Les modèles en Django

Table des matières

I. Notion de modèle

Contexte

À moins que vous ne vouliez créer un site web simple, il y a peu de chances d'éviter d'interagir avec une forme de base de données lors de la création d'applications web modernes. Malheureusement, cela signifie généralement que vous devez mettre la main à la pâte avec le SQL. Heureusement pour vous, Django simplifie le problème : vous n'êtes pas obligé d'utiliser du SQL.

A. Comment ça marche ?

Les modèles de Django fournissent un format objet-relationnel (ORM) à la base de données sous-jacente. L'ORM est une technique de programmation informatique puissante qui facilite grandement le travail avec des bases de données relationnelles. Les bases de données les plus courantes sont programmées en SQL, mais chaque base de données implémente SQL à sa manière. Ce dernier peut être compliqué et difficile à apprendre : c'est de ça que l'ORM est née. Elle simplifie la programmation de la base de données en fournissant un mappage simple entre un objet et la base de données. Cela signifie que le développeur n'a pas besoin de connaître la structure de la base de données, ni de SQL complexe pour manipuler et récupérer des données. L'étape la plus importante devient donc la structuration des données, et donc la réalisation de schéma entité/association abordé dans le cours sur les bases de données relationnelles.

Dans Django, les modèles vont représenter les objets stockés dans la base de données. Lorsque vous créez un modèle, Django va interpréter ces données et générer du code SQL pour créer une table correspondante à l'objet dans la base de données, sans que vous ayez à écrire une seule ligne de SQL. Django préfixe le nom de la table avec le nom de votre application Django. Le modèle relie également les informations associées dans la base de données.

Dans le cas où vous avez deux modèles et que l'un des attributs du premier modèle est l'attribut du deuxième, le premier modèle va effectuer un suivi de l'instruction du deuxième. Par exemple, nous avons deux modèles : le premier est une table « *Auteur* » et le deuxième est une table « *Film* » avec l'ID de l'auteur en attribut. Ici, on va vouloir créer une relation entre les deux modèles.

Les liens de clés étrangères dans les modèles Django créent des relations entre les tables. Cette relation est créée en liant les modèles à une clé étrangère, c'est-à-dire que le champ `ID` dans la table « *Film* » est un champ clé lié au champ `ID` dans la table étrangère « *Auteurs* ». C'est une simplification, mais c'est un aperçu pratique de la façon dont l'ORM de Django utilise les données du modèle pour créer des tables de base de données. Nous approfondirons prochainement les modèles, ne vous inquiétez donc pas si vous ne comprenez pas à 100 % ce qu'il se passe actuellement. Les choses deviennent plus claires une fois que vous avez eu la chance de construire des modèles réels.

B. Bases de données et avantages

Django prend officiellement en charge cinq bases de données : PostgreSQL, MySQL, SQLite, Oracle, MariaDB (Django 3 uniquement). Il existe également plusieurs autres applications disponibles si vous devez vous connecter à une base de données non officiellement prise en charge. La préférence de la plupart des développeurs Django est PostgreSQL. MySQL est également un back-end de base de données commun pour Django.

L'installation et la configuration d'une base de données n'est pas une tâche pour un débutant. Heureusement, Django installe et configure SQLite automatiquement, sans aucune intervention de votre part. Nous utiliserons donc SQLite.

L'écriture de modèles en Python présente plusieurs avantages :

- **Simple :** écrire en Python est non seulement plus facile qu'écrire du SQL, mais c'est en plus moins sujet aux erreurs et plus efficace, car votre cerveau n'a pas besoin de passer constamment d'une langue à une autre.
- **Cohérent :** comme mentionné précédemment, SQL est incohérent entre les différentes bases de données. Vous souhaitez décrire vos données une fois et non créer des ensembles distincts d'instructions SQL pour chaque base de données sur laquelle l'application sera déployée.

- **Rapide :** toute application doit savoir quelque chose sur la structure de la base de données sous-jacente. Il n'y a que deux façons de le faire : avoir une description explicite des données dans le code de l'application, ou parcourir la base de données au moment de l'exécution. Comme l'introspection est trop coûteuse pour la machine et n'est pas parfaite, les créateurs de Django ont choisi la première option.
- **Contrôle de version :** le stockage de modèles dans votre base de code facilite le suivi des modifications de conception.
- **Métadonnées avancées :** le fait d'avoir des modèles décrits dans du code plutôt que dans SQL permet des types de données spéciaux (par exemple, des adresses e-mail) et offre la possibilité de stocker beaucoup plus de métadonnées que SQL.

Un inconvénient est que la base de données peut se désynchroniser avec vos modèles, mais Django prend en charge ce problème avec les migrations. Notez également que vous pouvez parcourir une base de données existante avec Django en utilisant la commande de gestion `inspectdb`.

Exercice : Quiz

[solution n°1 p.17]

Question 1

L'ORM est une technique de programmation qui facilite le travail avec des bases de données relationnelles.

- ○ Vrai
- ○ Faux

Question 2

Quelles bases de données prennent en compte Django 3 ?

- ☐ MySQL
- ☐ HBase
- ☐ MariaDB
- ☐ Informix
- ☐ Oracle
- ☐ dBase
- ☐ HSQLDB
- ☐ PostgreSQL

Question 3

Quel est l'inconvénient de l'écriture de modèle en Python ?

- ○ La base de données peut se désynchroniser avec les modèles
- ○ L'écriture de modèle en Python est lente
- ○ L'écriture de modèle en Python est complexe
- ○ L'écriture de modèle en Python n'est pas optimisée

Question 4

Quelle commande permet de parcourir une base de données avec Django ?

- Inspectdb
- Browsedb
- Traveldb
- Checkdb
- scandb

Question 5

Django installe et configure SQLite automatiquement.

- Vrai
- Faux

III. Requête en Django

A. Création de modèles

Maintenant que vous avez une idée de ce que sont les modèles Django, il est temps de voir comment on va les générer. Il y a beaucoup d'informations que nous pouvons enregistrer pour un film, mais nous commencerons par un modèle de base. Lorsque vous concevez un modèle de données pour la première fois, il est toujours judicieux de mapper les champs de la table avant de créer le modèle. Il existe de nombreuses approches pour mapper les structures de données, des simples tableaux aux mappages de données complexes utilisant un balisage spécial. Évidemment, pour simplifier les choses, ici, on va juste utiliser des tableaux.

