1. Introducción.

Mediante el uso de bases de datos, la información con la que trabajan nuestras aplicaciones puede ser almacenada de forma permanente y recuperada posteriormente según nos convenga.

El acceso a las bases de datos puede llevarse a cabo básicamente mediante dos técnicas: utilizando una API junto al lenguaje SQL, o bien mediante un sistema ORM.

Un sistema ORM (Object-Relational Mapping) permite tratar como objetos a los registros guardados en una base de datos relacional. Cada registro o fila de una tabla se corresponde con un objeto.

Ambos sistemas tienen sus ventajas e inconvenientes que se detallan a continuación.

Acceso a bases de datos mediante DB-API y SQL	
Ventajas	**Inconvenientes**
Control total sobre las operaciones a ejecutar en la BD, ya que utilizamos directamente SQL.	Se necesita más código, ya que debe mezclarse código Python con SQL.
Mejor para consultas complejas.	Exposición a ataques del tipo SQL injection.
Algunas características de la BD sólo son accesibles de esta forma.	La conexión con el servidor suele ser una tarea más compleja.
La visión directa de las sentencias SQL aporta más claridad.	

Acceso a bases de datos mediante un sistema ORM	
Ventajas	**Inconvenientes**
Se utilizan sentencias Python. No es necesario pensar en SQL.	Se necesitan conocimientos extra para trabajar con estos sistemas.
Aplicaciones mejor organizadas.	Es más difícil realizar consultas complejas.
Independiente del SGBD.	Las aplicaciones pueden ser más lentas, sobre todo si se trabaja con grandes conjuntos de datos.
Un ORM previene problemas como SQL injection.	Aplicaciones más complejas de actualizar.
Agiliza la construcción de aplicaciones, ahorrando tiempo.	Falta de control o desconocimiento de lo que sucede por detrás.

En esta unidad estudiaremos estas dos formas de acceso. Además, el estudio de un sistema ORM nos servirá en temas futuros, como el desarrollo de aplicaciones web con Django.

Para sacar partido a esta unidad es imprescindible tener conocimientos relacionados con las bases de datos relacionales.

2. Acceso a bases de datos mediante DB-API y SQL.

Los pasos previos que debemos realizar antes de desarrollar una aplicación que trabaje con bases de datos son:

- Elegir un servidor de bases de datos e instalarlo en nuestro equipo.
- Descargar e instalar las librerías necesarias para que nuestra aplicación pueda comunicarse con ese servidor. Ojo, siempre dentro del entorno virtual de trabajo.
- Desarrollar la aplicación importando las clases/funciones a partir de las librerías instaladas.

Python permite trabajar con varios sistemas gestores de bases de datos: PostgreSQL, SQLite, MySQL, etc. **En esta parte de la unidad utilizaremos MySQL,** por tanto, la primera operación que debemos realizar será instalar este servidor en nuestro sistema.

Para instalar MySQL en nuestro ordenador disponemos de varias opciones:

- Descargar el servidor y el cliente MySQL Workbench desde la web de MySQL.
- Descargar Xampp. Una forma sencilla de instalar un servidor MariaDB (compatible con MySQL) y un cliente web.
- Crear un contenedor ***docker*** con una imagen de MySQL.

Puedes descargar las bases de datos de ejemplo y los archivos necesarios para usar MySQL con docker en los recursos de la unidad. Para probar las diferentes operaciones utilizaremos la base de datos ***almacén*** y sus tablas ***clientes, productos y categoriasproductos***. Dispones de la base de datos en el archivo ***almacen.sql.***

Podemos configurar además, nuestro VSC con las extensiones MySQL y Database Client JDBC, que permiten interactuar con la base de datos sin salir de nuestro editor.

Python ofrece acceso a bases de datos relacionales mediante la especificación DB-API (Database API). Esta especificación incluye una serie de clases estándar para trabajar con los distintos motores de bases de datos relacionales.

Para instalar en nuestro entorno virtual las librerías de MySQL debemos ejecutar en ese entorno ***pip install pymysql[rsa].***

Una vez instalados el servidor y las librerías podemos comenzar a escribir código Python.

