します。

では順適用と逆適用をしてみましょう。

`./manage.py migrate products 0002`	**結果**

`$ ./manage.py migrate products 0002`
`Operations to perform:`
`  Apply all migrations: products`
`Running migrations:`
`  Applying products.0002_manual... OK`
	`sqlite> select * from products_category`
	`1|a`
	`2|b`
	`3|c`
	`4|d`
	`5|e`
	`6|f`

`./manage.py migrate products 0001`	**結果**

`$ ./manage.py migrate products 0001`
`Operations to perform:`
`  Target specific migration: 0001_initia`
`Running migrations:`
`  Rendering model states... DONE`
`  Unapplying products.0002_manual... OK`
	`sqlite> select * from products_category`
	`-- なし`

逆適用もできました。

備考

- django.core.exceptions.ImproperlyConfigured: The sqlparse package is required if you don't split your SQL statements manually. と出た場合は sqlparse をインストールしましょう

Running a .sql file after migrations in django - Stack Overflow

- 第二引数 (reverse_sql) を指定せずに逆適用をしようとすると `django.db.migrations.exceptions.IrreversibleError` が発生します。

警告

RunSQL 中で DDL (CREATE TABLE, ALTER 文など) を実行することはあまりおすすめできません。

これらの 操作内でスキーママイグレーションが行われても Django は認識できないのです。

どうしても手動で DDL を実行する場合、 `state_operations` 引数に Django に認識してほしい Operation をリスト形式 (operations 属性と同じ) で指定します。

少しわかりにくいかもしれないので不整合が起こる例をあげます。

- 1, RunSQL で `DROP TABLE products_price` を指定したマイグレーションを実行
    - `products_price` テーブルが消える
    - Django は `products_price` テーブルが消えたことを知らない
- 2, models.py からも `Price` モデルを削除し makemigrations する
    - Django は `Price` モデルを削除するための Operation (DeleteModel) を含むマイグレーションが自動生成される
- 3, 2 で作成したマイグレーションを実行
    - `products_price` を削除しようとするが 既に削除されているためエラーが発生する (不整合)

上記の場合は 下記のように指定すればOKです。

```
operations = [
    migrations.RunSQL("DROP TABLE products_price", state_operations=[
        migrations.DeleteModel(name='Price'),
    ]),
]
```

こんなことをやるのは冗長ですよね。「どうしても」というときを除きできるだけ Django に任せましょう

RunPython

先ほどと同様に 空のマイグレーションファイルを作成します。

```
$ ./manage.py makemigrations products --empty --name='manual'
Migrations for 'products':
  products/migrations/0003_manual.py
```

今回はカテゴリを全部大文字にするようなものを考えてみました。 (ただしquerysetでやるようなメリットはない..)

```
from django.db import migrations
from django.db.models import Func, F


def set_categories_uppercase(apps, schema_editor):
    Category = apps.get_model("products", "Category")
    Category.objects.all().update(
        name=Func(F('name'), function='UPPER')
    )

def set_categories_lowercase(apps, schema_editor):
    Category = apps.get_model("products", "Category")
    Category.objects.all().update(
        name=Func(F('name'), function='LOWER')
    )

class Migration(migrations.Migration):

    dependencies = [
        ('products', '0002_manual'),
    ]

    operations = [
        migrations.RunPython(set_categories_uppercase, set_categories_lowercase)
    ]
```

./manage.py migrate products 0003	**結果**

`$ ./manage.py migrate products 0003`
`Operations to perform:`
`  Apply all migrations: products`
`Running migrations:`
`  Applying products.0003_manual... OK`
	`sqlite> select * from products_category`
	`7|A`
	`8|B`
	`9|C`
	`10|D`
	`11|E`
	`12|F`

./manage.py migrate products 0002	**結果**

`$ ./manage.py migrate products 0002`
`Operations to perform:`
`  Target specific migration: 0002_manual`
`Running migrations:`
`  Rendering model states... DONE`
`  Unapplying products.0003_manual... OK`
	`sqlite> select * from products_category`
	`7|a`
	`8|b`
	`9|c`
	`10|d`
	`11|e`
	`12|f`

警告

これはメリットでもあり、デメリットでもあるんですが、 migrations.RunPython 関数内でレコードに変更が生じても登録されているシグナルは発火しません。

後処理などをシグナルで行っている場合、マイグレーションで同様の処理を実施してあげる必要があります。

備考

Django では 初期データの投入方法として fixture が推奨されてきましたが、 Django 1.8 で データマイグレーションが推奨されるようになりました。

- Providing initial data for models | Django documentation | Django 1.8
- Providing initial data for models | Django documentation | Django 2.1

ただ、マイグレーションによって自動投入されてほしくないようなデータは fixture を使う想定なんだと思います。

とはいえ、JSON で表すのに適していない大量データや、 別のデータに依存しているデータ、 規則性のあるデータはマイグレーションで作ったほうが良いかもしれません。

状況に応じて柔軟に使い分けていきましょう。

😯 Operation クラス をカスタマイズする

ZIP	apps-custom-operation.zip

Operation クラスを継承してカスタムした操作を作成できます。

私の想像力が乏しいため、カスタマイズしないとできないような例が思い浮かばなかったので、 カテゴリを追加するような操作を自分で作ってみましょう。 (RunSQL や RunPython でもできます)

前回と同様、空のマイグレーションファイルを用意します.

```
$ ./manage.py makemigrations products --empty --name='manual'
Migrations for 'products':
  products/migrations/0002_manual.py
```

```
from django.db import migrations
from django.db.migrations.operations.base import Operation


class CreateCategory(Operation):
    reversible = True

    def __init__(self, name):
        self.name = name

    def state_forwards(self, app_label, state):
        pass

    def database_forwards(self, app_label, schema_editor, from_state, to_state):
```

```
        schema_editor.execute("INSERT INTO products_category (name) VALUES (%s);", (self.

    def database_backwards(self, app_label, schema_editor, from_state, to_state):
        schema_editor.execute("DELETE FROM products_category WHERE name=%s;", (self.name,

    def describe(self):
        return "Creates category %s" % self.name


class Migration(migrations.Migration):

    dependencies = [
        ('products', '0001_initial'),
    ]
```