FILM TABLE		
Film_ID	Id du film, clé primaire unique	int
Titre	Titre du film	String
Date	Date de sortie du film	Date
Réalisateur	Réalisateur du film	String

Cela devrait être simple : nous avons des noms adaptés aux bases de données pour les champs, une description du champ et le type de données à enregistrer dans le champ de la base de données. La description est à votre avantage lorsque vous revenez au modèle plus tard et que vous devez vous rappeler à quoi servait le champ. Vous pouvez également placer la description du champ dans votre modèle sous la forme de commentaires ou dans la `docstring` du modèle. Je n'ajouterais pas de commentaires ni de `docstrings` à mon code, mais ne vous gênez pas pour le faire.

En modèle Django, cette table se ferait comme cela :

```
from django.db import models

class Film(models.Model):
    Titre = models.CharField('Titre Film', max_length=120)
    Date = models.DateTimeField('Date Film')
    Realisateur = models.CharField(max_length=120)
```

Chacun de nos champs de modèle a un type de champ Django et des options de champ associées. Le modèle `Event` utilise trois types de champs différents : `CharField`, `DateTimeField` et `TextField`. Examinons plus en détails les types de champs et les options :

- **Ligne 4.** Le champ `Titre` est un `CharField`. Un `Charfield` est une courte ligne de texte (jusqu'à 255 caractères). Ici, l'option `max_length` définit la longueur maximale du nom de l'événement à 120 caractères. Dans le cas du `CharField`, le champ `max_Length` définit la longueur maximale du nom de l'événement à 120 caractères. Dans le cas du `CharField`, le champ `max_Length` est obligatoire. Le champ du nom a également un argument de nom détaillé. Le nom détaillé est utilisé pour créer un nom convivial pour le champ du modèle.
- **Ligne 5.** `Date` est un Django `DateTimeField`. Un `DateTimeField` enregistre un objet `datetime` Python. Le champ `Date` a un seul argument : le nom détaillé du champ.
- **Ligne 6.** Rien de nouveau, juste un `Charfield` Django. Vous pouvez vous renseigner sur les champs Django directement sur leur documentation en ligne, il y a toutes les options et arguments que prennent les champs.

Méthode

Ce modèle d'événement simple n'utilise qu'un petit sous-ensemble des types de champs de modèle et des options disponibles dans Django. Il existe également une référence complète à tous les champs et options du modèle dans la documentation de Django. Maintenant que nous avons créé le modèle, il est temps de l'ajouter à la base de données. On va commencer par migrer les modifications avec 2 commandes simples :

```
python manage.py makemigrations
```

```
python manage.py migrate
```

C'est tout ce que vous devez faire pour ajouter votre nouveau modèle à la base de données. À partir de là, il suffit de laisser faire Django pour générer le SQL adéquat. Il suffit d'écrire la commande suivante dans le terminal :

```
python3 manage.py sqlmigrate films 0001_initial
```

La commande `sqlmigrate` imprime le SQL de la migration nommée. Ici, il imprime le SQL pour la migration initiale, où Django crée la nouvelle table et ajoute les champs à la table. Vous pouvez vérifier ce que Django a créé dans la base de données via votre navigateur. Une chose que vous avez dû remarquer : on ne parle jamais de clef primaire tout au long de la création du modèle. En fait, Django crée automatiquement une clef primaire lorsqu'il ajoute un enregistrement à la base données. Il génère la clef primaire à l'aide de la fonction `AUTOINCREMENT` de votre base de données.

B. Méthodes de contrôle de données Django

Méthode

Django fournit les quatre fonctions de base de données (Créer, Lire, Mettre à jour et Supprimer (**CRUD**)) que vous attendez d'un framework web conçu pour les sites web basés sur les données. Cependant, Django utilise une API Python de haut niveau pour communiquer avec votre base de données, plutôt qu'avec SQL. Pour apprendre à utiliser l'API de base de données de Django, nous utiliserons le shell interactif Django. Le shell interactif de Django fonctionne comme le shell interactif Python classique, sauf qu'il charge le module de paramètres de votre projet et d'autres modules spécifiques à Django afin que vous puissiez travailler avec projet Django. Pour utiliser le shell interactif Django, vous devez d'abord avoir exécuté l'environnement virtuel. Ensuite, exécutez la commande suivante à partir de la racine de votre projet :

```
python3 manage.py shell
```

On se retrouve avec ça :

```
Python 3.8.5 (default, Jan 27 2021, 15:41:15)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
```

Maintenant, nous allons importer la classe que nous venons de créer :

```
from films.models import Film
```

Si vous essayez cette commande sur un shell Python classique, vous devriez avoir une erreur.

On va créer notre premier film avec les informations données dans la classe :

```
film1 = Film(Titre="Django the film", Date="2021-05-03", Realisateur="Ditesco")
```

Vous pouvez voir que j'ai importé la classe de modèle `Film` et que j'ai créé un nouvel objet `Film o`, nommé `film1`. Avant de continuer, jetez un œil à la base de données. Vous remarquerez que Django n'a pas encore enregistré l'objet dans la base de données : c'est dû à sa conception. L'accès à une base de données est coûteux en temps, donc Django n'atteint pas la base de données tant que vous n'avez pas explicitement indiqué à Django de sauvegarder l'objet. Pour sauvegarder un enregistrement dans la base de données, Django a une fonction de modèle appelée `save()` :

```
film1.save()
```

Vous recevrez probablement un avertissement de Django concernant l'enregistrement d'une date sans fuseau horaire, mais vous pouvez l'ignorer pour le moment ; nous allons corriger cet avertissement sous peu. Maintenant, lorsque vous vérifiez la base de données, vous pouvez voir que l'enregistrement a été ajouté à la table `films_film` dans la base de données. Vous pouvez aussi créer des films en utilisant la fonction instaurée par Django, nommée `create()` :

```
Film.objects.create(Titre="Django the horror", Date="2021-05-05", Realisateur="Moi")
```