Las aplicaciones que trabajen con bases de datos deben incluir los siguientes pasos:

- Obtener un objeto para la conexión (pymysql.connections.Connection).
- Obtener un cursor a partir de la conexión (pymysql.cursors.Cursor).
- Ejecutar una sentencia SQL a partir del cursor.

Si te has decidido por docker, la base de datos se cargará en el servidor de forma automática al ejecutar docker compose. Si has elegido instalar MySQL Workbench o Xampp, deberás importar la base de datos (almacen.sql) a tu servidor.

2.1. Sentencias SQL con y sin parámetros.

Como hemos indicado anteriormente, las sentencias SQL se ejecutan a través de un objeto cursor. Estas sentencias pueden incluir o no parámetros en su código.

Un ejemplo de sentencia ***SELECT*** sin parámetros podría ser el siguiente: ***select nombre, telefono from clientes.*** Las sentencias SELECT sin parámetros realizan siempre la misma acción y producen por tanto los mismos resultados cada vez que sean ejecutadas.

Una sentencia ***SELECT*** con parámetros proporcionará los registros requeridos en función de los parámetros pasados, ya que estos parámetros se suelen usar para expresar condiciones. Siguiendo con el ejemplo anterior, podríamos afinar un poco y mostrar nombre y teléfono de todos los clientes con nombre Luis de esta forma: ***select nombre, telefono from clientes where nombre = %s.*** En este ejemplo el parámetro está representado por ***%s*** y debería ser sustituido por un valor concreto (Luis) antes de ejecutar la sentencia.

Con ***PyMyql*** podremos pasar parámetros a las sentencias de dos formas, **mediante una tupla y mediante un diccionario**. Esto es aplicable a todas las sentencias de tipo DML, es decir, ***SELECT, INSERT, UPDATE y DELETE.***

En los ejemplos que siguen a continuación, se utilizarán tanto tuplas como diccionarios para pasar parámetros.

2.2. Consultar registros.

El proceso para realizar una consulta es sencillo:

- En primer lugar se obtiene la conexión. Para ello utilizaremos la sentencia ***with,*** que nos libera de cerrar dicha conexión, tal y como sucedía con los archivos.
- A partir de ella se obtiene el cursor. Un cursor es un objeto que permite interactuar con una base de datos. Mediante un cursor se puede recorrer un conjunto de filas, insertar nuevas filas, eliminarlas ,etc.

- Sobre este cursor se ejecuta la consulta con el método ***execute(sentencia_select)***. Este método retorna el número de filas devueltas por la ***select***, un valor entero.
- Un cursor dispone de los métodos fetchall() y fetchone(). El primero retorna todas las filas del cursor en forma de tupla de tuplas (cada registro es una tupla). El método fetchone() retorna una tupla con los datos de una fila.
- Mediante un bucle se recorren las filas devueltas. En la imagen siguiente, la variable ***fila*** es una tupla que permite acceder a los registros. Cada valor de la tupla coincide con una columna de la ***select***.

```
1   import pymysql
2
3   try:
4       # Establecer conexión con la base de datos
5       with pymysql.connect(host='localhost', user='user', password='robocop',
6                            port=3309, database='almacen') as conexion:
7
8           # Crear un cursor para ejecutar consultas SQL
9           with conexion.cursor() as cursor:
10
11              # Ejecución de la consulta
12              sql = 'SELECT * FROM clientes'
13              numero_filas = cursor.execute(sql)
14
15              # Recorrer los resultados contenidos en el cursor
16              for cliente in cursor.fetchall():
17                  print(cliente)
18
19  except pymysql.MySQLError as e:
20      print(f'Error en SQL: {e}')
```

En la imagen:

- Línea 5 y 9. Se obtienen la conexión y el cursor. Ojo, por defecto MySQL utiliza el puerto 3306, **si no estás usando docker-compose** deberás usar ese puerto en lugar del que aparece en la imagen (3309). El nombre de usuario, la contraseña y la base de datos deben existir en tu servidor MySQL.
- Línea 12. Se ejecuta el cursor con una SELECT sin parámetros.
- Línea 16 y siguientes. Se recuperan los datos del cursor.