重要なのは以下のメソッドです。

- database_forwards: 実行する操作を記述
- database_backwards: 実行した操作を打ち消すための操作を記述
- reversible: `True` であれば backwards 可能

今回の カスタムオペレーションを使うためには `operations` 属性に指定するだけです。

では適用してみます。

`./manage.py migrate products 0002`	結果

`$ ./manage.py migrate products`
`Operations to perform:`
`  Apply all migrations: products`
`Running migrations:`
`  Applying products.0001_initial... OK`
`  Applying products.0002_manual... OK`
	`sqlite> select * from products_category`
	`1|alpaca`
	`2|dog`

つづいて逆適用。

`./manage.py migrate products 0001`	結果

`$ ./manage.py migrate products 0001`
`Operations to perform:`
`  Target specific migration: 0001_initia`
`Running migrations:`
`  Rendering model states... DONE`
`  Unapplying products.0002_manual... OK`
	`sqlite> select * from products_category`
	`-- なし`

うまく動いたみたいです。

😠 既存データに合わないスキーママイグレーション

ZIP	apps-unmatch.zip

makemigrations コマンドは DBの状態を考慮せずに マイグレーションファイルを作ると言いました。

そのため、スキーマ的にはマイグレーション可能でも実際に適用してみると、 既存データが変更後の 型 や 制約 に合わずにマイグレーションが失敗することがあります。

色んなパターンがあるので網羅は難しいんですが、今回は型の変更について説明します。

以下のシナリオでやってみます。

- Category モデルの id列 を UUIDField から Autofield (integer) フィールドに変更する
    - id 列は 主キー
    - キャスト不可能
- created の順に連番を振りなおす
- Product.category は Category.id を外部参照している
    - レコードは 1件以上存在する
- 既にテーブルは作成されている

作成	適用

`./manage.py makemigrations products`
`Migrations for 'products':`
`  products/migrations/0001_initial.py`
`  - Create model Category`
`  - Create model Price`
`  - Create model Product`
`  - Add field product to price`
	`$ ./manage.py migrate products 0001`
	`Operations to perform:`
	`  Apply all migrations: products`
	`Running migrations:`
	`  Applying products.0001_initial... OK`

- 既に 次のようなレコードが入っている

```
insert into products_category(id, name, created_at) values
  ('5fe87b34-1d84-4dd2-bfdd-4c9d275bc5a2', 'alpaca', '2018-01-01'),
  ('c5a63139-3470-4e07-9668-d2c9edc478bd', 'dog', '2018-01-02'),
  ('4ca1bcbd-64b2-4f13-a445-6b096ee655aa', 'cat', '2018-01-03'),
  ('e12db291-875e-4a2b-aedd-5423fd8eb18c', 'capibara', '2018-01-01')
;

insert into products_product(name, category_id, created_at, updated_at) values
  ('アルパカの置物', '5fe87b34-1d84-4dd2-bfdd-4c9d275bc5a2', '2018-02-01', '2018-02-01')
;
```

この状態でマイグレーションを普通に適用しようとすると次のようなエラーになります。

```
$ ./manage.py migrate
Operations to perform:
  Apply all migrations: accounts, admin, auth, contenttypes, products, sessions
Running migrations:
  Applying products.0003_uuid_to_integer...Traceback (most recent call last):
  File "/venv/lib/python3.7/site-packages/django/db/backends/utils.py", line 85, in _execut
    return self.cursor.execute(sql, params)
  File "/venv/lib/python3.7/site-packages/django/db/backends/sqlite3/base.py", line 296, in
    return Database.Cursor.execute(self, query, params)
sqlite3.IntegrityError: datatype mismatch
```

では、情報が失われないように気をつけながら データとスキーマを変更していきます。

Category モデルに入れ替え用のフィールドを定義

- `id2 = models.IntegerField(default=0)`

作成	マイグレーションファイル

`$ ./manage.py makemigrations products`
`Migrations for 'products':`
`  products/migrations/0002_id2.py`
`  - Add field id2 to category`
	0002_id2.py
	`from django.db import migra`
	
	`class Migration(migrations.Migration):`
	
	`  dependencies = [`
	`  ('products', '0001_initial'),`
	`  ]`
	
	`  operations = [`
	`  migrations.AddField(`
	`  model_name='category',`
	`  name='id2',`
	`  field=models.IntegerField(`
	`  ),`
	`  ]`

入れ替え要のフィールド に 一意な連番を振る

- 空のマイグレーションファイルを作る
  - `./manage.py makemigrations products --empty --name="put_serial_number"`
- 連番で更新するような Operation (RunSQL) をマイグレーションに追加する
  - 今使ってる SQLite3 のバージョンでは Window関数が使えないのでこんな感じで..