Vous pouvez le voir ci-dessus, la méthode `create()` crée un nouvel enregistrement d'événement que Django inscrit automatiquement pour vous. Maintenant qu'on a créé quelques films, on aimerait pouvoir les récupérer. Une méthode simple pour récupérer tous les films dans la base de données est `all()` :

```
>>> film_list = Film.objects.all()
>>> film_list
<QuerySet [<Film: Film object (1)>, <Film: Film object (2)>, <Film: Film object (3)>,
 <Film: Film object (4)>, <Film: Film object (5)>]>
```

Comme vous pouvez le voir, ce n'est pas une sortie très utile : vous n'avez aucun moyen de savoir à quel événement `Film object` fait référence. Heureusement, Python a une méthode d'affichage qui est plus visuelle et que vous pouvez ajouter aux fonctions et classes Python. Revenons à la déclaration de notre modèle `Film` et ajoutons une fonction :

```
class Film(models.Model):
    Titre = models.CharField('Titre Film', max_length=120)
    Date = models.DateTimeField('Date sortie')
    Realisateur = models.CharField(max_length=120)

    def __str__(self):
        return "%s, %s" % (self.Titre, self.Realisateur)
```

Si vous essayez à nouveau d'afficher le résultat de `all()`, vous devriez avoir quelque chose de la sorte :

```
<QuerySet [<Film: Django the film, Ditesco>, <Film: Django the film, Ditesco>,
<Film: Django le film 2, Moi>, <Film: Django the horror, Moi>, <Film: Django th
e horror, Moi>]>
```

Vous pouvez aussi récupérer une seule donnée en passant par la fonction `get()` :

```
>>> film1 = Film.objects.get(id=1)
>>> film1
<Film: Django the film, Ditesco>
```

Comme vous pouvez le voir ici, on a juste récupéré le premier film en passant par l'ID dans la base de données. Vous pouvez aussi passer les attributs du film en paramètres : Django va générer la requête Query à votre place. La méthode `get()` ne retourne qu'une seule valeur, donc faites bien attention lorsque vous passez par un autre argument que l'ID que la requête ne retourne qu'une valeur. Dans le cas contraire, vous aurez une erreur :

```
>>> film2 = Film.objects.get(Realisateur="Moi")
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/carotte/Dev/django/lib/python3.8/site-packages/django/db/models/manager.py", line 85, in manager_method
    return getattr(self.get_queryset(), name)(*args, **kwargs)
  File "/home/carotte/Dev/django/lib/python3.8/site-packages/django/db/models/query.py", line 439, in get
    raise self.model.MultipleObjectsReturned(
films.models.Film.MultipleObjectsReturned: get() returned more than one Film -- it returned 3!
```

Évidemment, cette méthode retourne aussi une erreur si l'objet n'existe pas. Pour récupérer plusieurs objets ayant la même caractéristique, par exemple, le même réalisateur, il existe la méthode `filter()`. Cette méthode peut prendre autant d'arguments que le modèle possède d'attributs.

```
>>> film_list = Film.objects.filter(Realisateur="Moi")
>>> film_list
<QuerySet [<Film: Django le film 2, Moi>, <Film: Django the horror, Moi>, <Film: Django the horror, Moi>]>
```

La méthode `filter()` va chercher la valeur exacte qui correspond, mais utilisez l'argument `attribut contains=` en remplaçant l'attribut par l'attribut concerné.

Par exemple :

```
>>> Film.objects.filter(Titre__contains="Django")
```

Lorsqu'il n'y a qu'un seul underscore, le principal, c'est pour que les autres programmeurs comprennent que la méthode (ou l'attribut) est destinée à être privée.

Notez qu'il y a un double underscore après l'attribut. Si vous avez quelques connaissances en SQL, vous aurez facilement fait le lien entre `contains=` et la clause `LIKE`, ainsi que le lien entre `filter()` et la clause `WHERE`. Contrairement à la fonction `get()`, ici, la méthode `filter()` peut très bien retourner une liste vide si la requête Query ne représente rien.

Tout comme en SQL, vous avez la clause `ORDER BY`, en Django, vous avez la méthode `order_by()` qui prend en argument les attributs, avec quelques spécificités.

```
>>> Film.objects.order_by("-Realisateur", "Titre")
```

Cette méthode est très simple à utiliser, il suffit de mettre les arguments en paramètre. Par défaut, la méthode les affiche dans l'ordre croissant, le « - » juste avant `Realisateur`, c'est une option de Django pour les récupérer dans l'ordre décroissant. Il est très courant de vouloir définir un ordre de tri sur un sous-ensemble de vos enregistrements de base de données. La meilleure manière de chaîner les méthodes est `order_by`, car la méthode de base ne le fait pas. Les méthodes `all()` et `filter()` retournent des listes, donc vous pouvez facilement travailler avec à l'aide des méthodes implémentées de base par Python.

Il ne reste plus qu'à pouvoir mettre à jour les informations en cas de changement et mettre en place la suppression. Pour modifier un élément avec Django, rien de plus simple :

```
film1 = Film.objects.get(id=1)
film1.Titre = "Koe no katachi"
film1.save()
```

Ce n'est pas plus dur que de changer la valeur d'un objet dans une liste en Python, n'oubliez pas de sauvegarder la donnée modifiée à la fin. En réalité, on peut faire encore plus simple, car Django nous fournit encore une fois une méthode pour ça, la méthode `update()` :

```
>>> Film.objects.filter(Titre__contains="Django").update(Realisateur="Mickey")
4
```

Vous pouvez utiliser `update()` pour modifier un ou plusieurs éléments. En réalité, Django considère que vous modifiez forcément plusieurs éléments, car la fonction `update()` ne s'applique que pour des listes. Puisque `get()` retourne un seul élément, vous ne pouvez `update()` à partir de la méthode `get()`.

Pour supprimer les éléments dans une base de données, il y a la fonction `delete()` :

```
>>> Film.objects.get(id=1).delete()
(1, {'films.Film': 1})
>>> Film.objects.all().delete()
```

Fondamental

C'est la seule méthode abordée qui fonctionne aussi bien sur un seul élément que sur une liste d'éléments. La méthode `delete()` va retourner le nombre total d'éléments affectés, un dictionnaire avec les tables affectées par l'opération de suppression et le nombre d'éléments supprimés dans chaque table. Bien entendu, vous pouvez très bien supprimer à partir des méthodes `all()`, `filter()` et `order_by()`. Pour éviter la suppression accidentelle de toutes les données d'une table, Django vous oblige à utiliser explicitement la méthode `all()` pour supprimer tout ce qui se trouve dans une table.

