1. Création et gestion des migrations

Les migrations permettent de versionner la structure de la base de données et de faciliter la gestion des modifications de schéma.

- **Créer une migration** pour une application :

  ```
  manage.py makemigrations
  ```

- **Créer une migration pour une application spécifique** :

  ```
  manage.py makemigrations nom_de_l_application
  ```

- **Exécuter les migrations** (applique les changements sur la base de données) :

  ```
  manage.py migrate
  ```

- **Annuler la dernière migration** :

  ```
  manage.py migrate nom_de_l_application last
  ```

- **Revenir à une migration spécifique** :

  ```
  manage.py migrate nom_de_l_application 0001_initial
  ```

- **Afficher l'état des migrations** (voir quelles migrations ont été appliquées) :

  ```
  manage.py showmigrations
  ```

- **Vider la base de données (supprimer toutes les données et réappliquer les migrations)** :

  ```
  manage.py flush
  ```

- **Supprimer la base de données et recréer les tables** (réinitialiser la base de données) :

  ```
  manage.py migrate --run-syncdb
  ```

2. Gestion des modèles

Les modèles représentent les tables de la base de données dans Django et permettent d'interagir avec les données.

- **Créer un modèle** dans une application (`models.py`) :

  ```
  from django.db import models

  class Article(models.Model):
      titre = models.CharField(max_length=100)
      contenu = models.TextField()
      date_pub = models.DateTimeField(auto_now_add=True)
  ```

- **Ajouter un champ à un modèle** : Après avoir modifié un modèle, vous devez créer une migration pour appliquer les changements :

```
manage.py makemigrations
manage.py migrate
```

3. Gestion des données avec le shell Django

- **Accéder à l'interpréteur avec le shell Django** :

```
manage.py shell
```

- **Exemple d'utilisation dans le shell Django** :
  - Importer un modèle et interagir avec la base de données :

```
from monapp.models import Article
```

  - Ajouter un enregistrement :

```
article = Article(titre="Mon premier article", contenu="C'est
un article très intéressant.")
article.save()
```

  - Récupérer tous les articles :

```
articles = Article.objects.all()
```

  - Filtrer les articles :

```
articles = Article.objects.filter(titre="Mon premier article")
```

  - Mettre à jour un enregistrement :

```
article = Article.objects.get(id=1)
article.titre = "Titre mis à jour"
article.save()
```

  - Supprimer un enregistrement :

```
article = Article.objects.get(id=1)
article.delete()
```

4. Gestion des commandes SQL personnalisées

Django permet d'exécuter des requêtes SQL personnalisées directement dans la base de données.

- **Exécuter une requête SQL via Django** :

```
from django.db import connection

with connection.cursor() as cursor:
    cursor.execute("SELECT * FROM monapp_article")
```

```
rows = cursor.fetchall()
for row in rows:
    print(row)
```

5. Importation et exportation des données

- **Importer des données depuis un fichier JSON ou CSV** dans la base de données :

  Vous pouvez utiliser des packages comme `django-import-export` ou créer des commandes personnalisées pour importer des données depuis des fichiers.

- **Exporter des données dans un fichier JSON** :

  ```
  manage.py dumpdata nom_de_l_application > data.json
  ```

- **Importer des données depuis un fichier JSON** :

  ```
  manage.py loaddata data.json
  ```

6. Création de fixtures

Les fixtures permettent de pré-remplir la base de données avec des données de test ou de sauvegarde.

- **Créer une fixture JSON** pour une application :

  ```
  manage.py dumpdata nom_de_l_application > fixtures.json
  ```

- **Charger une fixture JSON** dans la base de données :

  ```
  manage.py loaddata fixtures.json
  ```

7. Gestion des utilisateurs et des permissions

- **Créer un utilisateur superutilisateur** (administrateur de l'application) :

  ```
  manage.py createsuperuser
  ```

- **Attribution de permissions et de rôles dans les modèles** : Vous pouvez définir des permissions personnalisées dans les modèles via les métadonnées `Meta` :

```
class Article(models.Model):
    titre = models.CharField(max_length=100)
    contenu = models.TextField()

    class Meta:
        permissions = [
            ("peut_lire_article", "Peut lire un article"),
            ("peut_editer_article", "Peut éditer un article"),
        ]
```

8. Optimisation des requêtes

- **Afficher les requêtes SQL exécutées par Django** : Django permet de voir les requêtes SQL générées par ORM en activant les logs :

```
from django.db import connection
print(connection.queries)
```

- **Optimiser les requêtes en utilisant `select_related` et `prefetch_related`** : Ces méthodes sont utilisées pour réduire le nombre de requêtes SQL lors des relations de type **`ForeignKey`**, **`OneToOne`**, ou **`ManyToMany`**.
    - Utilisation de **`select_related`** pour les relations **`ForeignKey`** et **`OneToOne`** :

      ```
      articles = Article.objects.select_related('categorie').all()
      ```

    - Utilisation de **`prefetch_related`** pour les relations **`ManyToMany`** :

      ```
      articles = Article.objects.prefetch_related('tags').all()
      ```

9. Exemples de commandes liées à la base de données

- **Effectuer des requêtes avec des filtres complexes** :

```
from monapp.models import Article
articles =
Article.objects.filter(titre__icontains='Django').order_by('date_pub'
)
```

- **Compter les objets dans un modèle** :

```
count = Article.objects.count()
```

10. Sauvegarde et restauration de la base de données

Bien que Django ne propose pas de commandes intégrées pour effectuer des sauvegardes de base de données, vous pouvez utiliser des outils de base de données tels que `pg_dump` (PostgreSQL), **`mysqldump`** (MySQL) ou d'autres solutions pour sauvegarder et restaurer la base de données.