作成	マイグレーションファイル

作成	マイグレーションファイル

./manage.py makemigrations products --
Migrations for 'products':
  products/migrations/0003_put_serial_
	from django.db
	
	SQL = """
	UPDATE products_category AS a
	SET id2 = (
	  SELECT COUNT(1) + 1
	  FROM products_category AS b
	  WHERE
	  -- 登録日時が同じ場合に重複してしまう
	  (a.created_at = b.created_at AND a
	  -- それ以外の場合は 登録日時でソート
	  OR a.created_at > b.created_at
	);
	"""
	
	class Migration(migrations.Migration):
	
	  dependencies = [
	  ('products', '0002_id2'),
	  ]
	
	  operations = [
	  migrations.RunSQL(SQL)
	  ]

0003_put_serial_number.py

Category を外部参照しているモデルに 入れ替え用のフィールドを追加

- 今回は Product モデルのみ
  - 変更するのが主キーでなければこの操作は不要
  - 外部参照しているモデルが他にある場合、その数だけ繰り返す
- `category_id2 = models.IntegerField(default=0)`
  - 追加するフィールドは Category.id2 の型と一致すること

作成	マイグレーションファイル

作成	マイグレーションファイル

`$ ./manage.py makemigrations products`
`Migrations for 'products':`
`  products/migrations/0004_category_id`
`    - Add field category_id2 to produc`
	0004_category_id2.py
	`from django.db impo`
	
	`class Migration(migrations.Migration):`
	
	`    dependencies = [`
	`        ('products', '0003_put_serial_`
	`    ]`
	
	`    operations = [`
	`        migrations.AddField(`
	`            model_name='product',`
	`            name='category_id2',`
	`            field=models.IntegerField(`
	`        ),`
	`    ]`

Category を外部参照しているモデルの入れ替え用フィールドに 現在の主キーに紐づく連番を格納する

- 今回は Product モデルのみ
    - 変更するのが主キーでなければこの操作は不要
    - 外部参照しているモデルが他にある場合、その数だけ繰り返す
- 空のマイグレーションファイルを作る
- Product.category_id に紐づく Category.id2 を Product.category_id2 に入れるような UPDATE 文を実行する Operation (RunSQL) を空マイグレーションに追加する
- 変更するのが主キーでない場合は不要

作成	マイグレーションファイル

作成	マイグレーションファイル

`$ ./manage.py makemigrations products`
`Migrations for 'products':`
`  products/migrations/0005_put_new_pk.`
	`from django.db impor` 0005_put_new_pk.py
	
	`SQL = """`
	`UPDATE products_product AS p`
	`SET category_id2 = (`
	`  SELECT id2`
	`  FROM products_category AS c`
	`  WHERE p.category_id = c.id`
	`)`
	`;`
	`"""`
	
	`class Migration(migrations.Migration):`
	
	`  dependencies = [`
	`  ('products', '0004_category_id`
	`  ]`
	
	`  operations = [`
	`  migrations.RunSQL(SQL),`
	`  ]`

Category を外部参照しているフィールドを削除する

- 今回は Product モデルのみ
  - 変更するのが主キーでなければこの操作は不要
  - 外部参照しているモデルが他にある場合、その数だけ繰り返す
- `# category = models.ForeignKey(Category, on_delete=models.CASCADE)`
  - あとから復活させるためコメントアウトに留める

作成	マイグレーションファイル

`$ ./manage.py makemigrations products`
`Migrations for 'products':`
`  products/migrations/0006_del_categor`
`  - Remove field category from produ`
	`from django.db im` 0006_del_category_id.py
	
	`class Migration(migrations.Migration):`
	
	`  dependencies = [`
	`  ('products', '0005_put_new_pk'`
	`  ]`
	
	`  operations = [`
	`  migrations.RemoveField(`
	`  model_name='product',`
	`  name='category',`
	`  ),`
	`  ]`

Category の入れ替え用フィールドを主キーにする

- id2 フィールドを AutoField に変更する (主キーの指定も必要)
    - `id2 = models.AutoField(primary_key=True)`
- id フィールドを削除する
    - `# id = models.UUIDField(primary_key=True, default=uuid.uuid4)`
        - 今回は差分をわかりやすくするためにコメントアウト

作成	マイグレーションファイル

`$ ./manage.py makemigrations products`
`Migrations for 'products':`
`  products/migrations/0007_id2_primary`
`  - Remove field id from category`
`  - Alter field id2 on category`
	0007_id2_primary_key.py
	`from django.db im`
	
	`class Migration(migrations.Migration):`
	
	`  dependencies = [`
	`  ('products', '0006_del_categor`
	`  ]`
	
	`  operations = [`
	`  migrations.RemoveField(`
	`  model_name='category',`
	`  name='id',`
	`  ),`
	`  migrations.AlterField(`
	`  model_name='category',`
	`  name='id2',`
	`  field=models.AutoField(pri`
	`  ),`
	`  ]`

Category の入れ替え用フィールドを id にする

- id2 のフィールド名を id に変更
    - `id = models.AutoField(primary_key=True)`
    - (フィールドを消してしまうと 削除して追加 になるので注意)

作成	マイグレーションファイル

作成	マイグレーションファイル

`$ ./manage.py makemigrations products`
`Did you rename category.id2 to categor`
`Migrations for 'products':`
`  products/migrations/0008_id2_to_id.p`
`    - Rename field id2 on category to`
	0008_id2_to_id.py
	`from django.db import migrations`
	
	`class Migration(migrations.Migration):`
	
	`    dependencies = [`
	`        ('products', '0007_id2_primary`
	`    ]`
	
	`    operations = [`
	`        migrations.RenameField(`
	`            model_name='category',`
	`            old_name='id2',`
	`            new_name='id',`
	`        ),`
	`    ]`

Product.category_id2 から Product.category_id に変更

- Category を外部参照しているモデルの入れ替え用フィールドを リネームする
- Product.category_id2 のフィールド名を category_id に変更する
    - `category_id2 = models.IntegerField(default=0)`

作成	マイグレーションファイル

`$ ./manage.py makemigrations products`
`Did you rename product.category_id2 to`
`Migrations for 'products':`
`  products/migrations/0009_category_id`
`    - Rename field category_id2 on pro`
	0009_category_id2_to_category_id.py
	`from django.db import migrations`
	
	`class Migration(migrations.Migration):`
	
	`    dependencies = [`
	`        ('products', '0008_id2_to_id')`
	`    ]`
	
	`    operations = [`
	`        migrations.RenameField(`
	`            model_name='product',`
	`            old_name='category_id2',`
	`            new_name='category_id',`
	`        ),`
	`    ]`

Product.category_id を ForeignKey に変更する

- Product.category_id のフィールド を削除
    - `# category_id = models.IntegerField(default=0)`
- Product.category を ForeignKey で定義する