Exercice : Quiz

[solution n°2 p.18]

Question 1

Django fournit trois fonctions de base de données.

- ○ Vrai
- ○ Faux

Question 2

Les quatre fonctions de base de données fournies par Django sont :

- ○ Créer, Lire, Mettre à jour, Supprimer
- ○ Créer, Lire, Ajouter, Mettre à jour
- ○ Créer, Lire, Supprimer, Déplacer
- ○ Lire, Ajouter, Déplacer, Mettre à jour

Question 3

Quelle est la traduction de la clause `ORDER BY` (en SQL) en Django ?

- ○ order_by()
- ○ orderby()
- ○ sort_in()
- ○ sort()
- ○ sortin()

Question 4

Que fait la méthode `filter(argument)` ?

- ☐ Filtre les données selon la caractéristique donnée en argument
- ☐ Supprime toutes les données liées à l'argument pour filtrer la table
- ☐ Ajoute des données liées à l'argument
- ☐ Est l'équivalent de `WHERE` en SQL
- ☐ Est l'équivalent de `SELECT` en SQL

Question 5

La méthode `get()` retourne plusieurs valeurs.

- ○ Vrai
- ○ Faux

V. Relations entre tables et modèles

A. Créer des relations entre tables

Dans un site web basé sur les données, il est rare d'avoir un tableau d'informations sans rapport avec les informations d'autres tableaux. Cela est principalement dû aux meilleures pratiques bien établies en matière de conception de bases de données.

Exemple

Prenons le champ `Realisateur` comme exemple. Si vous ne sauvegardez que le nom du réalisateur dans la base de données, vous pourriez vous en sortir en répétant le nom du lieu plusieurs fois dans vos enregistrements de base de données. Mais qu'en est-il si vous souhaitez enregistrer plus d'informations sur le réalisateur ? Les éléments de votre site doivent également inclure une date de naissance ou de mort, peut-être une biographie. Si vous ajoutez ces champs à votre table de films, vous pouvez voir que vous vous retrouverez avec beaucoup d'informations répétées, sans parler du travail de mettre à jour tous les éléments si certaines informations changent.

Fondamental

La normalisation de la base de données est le processus de conception de vos tables pour minimiser ou éliminer la répétition des données. En termes simples, la normalisation consiste à conserver les données associées dans des tables séparées et à lier des tables via des relations. La normalisation de la base de données est également conforme à la philosophie de Django.

Ainsi, comme vous vous en doutez, Django simplifie la création de relations entre des tables d'informations. Avec notre exemple de réalisateur, il serait bon d'avoir toutes les informations sur le réalisateur dans une seule table et de créer un lien vers ces informations à partir de la table *film*. On va créer ce lien dans Django avec une clé étrangère. Plusieurs « *`film`* » liés à un élément de « *`realisateur`* » sont un exemple de relation plusieurs-à-un dans le langage des bases de données relationnelles. En regardant la relation dans la direction opposée, nous obtenons une relation un-à-plusieurs, c'est-à-dire qu'un élément est lié à de nombreux éléments. Django fournit des méthodes `QuerySet` pour naviguer dans les relations dans les deux sens (de un-à-plusieurs, et de plusieurs-à-un).

Méthode

Commençons par créer le modèle « *`Realisateur`* » et ajoutons une `ForeignKey` à `Film` :

```
class Realisateur(models.Model):
    Nom = models.CharField(max_length=120)
    Prenom = models.CharField(max_length=120)
    Naissance = models.DateTimeField('Date de naissance')
    def __str__(self):
        return "%s, %s" % (self.Nom, self.Prenom)

class Film(models.Model):
    Titre = models.CharField('Titre Film', max_length=120)
    Date = models.DateTimeField('Date sortie')

    Realisateur = models.ForeignKey(Realisateur, blank=True, null=True, on_delete=models.CASCADE)

    def __str__(self):
        return "%s, %s" % (self.Titre, self.Realisateur)
```

Il existe un point important dans le modèle `Film` :

Le type de champ de « *realisateur* » d'un champ `CharField` à un champ `ForeignKey`. Le premier argument est le nom du modèle associé. Nous avons ajouté `blank = True` et `null = True` pour permettre l'ajout d'un nouveau film sans réalisateur. L'argument `on_delete` est requis pour les clés étrangères. `CASCADE` signifie que, si un élément est supprimé, toutes les informations associées dans d'autres tables seront également supprimées.

Avant d'ajouter les nouveaux modèles à la base de données, assurez-vous de supprimer les éléments de la table `Film`. Sinon la migration échouera, car les anciens éléments ne seront pas liés à la nouvelle table. Revenez au shell interactif Django et supprimez tous les enregistrements de films à l'aide de la méthode `all().delete()`. Après ça, redémarrez votre shell en faisant `exit()`, puis en relançant un shell Django. Après avoir migré les modifications, vous pouvez voir que Django a ajouté la table `films_realisateur`, et le champ `realisateur` dans la table `films_film` a été renommé `realisateur_id`.

B. Gérer les relations entre tables

Méthode

En raison de la nature bidirectionnelle des relations de base de données et de la nécessité de maintenir l'intégrité référentielle, les actions de base de la base de données fonctionnent différemment avec les objets associés. Par exemple, essayons d'ajouter un nouvel événement dans le shell Django en utilisant le même code que tout à l'heure :

```
>>> from films.models import Film
>>> films1 = Film(Titre="Tokyo revengers", Date="2021-05-05", Realisateur="Vous")
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/carotte/Dev/django/lib/python3.8/site-packages/django/db/models/base.py", line 485, in __init__
    _setattr(self, field.name, rel_obj)
  File "/home/carotte/Dev/django/lib/python3.8/site-packages/django/db/models/fields/related_descriptors.py", line 215, in __set__
    raise ValueError(
ValueError: Cannot assign "'Vous'": "Film.Realisateur" must be a "Realisateur" instance.
>>>
```