Las excepciones más comunes que pueden darse en esta situación son:

- Error 2003. Si el servidor está parado o existe algún error en la red.
- Error 1045. Acceso denegado para el usuario.
- Error 1064. Error de sintaxis en la sentencia SQL.
- Error 1146. Nombre de tabla inexistente.

Todas estas situaciones están capturadas con la excepción que se observa en el ejemplo, ***pymysql.MySQLError.***

En el ejemplo que sigue se busca a un cliente por su código, si no existe se muestra el mensaje correspondiente. Sería por tanto una sentencia con parámetros y estos se pasan mediante una tupla.

```
1   import pymysql
2
3   try:
4       # Establecer conexión con la base de datos
5       with pymysql.connect(host='localhost', user='user', password='robocop',
6               port=3309, database='almacen') as conexion:
7
8           # Crear un cursor para ejecutar consultas SQL
9           with conexion.cursor() as cursor:
10
11              # Ejecución de la consulta
12              sql = 'SELECT * FROM clientes where numerocliente = %s'
13              cliente_buscado = 100
14              numero_filas = cursor.execute(sql, (cliente_buscado,))
15
16              if numero_filas == 0:
17                  print('No se encontró el cliente')
18              else:
19                  # Acceso a un registro único
20                  cliente = cursor.fetchone()
21                  print(f'Datos del cliente: {cliente}')
22
23  except pymysql.MySQLError as e:
24      print(f'Error en SQL: {e}')
```

- Línea 5 y 9. Se obtienen la conexión y el cursor.
- Línea 12. Se crea la sentencia SQL para realizar la consulta. En este caso se busca a un cliente por su número, lleva por tanto un parámetro (%s).
- Línea 13. Se asigna el valor del cliente buscado a la variable que sustituirá el parámetro.
- Línea 14. El método execute del cursor recibe dos parámetros, la sentencia SQL y la tupla con los valores de los parámetros, en este caso un único valor.
- Línea 19. Se accede a los datos del cursor mediante ***fetchone()***, que dispondrá como mucho de una fila, ya que la búsqueda se realiza mediante la clave principal.

2.3. Insertar registros o filas en una tabla.

Los pasos a seguir para insertar filas en una tabla son similares a los vistos en el apartado anterior. Obtener la conexión y el cursor, y ejecutar la sentencia SQL, en este caso INSERT.

En el ejemplo que se muestra a continuación, se realiza una inserción en la tabla clientes de la base de datos almacén. Se utiliza una sentencia con parámetros y estos se pasan mediante una tupla. **Los parámetros pasados mediante una tupla deben pasarse en el orden en que aparecen en la sentencia.**

```
1   import pymysql
2
3   try:
4       # Establecer conexión con la base de datos
5       with pymysql.connect(host='localhost', user='user', password='robocop',
6               port=3309, database='almacen') as conexion:
7
8           # Crear un cursor para ejecutar consultas SQL
9           with conexion.cursor() as cursor:
10
11              # Sentencia SQL
12              sql = 'insert into clientes values (%s, %s, %s, %s, %s)'
13
14              # Valores a insertar
15              numero_cliente = 1; nombre = 'Juan'; apellido = 'Pérez'; telefono = '12345678'; email = 'jj@an.es'
16
17              # Ejecución de la consulta
18              numero_filas = cursor.execute(sql, (numero_cliente, nombre, apellido, telefono, email,))
19
20              # Confirmar la transacción
21              conexion.commit()
22
23              print(f'Filas insertadas: {numero_filas}')
24
25  except pymysql.MySQLError as e:
26      print(f'Error en SQL: {e}')
```

En el proceso que se muestra:

- Línea 5 y 9. Se obtienen la conexión y el cursor.
- Línea 12. Se crea la sentencia SQL para realizar la inserción con tantos parámetros (%s) como columnas de la tabla a las que se dará valor (en este caso todas).
- Línea 15. Se asignan los valores a las variables que sustituirán a los parámetros de la sentencia.
- Línea 18. Se ejecuta la sentencia SQL a partir del cursor con el método execute. Este método lleva como primer parámetro la sentencia SQL y como segundo una tupla con las variables que ocuparán el lugar de los parámetros.
- Línea 21. Confirmar la transacción.