    - `category = models.ForeignKey(Category, on_delete=models.CASCADE)`
        - 先程コメントアウトしたものを外すだけでよい
- 自動で ForeignKey にするのは無理なので、手動で空のマイグレーションファイルを作る
    - `./manage.py makemigrations products --empty --name="category_id_to_foreignkey"`
    - migrations.AlterField を使ってフィールドを変更

作成	マイグレーションファイル

`$ ./manage.py makemigrations products`
`Migrations for 'products':`
`  products/migrations/0010_category_id`
	0010_category_id_to_foreignkey.py
	`from django.db import migrations, models`
	
	`class Migration(migrations.Migration):`
	
	`  dependencies = [`
	`  ('products', '0009_category_id`
	`  ]`
	
	`  operations = [`
	`  migrations.AlterField(`
	`  model_name='product',`
	`  name='category_id',`
	`  field=models.ForeignKey(on_delete=models.CASCADE, to='products.Category'),`
	`  ),`
	`  ]`

実行する

これを実行すると

```
$ ./manage.py migrate products
Operations to perform:
  Apply all migrations: products
Running migrations:
  Applying products.0002_id2... OK
  Applying products.0003_put_serial_number... OK
  Applying products.0004_category_id2... OK
  Applying products.0005_put_new_pk... OK
  Applying products.0006_del_category_id... OK
  Applying products.0007_id2_primary_key... OK
  Applying products.0008_id2_to_id... OK
  Applying products.0009_category_id2_to_category_id... OK
  Applying products.0010_category_id_to_foreignkey... OK
```

migrate は無事終わったようですが、

さて、どうなったかな。

Before	After

Before	After
sqlite> select * from products_categor 5fe87b34-1d84-4dd2-bfdd-4c9d275bc5a2|a c5a63139-3470-4e07-9668-d2c9edc478bd|c 4ca1bcbd-64b2-4f13-a445-6b096ee655aa|c e12db291-875e-4a2b-aedd-5423fd8eb18c|c sqlite> select * from products_product 1|アルパカの置物|2018-02-01|2018-02-01|	sqlite> select * from products_categor alpaca|2018-01-01|1 capibara|2018-01-01|2 dog|2018-01-02|3 cat|2018-01-03|4 sqlite> select * from products_product 1|アルパカの置物|2018-02-01|2018-02-01|

無事にデータ移行できたようです。

警告

別の口から処理を受け付けていると連番がユニークにならずマイグレーションが失敗する可能性があります。

マイグレーションを実行するときは、 できるだけメンテナンス時間などを設けて外部と遮断した上で適用することをおすすめします。

備考

キャスト可能な型の場合はそのまま格納可能です。

AutoField (integer) から UUIDField に変えた場合、 SQLite3では UUID は `char` 型 なので 単純に文字列キャストされて格納されます。

```
sqlite> .schema products_category
CREATE TABLE IF NOT EXISTS "products_category" ("name" varchar(30) NOT NULL, "id" char(32)

sqlite> select * from products_category ;
test|1
```

PostgreSQL のように異なる型として認識される場合は先程と同様にデータを調整してあげる必要があります。

今更ですが、主キーをキャスト不可能な型に変更するとか 正気の沙汰ではないので普通はやめましょう。

😌 同じ親を持つマイグレーション

ZIP	apps-multiple-heads.zip

あなたが一人で開発をしていて、一つのブランチ(Gitなど) で作業している場合、 このようなケースに遭遇することはめったにないと思いますが、 複数人で開発していると同じアプリ内の同じ親を参照するマイグレーションができあがってしまうことが稀によくあります。

Mercurial (VCS) で言うところの いわゆる マルチプルヘッド (双頭) と呼ばれる、先端が二又以上に分かれた状態です。

Django 的にもこの状態は好ましくありません。

実際に試してみましょう。 今回は accounts アプリの 0004, 0005 がいずれも 0003 を参照しているという状態、 つまり末端が 2又に分かれています。

accounts/migrations/0004_dummy.py

```
from django.db import migrations


class Migration(migrations.Migration):

    dependencies = [
        ('accounts', '0003_dummy'),
    ]

    operations = [
    ]
```

accounts/migrations/0005_dummy.py

```
from django.db import migrations


class Migration(migrations.Migration):

    dependencies = [
        ('accounts', '0003_dummy'),
    ]

    operations = [
    ]
```

この状態でマイグレーションを適用すると `CommandError: Conflicting migrations detected; multiple leaf nodes in the migration graph: (0005_dummy, 0004_dummy in accounts). To fix them run 'python manage.py makemigrations --merge'` というエラーが発生します。

エラーメッセージでも言っている通り これらのファイルを一つに束ねるのが makemigrations の `--merge` オプションです。

備考

本来、上記のように連続した番号で 参照する親が重複するということはまず発生しませんが、わかりやすさのためこのようにしています。

```
$ ./manage.py makemigrations accounts --merge --name='merged'
Merging accounts
  Branch 0004_dummy
  Branch 0005_dummy

Merging will only work if the operations printed above do not conflict
with each other (working on different fields or models)
Do you want to merge these migration branches? [y/N] y

Created new merge migration apps-multiple-heads/accounts/migrations/0006_merged.py
```