Nous avons créé une relation entre la table `film` et la table `réalisateur`, donc Django s'attend à ce que nous transmettions une instance d'un objet `Réalisateur`, pas le nom d'un réalisateur. C'est une façon pour Django de maintenir l'intégrité référentielle entre les tables de la base de données. Pour que vous puissiez sauvegarder un nouveau `Film`, il doit y avoir un enregistrement de réalisateur correspondant dans la table `réalisateur`. Nous allons donc créer un `Réalisateur` pour notre `Film` :

```
Realisateur.objects.create(Nom="Sartre", Prenom="Jean-Paul", Naissance="1990-01-01")
```

Grâce à cette instance de réalisateur, on peut donc créer notre film :

```
films1 = Film(Titre="Tokyo revengers", Date="2021-05-05", Realisateur=Realisateur.objects.get(id=1))
```

(Il est bien sûr plus simple de le faire avec la méthode `rea = Realisateur() ; rea.save()`)

En raison de la relation créée entre la table `film` et la table `réalisateur`, lorsque vous accédez au champ `ForeignKey` à partir d'une instance de film, Django renvoie une instance de l'objet `réalisateur` associé directement :

```
>>> films1.Realisateur
<Realisateur: Sartre, Jean-Paul>
```

On va aussi vouloir récupérer tous les films réalisés par Jean-Paul Sartre, Django nous offre la fonctionnalité `film_set` qui est une liste de films :

```
>>> rea1.film_set.all()
<QuerySet [<Film: Tokyo revengers, Sartre, Jean-Paul>]>
```

Comme toutes les listes, vous pouvez y appliquer les méthodes `all()`, `filter()`, etc.

Exercice : Quiz

[solution n°3 p.19]

Question 1

À quoi sert la normalisation d'une table ?

- Minimiser ou éliminer la répétition des données
- Ajouter des données manquantes
- Retirer les données hors sujet
- Minimiser le nombre d'erreurs dans une table

Question 2

Que signifie la clause `CASCADE` ?

- Si un élément est supprimé, alors toutes les informations liées à cet élément seront supprimées
- Ajoute automatiquement des `Foreignkey` entre les tables
- Remplace automatiquement la clé primaire si elle est supprimée

Question 3

Qu'est-ce qu'une `Foreignkey` ?

- Une clé étrangère
- Une clé primaire
- Un attribut
- Une clé perdue

Question 4

Si l'on veut rajouter des éléments dans la table fictive `Jeu`, on utilise la commande :

```
Jeu.objects.create(contenu)
```

- Vrai
- Faux

Question 5

Qu'est-ce qu'une `primary key` ?

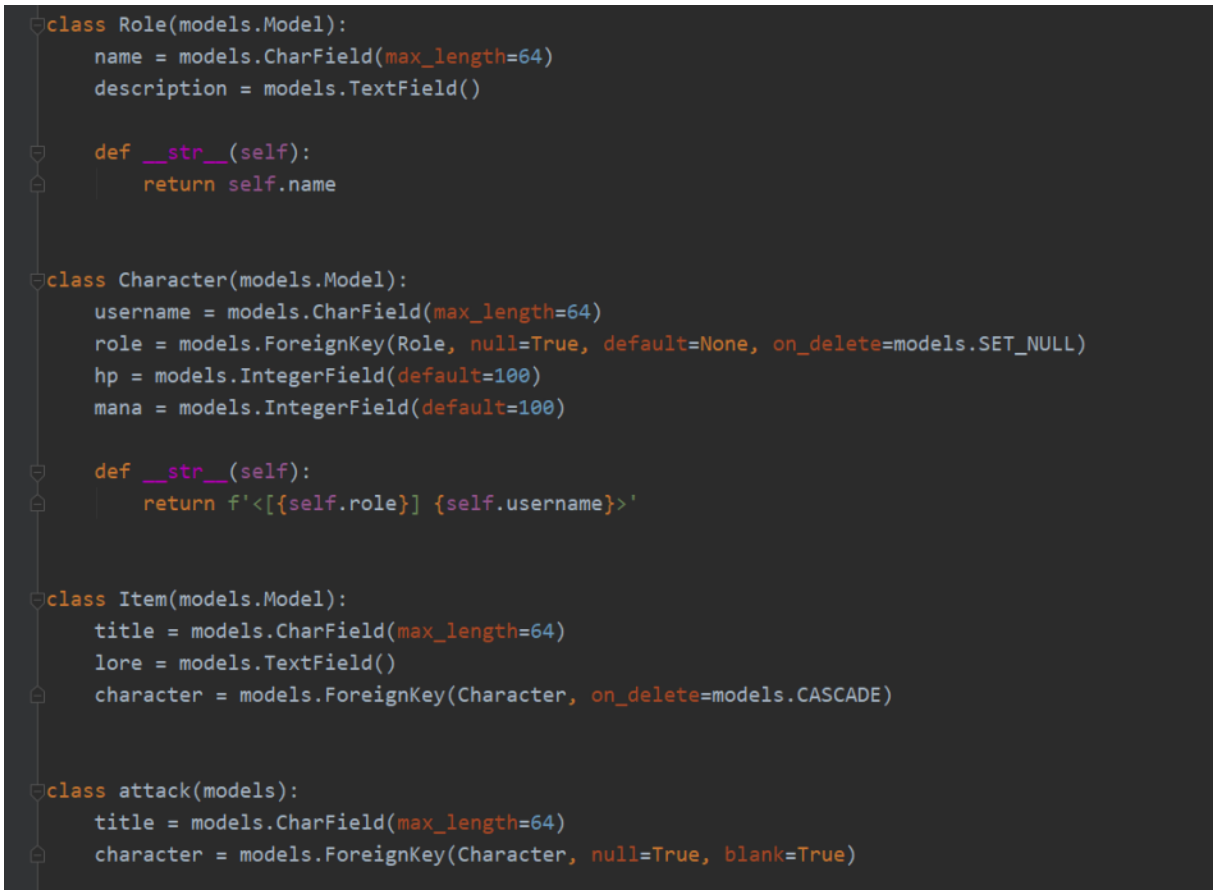
- Une clé étrangère
- Une clé primaire
- Un attribut
- Une clé perdue

VII. Essentiel

Les modèles sont une partie essentielle de l'ORM Django. Ils permettent de faire le lien entre le code Python du serveur web et la base de données utilisée. Les modèles permettent aussi bien une gestion de l'architecture de votre base de données via Django (grâce aux migrations, par exemple) qu'une facilitation d'usage dans le développement de vos vues.