Las excepciones más comunes que pueden darse en esta situación son:

- Error 2003. Si el servidor está parado o existe algún error en la red.
- Error 1045. Acceso denegado para el usuario.
- Error 1064. Error de sintaxis en la sentencia SQL.
- Error 1062. Si se está duplicando el valor de la clave principal.

2.4. Eliminar registros o filas.

Una operación de eliminación es irreversible, lo que significa que las filas eliminadas no podrán recuperarse. **Sería conveniente pedir confirmación antes de realizar el borrado definitivo.**

La sentencia SQL asociada a la eliminación es **“delete from tabla where condición**”. Con esta sentencia se eliminan todas las filas que satisfacen la condición, y si no se especifica condición, se eliminan todas las filas de la tabla.

En el ejemplo que se muestra a continuación, se intenta eliminar un cliente a partir de su número, clave principal de la tabla. El método ***execute()*** del cursor recibe dos parámetros, la sentencia sql parametrizada y un diccionario con los parámetros de la misma. Observa cómo cambia la forma de indicar el parámetro.

Para confirmar la operación es necesario llamar a ***conexion.commit().***

```
1   import pymysql
2
3
4   try:
5       # Establecer conexión con la base de datos
6       with pymysql.connect(host="localhost", user="root", \
7           password="10nuevo10", database="almacen") as conexion:
8
9
10          # Crear un cursor para ejecutar sentencias SQL
11          with conexion.cursor() as cursor:
12              # Sentencia SQL para eliminar un registro en la tabla de clientes
13              cliente_eliminar = 110
14              sql_delete = "delete from clientes where numerocliente = %(numerocliente)s"
15
16              # Ejecutar una sentencia SQL
17              cursor.execute(sql_delete, {
18                      'numerocliente': cliente_eliminar
19                  })
20              # Confirmar la transacción
21              conexion.commit()
22              print(f"Cliente con número {cliente_eliminar} eliminado")
23  except pymysql.MySQLError as e:
24      print(f"Error: {e}")
```

En la línea 17, el método execute recibe la sentencia SQL y el diccionario con los parámetros. Las claves del diccionario debe coincidir con el nombre asignado al parámetro en la sentencia.

Las excepciones más comunes que pueden darse en esta situación son:

- Error 2003. Si el servidor está parado o existe algún error en la red.
- Error 1045. Acceso denegado para el usuario.
- Error 1064. Error de sintaxis en la sentencia SQL.
- Error 1451. El registro que se pretende eliminar tiene datos relacionados en otra tabla.

2.5. Actualizar registros.

Las operaciones de actualización implican utilizar la sentencia SQL "***update tabla set columna1 = valor1, columna2 = valor2… where condición***". Igual que en las eliminaciones, los cambios realizados con ***update*** son irreversibles. Si no se especifica una condición, la operación afecta a todos los registros de la tabla.

Es necesario confirmar los cambios mediante ***conexion.commit().***

En el ejemplo que sigue, se realiza una actualización del correo electrónico de un cliente. La consulta necesita por tanto, dos parámetros, el número del cliente que se va a actualizar y el nuevo correo electrónico.

```python
import pymysql


try:
    # Establecer conexión con la base de datos
    with pymysql.connect(host="localhost", user="root", \
        password="10nuevo10", database="almacen") as conexion:

        # Crear un cursor para ejecutar sentencias SQL
        with conexion.cursor() as cursor:
            # Sentencia SQL para actualizar el correo de un cliente
            cliente_actualizar = 111
            nuevo_correo = 'corredor@popular.es'
            sql_update = "update clientes set correo = \
                %(correo)s where numerocliente = %(numerocliente)s"

            # Ejecutar una sentencia SQL
            cursor.execute(sql_update, {
                    'correo': nuevo_correo,
                    'numerocliente': cliente_actualizar
                })

            # Confirmar la transacción
            conexion.commit()
            print(f"Correo actualizado para el cliente con número {cliente_actualizar}")

except pymysql.MySQLError as e:
    print(f"Error: {e}")
```

Las excepciones más comunes que pueden darse en esta situación son:

- Error 2003. Si el servidor está parado o existe algún error en la red.
- Error 1045. Acceso denegado para el usuario.
- Error 1064. Error de sintaxis en la sentencia SQL.
- Error 1451. No se puede modificar el valor de un campo clave principal que tiene registros relacionados en otra tabla.