これにより 以下のような 0006 が作られ、今回は無事適用できたようです。

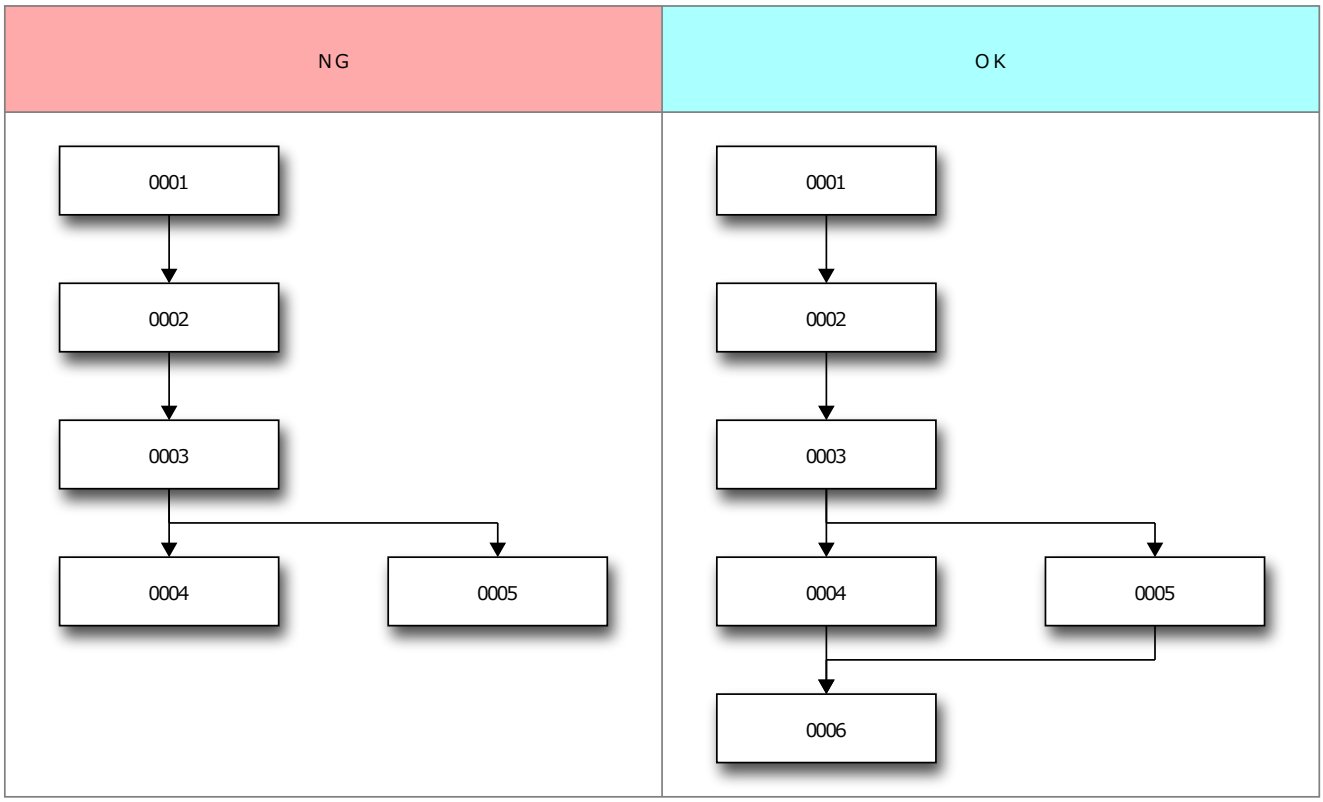
accounts/migrations/0006_dummy.py

適用

accounts/migrations/0006_dummy.py	適用

from django.db import migrations
 	
class Migration(migrations.Migration):
 	
    dependencies = [
        ('accounts', '0004_dummy'),
        ('accounts', '0005_dummy'),
    ]
 	
    operations = [
    ]
	$ ./manage.py migrate accounts
	Operations to perform:
	  Apply all migrations: accounts
	Running migrations:
	  Applying accounts.0001_initial... OK
	  Applying accounts.0002_dummy... OK
	  Applying accounts.0003_dummy... OK
	  Applying accounts.0005_dummy... OK
	  Applying accounts.0004_dummy... OK
	  Applying accounts.0006_merged... OK

このことから 先端さえ収束していれば Django 的にはOKということになりますね。



上記の図でいうと

0004, 0005 までで終わっている場合は アウト ですが、 0006 で束ねている状態は セーフ なのです。

備考

--merge は 先端が分岐してしまったマイグレーションを一つに束ねるだけのオプションであり、 squashmigrations のように複数のマイグレーションを意味のある 1 つのマイグレーションに 統合する機能では **ありません**

その心はおそらく、実行順の制御のためではなく、 次回マイグレーションを自動生成する場合に終端がわからないと Dependencies に指定するマイグレーションが一意に特定できないからだと思います。

分岐している部分のマイグレーション(0004, 0005)は 番号の順番とは関係なく適用されることがわかります。

重複したマイグレーションの順番を制御したい場合、 マイグレーションファイルの依存関係を自分で調整してあげましょう。 (ローカルDBへ既に適用されていることがほとんどなので、未適用のDBとは順番のズレが生じる可能性があります)

😱 マイグレーションが10000回を突破する

ZIP	apps-exceed.zip

番号が 0パディングの 4桁なので 10000 を超えても大丈夫なのかなーと心配する 僕みたいなひねくれた方がいるかもしれません。

コードを見てもマイグレーション数を制限している部分はありませんが、 一応検証してみました。

結論から言うと特に限界はなく、 10000超えても作成、適用はできます。

```
$ ./manage.py makemigrations accounts --empty --name='dummy'
Migrations for 'accounts':
  accounts/migrations/10000_dummy.py

$ ./manage.py makemigrations accounts --empty --name='dummy'
Migrations for 'accounts':
  accounts/migrations/10001_dummy.py

$ ./manage.py migrate accounts
Operations to perform:
  Apply all migrations: accounts
Running migrations:
  Applying accounts.0001_initial... OK
  Applying accounts.9999_dummy... OK
  Applying accounts.10000_dummy... OK
  Applying accounts.10001_dummy... OK
```

警告

今回は ファイル名を適当に調整して 9999 以降のマイグレーションを作りましたが、 実際に 9999 件ある状態で試したところ `makemigrations` だけで 時間(5分くらい)がかかって、

```
RuntimeWarning: Maximum recursion depth exceeded while generating migration graph, falling
```