VIII. Auto-évaluation

Le but de cet exercice final est d'utiliser nos modèles et de développer du code autour. Les questions de cet exercice final seront principalement axées sur les modèles décrits dans l'image suivante. Pour l'intégralité des questions, nous partirons du principe que la base de données est déjà peuplée d'au moins 1 élément dans chaque table avec pour clé primaire (ID) 1.



```python
class Role(models.Model):
    name = models.CharField(max_length=64)
    description = models.TextField()

    def __str__(self):
        return self.name


class Character(models.Model):
    username = models.CharField(max_length=64)
    role = models.ForeignKey(Role, null=True, default=None, on_delete=models.SET_NULL)
    hp = models.IntegerField(default=100)
    mana = models.IntegerField(default=100)

    def __str__(self):
        return f'<[{self.role}] {self.username}>'


class Item(models.Model):
    title = models.CharField(max_length=64)
    lore = models.TextField()
    character = models.ForeignKey(Character, on_delete=models.CASCADE)


class attack(models):
    title = models.CharField(max_length=64)
    character = models.ForeignKey(Character, null=True, blank=True)
```

Exercice 1 : Quiz

[solution n°4 p.20]

Question 1

Lors de la migration Django, le modèle « *Attack* » n'a pas pu être créé. Quel(s) problème(s) y a-t-il avec la classe de ce modèle ?

- ☐ Le nom du modèle est en minuscules
- ☐ Le champ `ForeignKey` demande de définir une valeur `on_delete`
- ☐ Le nom du champ `title` est déjà utilisé par le modèle « *Item* »
- ☐ La classe du modèle n'est pas du bon type

Question 2

Quelles sont les relations définies par les classes ?

- ☐ Un « *Character* » DOIT être lié à UN « *Role* »
- ☐ Un « *Role* » PEUT être rattaché à PLUSIEURS « *Character* » (donc plusieurs OU aucun)
- ☐ Un « *Item* » DOIT être lié à UN « *Character* »
- ☐ Un « *Character* » PEUT être lié à UN « *Role* » (et donc peut aussi ne pas en avoir)

Question 3

Cochez ce qui est vrai.

- ☐ Le champ « *name* » du modèle « *Role* » est limité à 64 caractères
- ☐ Le champ « *description* » du modèle « *Role* » est limité à 64 caractères
- ☐ Le champ « *role* » du modèle « *Character* » peut être nul
- ☐ Le champ « *role* » du modèle « *Character* » peut être un objet « *Role* »
- ☐ Les champs « *hp* » et « *mana* » du modèle « *Character* » sont des champs contenant des valeurs numériques

Question 4

Avec un objet `Role`, il est possible de récupérer la liste des objets `Character` qui lui sont liés.

- ○ Vrai
- ○ Faux

Question 5

Quelle(s) commande(s) permet(tent) la récupération de tous les modèles « *Character* » dont le nom d'utilisateur (`username`) contient « *Plop* » ?

- ○ Character.objects.filter(username__contains=”Plop”)
- ○ Character.objects.get(username=”Plop”)
- ○ Character.objects.exclude(username=”Plop”)
- ○ Character(username=”Plop”).get()

Question 6

Quelle(s) commande(s) permet(tent) la récupération de tous les modèles « *Character* » dont le nom d'utilisateur (`username`) contient « *Plop* » et est rattaché à la variable `target_role` (contenant bien évidemment un objet de type `Role`) ?

- ☐ target_role.character_set.filter(username__contains=”Plop”)
- ☐ Character.filter(username__contains=”Plop”, role=target_role)
- ☐ target_role.get(username_contains=”Plop”)
- ☐ Character.objects.filter(username__contains=”Plop”)

Question 7

Quelle(s) commande(s) est/(sont) invalide(s) ?

- ☐ Character.objects.filter(username__contains="Plop").delete()
- ☐ Role.objects.get(id=1).update(role="Some text")
- ☐ Character.objects.all().update(hp=100)
- ☐ Role.objects.delete()

Question 8

Quelle chaîne de caractères s'affiche après les commandes suivantes ?

```
r = Role.objects.create(name="Warrior", description="A strong fighter")
c = Character.objects.create(username="Plop", role=r)
print(c)
```

- ○ <Character: Character object (2)>
- ○ Plop
- ○ <Plop A strong fighter>
- ○ <[{self.role}] {self.username}>
- ○ <[Warrior] Plop>

Question 9

La commande suivante produit des erreurs, pourquoi ?

```
Character.objects.create(username="Plop", hp="100", description="Some text")
```

- ☐ On ne définit pas de valeur pour l'attribut `role`
- ☐ La valeur utilisée pour l'attribut `hp` n'est pas valide
- ☐ Les objets `Character` n'ont pas d'attribut `description`
- ☐ La méthode utilisée n'est pas bonne

Exercice 11

[solution n°5 p.23]

Dans le cas où je voudrais créer un nouveau modèle dans ma base de données, et définir manuellement un élément dans cette nouvelle table, dans quel ordre dois-je effectuer les actions suivantes ?

1. Utiliser la commande `python manage.py makemigrations`
2. Utiliser la commande `python manage.py shell`
3. Éditer le fichier **models.py**
4. Utiliser la commande `python manage.py migrate`

Réponse : ___ ___ ___ ___

Solutions des exercices

Exercice p. 4 Solution n°1

Question 1

L'ORM est une technique de programmation qui facilite le travail avec des bases de données relationnelles.

- (●) Vrai
- ( ) Faux

L'ORM est une technique de programmation informatique puissante qui facilite grandement le travail avec des bases de données relationnelles.

Question 2

Quelles bases de données prennent en compte Django 3 ?

- [x] MySQL
- [ ] HBase
- [x] MariaDB
- [ ] Informix
- [x] Oracle
- [ ] dBase
- [ ] HSQLDB
- [x] PostgreSQL

Django 3 est pris en compte par PostgreSQL, MySQL, SQLite, Oracle, et MariaDB.

Question 3

Quel est l'inconvénient de l'écriture de modèle en Python ?