3. El patrón de diseño Singleton.

El patrón de diseño Singleton nos asegura que solo exista una instancia de una clase concreta. Como sabéis, la creación de conexiones con la base de datos es un proceso costoso en cuanto a recursos y no sería conveniente realizarlo constantemente en nuestras aplicaciones.

La filosofía de trabajo de este patrón de diseño nos permitirá obtener una conexión única con la base de datos para nuestra aplicación y utilizarla siempre que deseemos lanzar alguna sentencia SQL contra la base de datos..

En la imagen siguiente se muestra un ejemplo de utilización de este patrón de diseño.

```
1   # Clase que se encarga de la conexión a la base de datos mediante el patrón Singleton
2   import pymysql
3
4   class ConectarBd:
5
6       __conexion = None
7
8       @classmethod
9       def get_conexion(cls):
10          try:
11              if ConectarBd.__conexion == None:
12                  ConectarBd.__conexion = pymysql.connect(host='localhost', user='user',
13                                                          port=3309, passwd='robocop', database='almacen')
14              return ConectarBd.__conexion
15          except Exception as e:
16              return False
17
18      @classmethod
19      def close_conexion(cls):
20          try:
21              if ConectarBd.__conexion != None:
22                  ConectarBd.__conexion.close()
23                  ConectarBd.__conexion = None
24              return True
25          except Exception as e:
26              return False
```

En la imagen tenemos:

- Línea 4. Clase ConectarBd. Su función es la de obtener una conexión con la base de datos ***almacén***.
- Línea 6. Declaración de una variable de clase para almacenar la conexión.
- Línea 9. Declaración del método get_conexion() que retorna la conexión con la base de datos. En la primera llamada al método se crea la conexión y se retorna, en las restantes se devuelve la conexión creada.
- Línea 19. Declaración del método cerrar la conexión.

Con el siguiente código podemos probar que efectivamente, al llamar varias veces al método ***get_conexion()*** obtenemos el mismo objeto.

```
conexion1 = ConectarBd().get_conexion()
conexion2 = ConectarBd().get_conexion()
print(id(conexion1))
print(id(conexion2))
```

```
2519195217232
2519195217232
```

4. Acceso a bases de datos mediante un sistema ORM.

Un sistema ORM (Object-Relational Mapping) permite tratar como objetos los registros guardados en una base de datos relacional. **Cada registro o fila de una tabla se corresponde con un objeto.** Para llevar a cabo esta tarea, **cada tabla de la base de datos está mapeada a una clase** de la aplicación que se desarrolla.

En esta unidad trabajaremos con el sistema ORM de Django. Django es un framework para desarrollar aplicaciones web en Python y dispone de un módulo para el acceso a bases de datos mediante un sistema ORM propio.

Los conocimientos adquiridos en esta unidad nos servirán para entender más "fácilmente" el diseño de aplicaciones web con Django.

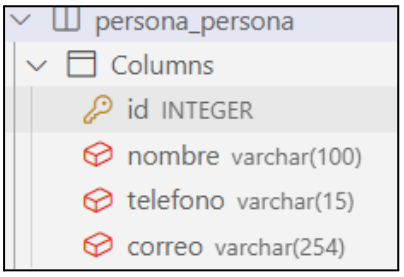
4.1. Modelos.

Un modelo de Django es una **clase** cuya descripción se corresponde con la estructura de una tabla de la base de datos del proyecto. Esta clase se crea en el archivo ***models.py*** de la aplicación que se está desarrollando.