みたいな警告が出ます。

ある程度溜まったら `squashmigrations` するのがおすすめです。

😅 複数のフィールドをリネームする

ZIP	apps-rename-fields.zip

一つのモデルにおいてフィールドの追加と削除を同時に検知した場合、 Django は フィールドがリネームされたと推測し、対話的に 紐づけてよいか確認してきます。賢いですね。

一つの場合は `y (yes)` を選んで終わりですが これが 複数あった場合について考えてみましょう。

特に難しいわけではないですが、初学者にとっては戸惑うポイントかも知れないので一応やっておきます。

たとえば、 Price というモデルのフィールドを同時に以下のように変更し

- `effective_date_from` -> `effective_date_start`
- `effective_date_to` -> `effective_date_end`

makemigrations をします。

すると すべての組み合わせについてヒモ付が正しいか確認してくるので、 `y/N` で教えてあげます。

```
$ ./manage.py makemigrations products --name='rename_fields'
Did you rename price.effective_date_from to price.effective_date_end (a DateTimeField)? [y/
Did you rename price.effective_date_to to price.effective_date_end (a DateTimeField)? [y/N]
Did you rename price.effective_date_from to price.effective_date_start (a DateTimeField)? [
Migrations for 'products':
  products/migrations/0002_rename_fields.py
    - Rename field effective_date_to on price to effective_date_end
    - Rename field effective_date_from on price to effective_date_start
```

備考

解説

- Did you rename price.effective_date_from to price.effective_date_end (a DateTimeField)? [y/N] n
  - `effective_date_from` から `effective_date_end` にリネームするけど正しい？と聞いてくるので `No`
  - `No` を選択すると このリネームはスキップされます
- Did you rename price.effective_date_to to price.effective_date_end (a DateTimeField)? [y/N] y
  - `effective_date_to` から `effective_date_end` にリネームするけど正しい？と聞いてくるので `Yes`
- Did you rename price.effective_date_from to price.effective_date_start (a DateTimeField)? [y/N] y

- effective_date_from から effective_date_start にリネームするけど正しい？と聞いてくるので Yes

でこんな感じのマイグレーションが作られます。 ZIPファイルはここまでの操作が記録されています。

適用してみます。

0002_rename_fields.py

```
from django.db import migrations


class Migration(migrations.Migration):

    dependencies = [
        ('products', '0001_initial'),
    ]

    operations = [
        migrations.RenameField(
            model_name='price',
            old_name='effective_date_to'
            new_name='effective_date_end
        ),
        migrations.RenameField(
            model_name='price',
            old_name='effective_date_fro
            new_name='effective_date_sta
        ),
    ]
```

適用

```
$ ./manage.py migrate products
Operations to perform:
  Apply all migrations: products
Running migrations:
  Applying products.0001_initial... OK
  Applying products.0002_rename_fields..
```

適用後に sales_price のスキーマを比較してみましょう

Before

```
sqlite> .schema products_price
CREATE TABLE IF NOT EXISTS "products_pri
  "id" integer NOT NULL PRIMARY KEY AUTO
  "price" integer NOT NULL,
  "effective_date_from" datetime NULL,
  "effective_date_to" datetime NULL,
  "product_id" integer NOT NULL REFERENC
CREATE INDEX "products_price_product_id_
```

After

```
sqlite> .schema products_price
CREATE TABLE IF NOT EXISTS "products_pri
  "id" integer NOT NULL PRIMARY KEY AUTO
  "price" integer NOT NULL,
  "effective_date_end" datetime NULL,
  "product_id" integer NOT NULL REFERENC
  "effective_date_start" datetime NULL);
CREATE INDEX "products_price_product_id_
```

ちゃんとリネームされているようです。

警告

対話で紐づけ関係が解決できない場合、「フィールドを一旦削除し、新規フィールドを追加する」 という挙動になるので注意してください。

本当にフィールドを削除して別のフィールドを追加したいこともあるでしょうから、 この挙動が一概に誤りとは言えません。

不安な方は「削除のマイグレーション」と「追加のマイグレーション」を分けて作成するというのも一つの手です。

ビューをモデルとして登録する

ZIP	apps-view.zip

VIEW を モデルとして登録したいことがあるかもしれません。 (Django の View ではないですよ)

単純に同じテーブル名とフィールドを定義すればよいのですが、 マイグレーション対象として管理したくないので `managed = False` を指定します。

これを使ったモデルは次のように定義できます。

```
class DoubleCategory(models.Model):
    id = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=255)

    class Meta:
        db_table = 'double_category'
        managed = False
```

特に良い例が思い浮かばなかったので category を 2 つつなげるだけの単純な VIEWを想定します。

作成

```
$ ./manage.py makemigrations products --
Migrations for 'products':
  products/migrations/0002_double_catego
    - Create model DoubleCategory
```

適用

```
$ ./manage.py migrate products
Operations to perform:
  Apply all migrations: products
Running migrations:
  Applying products.0001_initial... OK
  Applying products.0002_double_category
```

期待通り、テーブルはできてませんね

```
sqlite> .schema double_category
-- なし
```

必要に応じて 適当な VIEW を定義します。 (管理されないのでVIEWはあってもなくてもいいです)

```
sqlite> CREATE VIEW double_category AS SELECT id, name || ' ' || name AS name FROM products
sqlite> INSERT INTO products_category (name) VALUES ('a'), ('b'), ('c');
sqlite> select * from double_category ;
1|a a
2|b b
3|c c
```

最後にインタラクティブシェルから使ってみます。

```
>>> from products.models import DoubleCategory
>>> DoubleCategory.objects.values()
<QuerySet [{'id': 1, 'name': 'a a'}, {'id': 2, 'name': 'b b'}, {'id': 3, 'name': 'c c'}]>
```

はい、できました。

備考

今回は VIEW を使う例として `managed = False` を定義しましたが、 他には別のシステムで使っているテーブルを Django から参照したい(ただ、管理はしたくない) という場合も 同様に対処できます。