- (●) La base de données peut se désynchroniser avec les modèles
- ( ) L'écriture de modèle en Python est lente
- ( ) L'écriture de modèle en Python est complexe
- ( ) L'écriture de modèle en Python n'est pas optimisée

L'inconvénient que l'écriture de modèle en Python présente est que la base de données peut se désynchroniser avec les modèles. Les autres sont l'inverse des avantages que présente l'écriture de modèle en Python.

Question 4

Quelle commande permet de parcourir une base de données avec Django ?

- (●) Inspectdb
- ( ) Browsedb
- ( ) Traveldb

- Checkdb
- scandb

La commande `inspectdb` permet de parcourir une base de données et de produire un fichier *models.py*. Cette commande est très utile pour intégrer à Django une base de données déjà existante.

Question 5

Django installe et configure SQLite automatiquement.

- (•) Vrai
- Faux

Par défaut, lors de la création d'un projet Django, sans changer les *settings*, Django va créer et utiliser une base de données SQLite.

Exercice p. 10 **Solution n°2**

Question 1

Django fournit trois fonctions de base de données.

- Vrai
- (•) Faux

Django fournit en réalité les quatre fonctions de BDD.

Question 2

Les quatre fonctions de base de données fournies par Django sont :

- (•) Créer, Lire, Mettre à jour, Supprimer
- Créer, Lire, Ajouter, Mettre à jour
- Créer, Lire, Supprimer, Déplacer
- Lire, Ajouter, Déplacer, Mettre à jour

Les quatre fonctions de base de données fournies par Django sont :

- Créer (INSERT) : ajouter de nouveaux éléments à la base de données.
- Lire (SELECT) : récupérer un ou plusieurs éléments de la base de données.
- Mettre à jour (UPDATE) : mettre à jour les valeurs des champs d'un ou plusieurs éléments dans la base de données.
- Supprimer (DELETE) : retirer un ou plusieurs éléments de la base de données.

Question 3

Quelle est la traduction de la clause `ORDER BY` (en SQL) en Django ?

- (•) order_by()
- orderby()
- sort_in()

- ○ sort()
- ○ sortin()

Q `ORDER BY` en SQL se traduit en Django par la fonction `order_by()`. Cette fonction permet de ranger les éléments en fonction d'un ou plusieurs champs.

Question 4

Que fait la méthode `filter(argument)` ?

- ☑ Filtre les données selon la caractéristique donnée en argument
- ☐ Supprime toutes les données liées à l'argument pour filtrer la table
- ☐ Ajoute des données liées à l'argument
- ☑ Est l'équivalent de `WHERE` en SQL
- ☐ Est l'équivalent de `SELECT` en SQL

Q La méthode `filter()` filtre les données selon la caractéristique donnée en argument et fait le même travail que la clause `WHERE` en SQL.

Question 5

La méthode `get()` retourne plusieurs valeurs.

- ○ Vrai
- ◉ Faux

Q La méthode `get()` ne retourne qu'une seule valeur. C'est de cette manière que l'on récupère un unique objet (en opposition avec `filter()`).

Exercice p. 13 **Solution n°3**

Question 1

À quoi sert la normalisation d'une table ?

- ◉ Minimiser ou éliminer la répétition des données
- ○ Ajouter des données manquantes
- ○ Retirer les données hors sujet
- ○ Minimiser le nombre d'erreurs dans une table

Q La normalisation est le processus de conception de vos tables pour minimiser ou éliminer la répétition des données.

Question 2

Que signifie la clause `CASCADE` ?

- ◉ Si un élément est supprimé, alors toutes les informations liées à cet élément seront supprimées
- ○ Ajoute automatiquement des `Foreignkey` entre les tables
- ○ Remplace automatiquement la clé primaire si elle est supprimée

Si un élément est supprimé, alors toutes les informations liées à cet élément seront supprimées grâce à la clause `CASCADE`.

Question 3

Qu'est-ce qu'une `Foreignkey` ?

- (●) Une clé étrangère
- ( ) Une clé primaire
- ( ) Un attribut
- ( ) Une clé perdue

Une `Foreignkey` permet une liaison via une clé étrangère. C'est un champ qui va stocker, chez un premier élément, l'ID d'un autre élément pour permettre de faire un lien entre ces deux éléments.

Question 4

Si l'on veut rajouter des éléments dans la table fictive `Jeu`, on utilise la commande :

```
Jeu.objects.create(contenu)
```

- (●) Vrai
- ( ) Faux

La méthode `create` d'un gestionnaire de modèle (`Jeu.objects`) sera traduite par la création d'un nouvel élément dans la table `Jeu`. En paramètre, l'on peut fournir à Django les valeurs que l'on veut donner aux champs de ce nouvel élément.

Question 5

Qu'est-ce qu'une `primary key` ?

- ( ) Une clé étrangère
- (●) Une clé primaire
- ( ) Un attribut
- ( ) Une clé perdue

Une `primary key` est une clé primaire. C'est un identifiant unique qui est donné à un élément à sa création et qui restera inchangé. Cette clé permet à Django de dissocier les éléments d'une même table.

Exercice p. 14 Solution n°4

Question 1

Lors de la migration Django, le modèle « *Attack* » n'a pas pu être créé. Quel(s) problème(s) y a-t-il avec la classe de ce modèle ?

- [ ] Le nom du modèle est en minuscules
- [x] Le champ `ForeignKey` demande de définir une valeur `on_delete`
- [ ] Le nom du champ `title` est déjà utilisé par le modèle « *Item* »
- [x] La classe du modèle n'est pas du bon type

Pour créer un modèle, il faut créer une classe héritant du type `models.Model` pour qu'elle soit considérée comme un modèle par Django. Il suffit ensuite d'ajouter les bons attributs à cette classe pour que Django les reporte sur son modèle en base de données. Certains types d'attribut comme les `ForeignKey` ont besoin de certains paramètres pour pouvoir fonctionner. Ici, le paramètre `on_delete` permet de dire à Django quoi faire lors de la suppression d'un élément qui doit rompre une relation.

Question 2

Quelles sont les relations définies par les classes ?