Las imágenes siguientes muestran un ejemplo de modelo que dará lugar a la creación de una tabla llamada Persona en la base de datos. Los atributos de la clase Persona se convertirán en las columnas de la tabla Persona de la base de datos. La clave principal, id, se añade de forma automática.

```
class Persona(models.Model):
    nombre = models.CharField(max_length=100)
    telefono = models.CharField(max_length=15)
    correo = models.EmailField()

    def __str__(self):
        return f"{self.nombre} - {self.telefono}
            - {self.correo}"
```



En los siguientes apartados de la unidad se estudiarán las operaciones más comunes que pueden llevarse a cabo con el ORM de Django.

Ten en cuenta que para llevar a la práctica estos ejemplos debes:

- **Crear un proyecto Django tal y como se detalla en el Anexo VI. El proyecto Django que desarrollaremos se llamará "test_django_orm" y la aplicación "persona".**
- **Este proyecto Django solo sirve para probar el ORM de Django, por eso cada script lleva las líneas de la imagen siguiente.**

```
1  import os
2  import django
3
4  os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'test_django_orm.settings')
5  django.setup()
```

- **Utilizar como motor de base de datos SQLite (opción por defecto).**
- **Instalar algún cliente para SQLite. Puede servirnos la extensión de VSC Database Cliente JDBC comentada al principio de la unidad.**
- **Un aspecto muy importante a tener en cuenta es tener seleccionado en VSC, el intérprete que ejecutará nuestro código, que no es otro que el del entorno virtual donde hemos instalado Django. Para hacer esto debes pulsar CTRL+SHIFT+P y seleccionar el intérprete.**

4.2. Crear una tabla en una base de datos.

Para crear una tabla se accede al archivo ***models.py*** de la aplicación y se crea la clase que la mapea. Abajo en el apartado enlaces de interés tienes más información sobre los tipos de datos que pueden utilizarse en los modelos.

Para desarrollar los ejemplos que siguen, crearemos el modelo ***Persona.*** Esta clase dará lugar a la tabla ***persona_persona*** en la base de datos. Por defecto Django crea una columna id como clave primaria de la tabla asociada al modelo.

```python
from django.db import models

# Create your models here.


class Persona(models.Model):
    nombre = models.CharField(max_length=100)
    telefono = models.CharField(max_length=15)
    correo = models.EmailField()

    def __str__(self):
        return f"{self.nombre} - {self.telefono} - {self.correo}"
```

Hecho esto se deben aplicar los cambios del modelo en la base de datos, a esto se le llama ***migraciones***. Para ello accede a un Terminal donde tengas activado el entorno virtual y teclea:

- ***python manage.py makemigrations***
- ***python manage.py migrate***

Tras esta operación se habrá creado la base de datos SQLite con nombre por defecto ***db.sqlite3*** y dentro de ella la tabla ***persona_persona.*** Es el momento de acceder a la extensión **Database Cliente JDBC,** crear una conexión nueva y abrir el archivo ***db.sqlite3*** para probar si todo va según lo esperado.

Para cambiar el nombre de la base de datos debe editarse el archivo ***settings.py*** del proyecto, buscar el diccionario ***DATABASES*** y cambiar el nombre por defecto ***(db.sqlite3***) por otro más afín al proyecto.

4.3. Insertar filas en una tabla.

Para ello debemos crear un archivo Python y teclear el código Python que permita insertar filas en la tabla creada.

Las sentencias son sencillas: crear varias instancias de Persona y llamar al método save() del objeto.

```python
import os
import django

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'test_django_orm.settings')
django.setup()

from persona.models import Persona

personas = [
    Persona(nombre='Juan', telefono='123456789', correo='juan@gmail.com'),
    Persona(nombre='Ana', telefono='987654321', correo='ana@gmail.com'),
    Persona(nombre='Pedro', telefono='456789123', correo='peter@pan.es'),
    Persona(nombre='María', telefono='654987321', correo='maria@cuetara.es'),
    Persona(nombre='Luis', telefono='789123456', correo='luis@ito.es')
]

for persona in personas:
    persona.save()
    print(f"Persona {persona.nombre} guardada en la base de datos")
```