😛 ファクトリで生成した関数をモデルに指定する

ZIP	apps-factory.zip

Django の makemigrations は モデルの変更を検知してマイグレーションを作成するわけですが、 いくつかのフィールドでは 関数を引数として受け取れます。

具体的には `default` や FileField の `upload_to` ですが、 ここにファクトリ関数を指定しようとすると一苦労なのです。

まず、ファクトリ関数 (クロージャ機能を用いて関数等のオブジェクトを生成する関数) についてわからない方もいると思うので簡単な例をあげます。 ID (UUID) から 自動的に ファイルの保存パスを決定するような 関数を作成する ファクトリを作ります。

以下のように ユーザレコード の ID(UUID) を 4桁ずつのディレクトリに区切ったパスを生成する関数を作ってみます。 (この時点では ファクトリではないです)

```
import os
import unicodedata

def make_user_path(instance, filename):
    prefix = 'user/'
    path = [prefix.strip('/')]
    path += [instance.id.hex[i:i+4] for i in range(0, 32, 4)]
    path += [unicodedata.normalize('NFC', filename)]
    return os.path.join(*path)
```

これを実行すると期待したようなパスを得られることがわかります。

```
>>> from accounts.models import User
>>> u = User.objects.create(email='test@example.com', nickname='tester', age=75)
>>> make_user_path(u, 'test.txt')
>>> 'user/64bb/fa2e/36fc/4faa/aa2b/3367/4153/7c9e/test.txt'
```

このまま upload_to 引数に指定してもよいのですが、 同じ生成ルールで 他テーブルの添付ファイルも配置したいということになりました。 ただ、出力先の ディレクトリ (prefix) は任意のものを指定したいのです。

ここでファクトリ関数の出番です。 やることは割と単純で 先程の関数 を別の関数でラップして返却するイメージです。

prefix は外側の関数から渡してあげれば内側の関数でも参照でき、出力された関数では 常に prefix が固定されているというわけです。変数を閉じ込めていると表現することもあります。

```
import os
import unicodedata

def make_path_factory(prefix):
    def make_path(instance, filename):
        path = [prefix.strip('/')]
        path += [instance.id.hex[i:i+4] for i in range(0, 32, 4)]
        path += [unicodedata.normalize('NFC', filename)]
        return os.path.join(*path)
    return make_path
```

一旦関数を出力してから先ほどと同様に使ってみましょう。今回は avatar に変えてみました。

```
>>> make_user_path = make_path_factory('avatar')
>>> make_user_path(u, 'test.txt')
'avatar/64bb/fa2e/36fc/4faa/aa2b/3367/4153/7c9e/test.txt'
```

これですべてのモデルの添付ファイルを別々のディレクトリに出力することができる！

と思ってしまいそうですが、そう簡単には物事は運ばないようです。

試しに、このファクトリをモデルに指定してマイグレーションをしてみましょう。

models.py	適用

models.py	適用

import os
import unicodedata
 	
from django.db import models
from django.contrib.auth.models import (
    BaseUserManager, AbstractBaseUser
)
 	
def make_path_factory(prefix):
    def make_path(instance, filename):
        path = [prefix.strip('/')]
        path += [instance.id.hex[i:i+4]
        path += [unicodedata.normalize('
        return os.path.join(*path)
    return make_path
 	
make_path = make_path_factory('user/')
 	
class User(AbstractBaseUser):
    USERNAME_FIELD = 'email'
 	
    email = models.EmailField(max_length
    nickname = models.CharField(max_leng
    age = models.IntegerField(null=True)
    file = models.FileField(upload_to=ma
	./manage.py makemigrations accounts
	Migrations for 'accounts':
	  accounts/migrations/0001_initial.py
	    - Create model User
	Traceback (most recent call last):
	  File "./manage.py", line 15, in <mod
	    execute_from_command_line(sys.argv
	  File "/venv/lib/python3.7/site-packa
	    utility.execute()
	  File "/venv/lib/python3.7/site-packa
	    self.fetch_command(subcommand).run
	  File "/venv/lib/python3.7/site-packa
	    self.execute(*args, **cmd_options)
	  File "/venv/lib/python3.7/site-packa
	    output = self.handle(*args, **opti
	  File "/venv/lib/python3.7/site-packa
	    res = handle_func(*args, **kwargs)
	  File "/venv/lib/python3.7/site-packa
	    self.write_migration_files(changes
	  File "/venv/lib/python3.7/site-packa
	    migration_string = writer.as_strin
	  File "/venv/lib/python3.7/site-packa
	    operation_string, operation_import
	  File "/venv/lib/python3.7/site-packa
	    _write(arg_name, arg_value)
	  File "/venv/lib/python3.7/site-packa
	    arg_string, arg_imports = Migratio
	  File "/venv/lib/python3.7/site-packa
	    return serializer_factory(value).s

期待した関数は `accounts.models` には定義されていないと言われているようです。

関数名を合わせているのになぜでしょうか。 実はファクトリで作られた関数名 (Python3 の場合は `__qualname__`) は make_path_factory.<locals>.make_path なので 名前が合わないようです。

もちろん、これを手動で合わせてあげればいいんですが、 こういうメタ情報はできれば自分で触りたくありませんね。

Python にはこういった関数から返却された関数にメタ情報を引き継ぐための関数があります。 そうです。主にデコレータに使う functools.wraps ですね。

今回は functools.wraps を使ったデコレータとして再定義してあげましょう。 (他にいいやり方がある方は教えてください)

ついでに Group モデルの添付ファイルパスも作れるようにしてみます。

models.py	作成＆適用

models.py	作成＆適用

import os
import functools
import uuid
import unicodedata
 	
from django.db import models
from django.contrib.auth.models import
    BaseUserManager, AbstractBaseUser
)
 	
def make_path_factory(prefix):
    def wrapper(f):
        @functools.wraps(f)
        def unique_path(instance, file
            path = [prefix.strip('/')]
            path += [instance.id.hex[i
            path += [unicodedata.norma
            return os.path.join(*path)
        return unique_path
    return wrapper
 	