- ☐ Un « *Character* » DOIT être lié à UN « *Role* »
- ☑ Un « *Role* » PEUT être rattaché à PLUSIEURS « *Character* » (donc plusieurs OU aucun)
- ☑ Un « *Item* » DOIT être lié à UN « *Character* »
- ☑ Un « *Character* » PEUT être lié à UN « *Role* » (et donc peut aussi ne pas en avoir)

Avec une relation `ForeignKey`, on lie un élément A à un élément B. Mais plusieurs éléments A peuvent être liés à un même élément B (sinon, on utilise un `OneToOneField`). Ici, deux personnages (`character`) peuvent avoir le même rôle, ce qui fait que ce rôle en question est lié à deux personnages. Enfin, quand l'on précise `null=True`, cela signifie que le champ peut être vide, ou, dans notre cas, qu'une relation est facultative.

Question 3

Cochez ce qui est vrai.

- ☑ Le champ « *name* » du modèle « *Role* » est limité à 64 caractères
- ☐ Le champ « *description* » du modèle « *Role* » est limité à 64 caractères
- ☑ Le champ « *role* » du modèle « *Character* » peut être nul
- ☑ Le champ « *role* » du modèle « *Character* » peut être un objet « *Role* »
- ☑ Les champs « *hp* » et « *mana* » du modèle « *Character* » sont des champs contenant des valeurs numériques

Les champs « *TextField* » ne sont pas limités en taille, contrairement aux champs « *CharField* », qui sont limités par leur valeur « *max_length* ». Comme dit plus tôt, un champ « *ForeignKey* » avec « *null=True* » peut être `null`, sinon il contiendra un objet avec un type correspondant au modèle passé en premier paramètre. Enfin, les champs « *IntegerField* » sont des nombres entiers.

Question 4

Avec un objet `Role`, il est possible de récupérer la liste des objets `Character` qui lui sont liés.

- ◉ Vrai
- ○ Faux

C'est tout à fait possible. Tout comme il est possible d'obtenir un objet `Role` depuis un objet `Character` Django permet son contraire. Et le contraire d'une `ForeignKey`, c'est une relation plusieurs-à-un.

Question 5

Quelle(s) commande(s) permet(tent) la récupération de tous les modèles « *Character* » dont le nom d'utilisateur (`username`) contient « *Plop* » ?

- ◉ Character.objects.filter(username__contains=”Plop”)
- ○ Character.objects.get(username=”Plop”)
- ○ Character.objects.exclude(username=”Plop”)
- ○ Character(username=”Plop”).get()

Pour récupérer tous les modèles « *Character* » dont le nom contient « *Plop* », il faut soumettre une requête (`filter`) avec les bons paramètres (`username__containes=”Plop”`) en passant par le gestionnaire d'objets de la classe `Character` (`Character.objects`).

Question 6

Quelle(s) commande(s) permet(tent) la récupération de tous les modèles « *Character* » dont le nom d'utilisateur (`username`) contient « *Plop* » et est rattaché à la variable `target_role` (contenant bien évidemment un objet de type `Role`) ?

- ☑ target_role.character_set.filter(username__contains=”Plop”)
- ☑ Character.filter(username__contains=”Plop”, role=target_role)
- ☐ target_role.get(username_contains=”Plop”)
- ☐ Character.objects.filter(username__contains=”Plop”)

Deux méthodes sont possibles. Soit faire une requête via le gestionnaire d'objets de la classe « *Character* », comme pour la question précédente, en ajoutant un paramètre à la requête (`role=target_role`) ; soit utiliser la variable `character_set` qui permet de récupérer les objets `Character` liés à notre objet cible.

Question 7

Quelle(s) commande(s) est/(sont) invalide(s) ?

- ☐ Character.objects.filter(username__contains=”Plop”).delete()
- ☑ Role.objects.get(id=1).update(role=”Some text”)
- ☐ Character.objects.all().update(hp=100)
- ☑ Role.objects.delete()

La classe `Role` n'a tout simplement pas d'attribut `lore` et il n'est pas possible de faire directement appel à `delete()` sur le gestionnaire d'objets d'une classe, cette méthode n'est accessible que sur un objet ou une requête. La commande suivante est valide : `Role.objects.all().delete()`.

Question 8

Quelle chaîne de caractères s'affiche après les commandes suivantes ?

```
r = Role.objects.create(name=”Warrior”, description=”A strong fighter”)
c = Character.objects.create(username=”Plop”, role=r)
print(c)
```

- ○ <Character: Character object (2)>
- ○ Plop
- ○ <Plop A strong fighter>
- ○ <[{self.role}] {self.username}>
- ◉ <[Warrior] Plop>

Avec nos trois commandes, nous créons un nouvel objet `Role`, ainsi qu'un nouvel objet `Character` rattaché à notre objet `Role`. Comme précisé dans l'image de contexte, nos classes `Role` et `Character` ont la méthode `__str__()` de définie et cette méthode sert pour l'affichage d'un objet, donc ce n'est pas l'affichage par défaut des modèles Django. Il suffit ensuite de décoder le retour de ces méthodes.

Question 9

La commande suivante produit des erreurs, pourquoi ?

```
Character.objects.create(username="Plop", hp="100", description="Some text")
```

- ☐ On ne définit pas de valeur pour l'attribut `role`
- ☑ La valeur utilisée pour l'attribut `hp` n'est pas valide
- ☑ Les objets `Character` n'ont pas d'attribut `description`
- ☑ La méthode utilisée n'est pas bonne

La méthode utilisée est bonne et l'attribut `role` a une valeur par défaut (`None`), tout comme les attributs `mana` ou même `hp`, qui ont tous deux 100 comme valeur par défaut. Par contre, `hp` est un champ nombre, pas un champ texte, et les objets `Character` n'ont pas d'attribut `description`.

Exercice p. 16 **Solution n°5**

Dans le cas où je voudrais créer un nouveau modèle dans ma base de données, et définir manuellement un élément dans cette nouvelle table, dans quel ordre dois-je effectuer les actions suivantes ?

Éditer le fichier **models.py**

Utiliser la commande `python manage.py makemigrations`

Utiliser la commande `python manage.py migrate`

Utiliser la commande `python manage.py shell`

Les étapes sont les suivantes : créer une classe pour notre modèle dans le fichier **models.py**, créer un fichier de migration avec la commande `python manage.py makemigrations`, appliquer cette nouvelle migration avec la commande `python manage.py migrate`. Une fois la migration appliquée, je peux utiliser la commande `python manage.py shell` pour peupler manuellement ma table.