@make_path_factory('user')
def make_user_path():
    """ユーザIDを元に添付ファイルを保存す
 	
@make_path_factory('group')
	$ ./manage.py makemigrations accounts
	Migrations for 'accounts':
	  accounts/migrations/0001_initial.py
	    - Create model User
	    - Create model Group
	
	$ ./manage.py migrate accounts
	Operations to perform:
	  Apply all migrations: accounts
	Running migrations:
	  Applying accounts.0001_initial... OK

最後にこのモデルを使ってレコードを作って 期待通りのパスにファイルが吐かれるか確かめてみます。

レコードを作ってみる	ファイルが配置される

レコードを作ってみる	ファイルが配置される

>>> from django.core.files.base import C
>>> from accounts.models import User, Gr
 	
>>> f = ContentFile(b'file content', 'te
 	
>>> u = User.objects.create(email='test@
>>> u.id
UUID('a1bc0f6c-fc58-4df5-a840-1f589f0a4d
 	
>>> g = Group.objects.create(name='test'
>>> g.id
UUID('15c6ee18-b3b2-4739-8396-9d8f506a56
	$ ls -R media
	group user
	
	media/group:
	15c6
	
	media/group/15c6:
	ee18
	
	media/group/15c6/ee18:
	b3b2
	
	media/group/15c6/ee18/b3b2:
	4739
	
	media/group/15c6/ee18/b3b2/4739:
	8396
	
	media/group/15c6/ee18/b3b2/4739/8396:
	9d8f
	
	media/group/15c6/ee18/b3b2/4739/8396/9
	506a
	
	media/group/15c6/ee18/b3b2/4739/8396/9
	568f
	
	media/group/15c6/ee18/b3b2/4739/8396/9
	test.txt

計画通り！

複数のデータベースに対してマイグレーションを適用する

ZIP	apps-multidb.zip

案件によっては複数のデータベースを管理していることがあります。

今回は次のような設定にしました。 (該当部分)

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    },
    'users': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'users.sqlite3'),
```

```
    },
}
```

migrate コマンドの --database オプションで対象となるデータベースを指定すると マイグレーションは対象DBにだけ適用されます。

default

```
$ ./manage.py migrate
Operations to perform:
  Apply all migrations: accounts, admin,
Running migrations:
  Applying accounts.0001_initial... OK
  Applying contenttypes.0001_initial...
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_au
  Applying admin.0003_logentry_add_actio
  Applying contenttypes.0002_remove_cont
  Applying auth.0001_initial... OK
  Applying auth.0002_alter_permission_na
  Applying auth.0003_alter_user_email_ma
  Applying auth.0004_alter_user_username
  Applying auth.0005_alter_user_last_log
  Applying auth.0006_require_contenttype
  Applying auth.0007_alter_validators_ad
  Applying auth.0008_alter_user_username
  Applying auth.0009_alter_user_last_nam
  Applying products.0001_initial... OK
  Applying sales.0001_initial... OK
  Applying sessions.0001_initial... OK
```

users

```
$ ./manage.py migrate --database=users
Operations to perform:
  Apply all migrations: accounts, admin,
Running migrations:
  Applying accounts.0001_initial... OK
  Applying contenttypes.0001_initial...
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_au
  Applying admin.0003_logentry_add_actio
  Applying contenttypes.0002_remove_cont
  Applying auth.0001_initial... OK
  Applying auth.0002_alter_permission_na
  Applying auth.0003_alter_user_email_ma
  Applying auth.0004_alter_user_username
  Applying auth.0005_alter_user_last_log
  Applying auth.0006_require_contenttype
  Applying auth.0007_alter_validators_ad
  Applying auth.0008_alter_user_username
  Applying auth.0009_alter_user_last_nam
  Applying products.0001_initial... OK
  Applying sales.0001_initial... OK
  Applying sessions.0001_initial... OK
```

default

```
sqlite> .table
accounts_user           django_content_t
auth_group              django_migration
auth_group_permissions  django_session
auth_permission         products_categor
django_admin_log        products_price
```

users

```
sqlite> .table
accounts_user           django_content_t
auth_group              django_migration
auth_group_permissions  django_session
auth_permission         products_categor
django_admin_log        products_price
```

同じように適用されていますが、 せっかくDBを分けているのに全部適用されるのはうれしくないです。NGなケースもあるでしょう。

Django には database router という機能があり、これを使うことで適用DBを透過的に振り分けられます。

今回は accounts/models.py の テーブルは users データベース に作られるようにしてみます。

apps/router.py

```
class Router:
    def allow_migrate(self, db, app_labe
        if db == 'users':
            return app_label == 'account
```

apps/settings.py

```
DATABASE_ROUTERS = ['apps.router.Router'
```

- マイグレーションを制御するには Router クラスの allow_migrate というメソッドにて、適用 **する** / **しない** を True / False で返却します。

    - DB名, アプリ名, モデル名(小文字) が文字列で参照できます。
    - `None` は `True` と判断され、適用されるようです。
        - ドキュメントでは None の動作は明記されていなかったので、 確実な動作を望む場合はもれなく真偽値を返却するようにしましょう。

- Router を指すモジュールパスを settings の `DATABASE_ROUTER` という変数にリスト形式で指定します。

DBをリセットしてもう一度適用してみます。

default	users
sqlite> .tables auth_group django_content_t auth_group_permissions django_migration auth_permission django_session django_admin_log products_categor	sqlite> .tables accounts_user django_migrations

期待通り `accounts/models.py` のテーブルは users DBにのみ存在していますね。

備考

今回は扱いませんでしたが、 Database router は他にもメソッドがあります。

気になる方は調べてみてください。