4.4. Obtener todos los registros de una tabla.

Método: ***Nombre_Modelo.objects.all().*** Retorna un objeto ***QuerySet*** con el conjunto de registros devueltos. Un QuerySet es un conjunto de objetos de un modelo, en este caso un conjunto de objetos Persona. Este QuerySet puede recorrerse mediante un bucle for.

```python
import os
import django

os.environ.setdefault('DJANGO_SETTINGS_MODULE',
                      'test_django_orm.settings')
django.setup()

from persona.models import Persona

# Obtener todas las instancias de la clase Persona
personas = Persona.objects.all()

# Mostrar las instancias de la clase Persona
for persona in personas:
    print(persona)
```

4.5. Obtener un único registro.

Método: ***Nombre_Modelo.objects.get(atributo=valor).*** Retorna el objeto con ese valor en el atributo. Si no existe lanza una excepción ***DoesNotExists***. Si existen varios

registros con ese valor lanza una excepción ***MultipleObjectsReturned***. Es habitual utilizar la clave principal (id) para las búsquedas.

```
1   import os
2   import django
3
4   os.environ.setdefault('DJANGO_SETTINGS_MODULE',
5                         'test_django_orm.settings')
6   django.setup()
7
8   from persona.models import Persona
9
10  # Buscar por id
11  persona = Persona.objects.get(id=1)
12  print(persona)
```

4.6. Filtros.

Método: ***Nombre_Modelo.objects.filter(condición).*** Este método retorna un objeto ***QuerySet*** con los objetos que satisfacen la condición. Las condiciones sólo permiten valores exactos (operador =). Pueden separarse varias condiciones separadas por comas (and).

```
1   import os
2   import django
3
4   os.environ.setdefault('DJANGO_SETTINGS_MODULE',
5                         'test_django_orm.settings')
6   django.setup()
7
8   from persona.models import Persona
9
10  # Filtrar por nombre
11  personas= Persona.objects.filter(nombre='Juan')
12
13  for persona in personas:
14      print(persona)
```

4.7. Combinar condiciones con AND y OR.

En este caso se combinan los filtros mediante **& (and) y | (or)**.

```python
import os
import django

os.environ.setdefault('DJANGO_SETTINGS_MODULE',
                      'test_django_orm.settings')
django.setup()

from persona.models import Persona

# Filtrar por nombre con OR-> |
personas= Persona.objects.filter(nombre='Juan') \
    | Persona.objects.filter(nombre='Ana')

for persona in personas:
    print(persona)
```

4.8. Exclude.

Método: ***Nombre_Modelo.objects.exclude(condición).*** Este método retorna un objeto ***QuerySet*** con los elementos **que no cumplen la condición**.

```python
import os
import django

os.environ.setdefault('DJANGO_SETTINGS_MODULE',
                      'test_django_orm.settings')
django.setup()

from persona.models import Persona

# Filtrar por nombre con exclude
personas= Persona.objects.exclude(nombre='Juan')

for persona in personas:
    print(persona)
```

4.9. Búsquedas de campo (field lookups).

Se utilizan con el método filter() y su formato es ***filter(atributo__condicion=valor).*** Algunos ejemplos son: __startswith, __endswith, __contains, __gt, __gte, __lt, __lte, __year, __day, etc.

En el ejemplo se muestran las personas cuyo correo contiene el valor ***gmail***.

```
1   import os
2   import django
3
4   os.environ.setdefault('DJANGO_SETTINGS_MODULE',
5                         'test_django_orm.settings')
6   django.setup()
7
8   from persona.models import Persona
9
10  # Búsquedas de campo - field lookups.
11  personas= Persona.objects.filter(correo__contains='gmail')
12
13  for persona in personas:
14      print(persona)
```

4.10. Limitar el número de registros.

Para limitar el número de registros devueltos se aplica ***slicing al QuerySet.*** En el ejemplo se seleccionan los tres primeros registros.

```
1   import os
2   import django
3
4   os.environ.setdefault('DJANGO_SETTINGS_MODULE',
5                         'test_django_orm.settings')
6   django.setup()
7
8   from persona.models import Persona
9
10  # Limitar la cantidad de registros a obtener
11  personas= Persona.objects.all()[:2]
12
13  for persona in personas:
14      print(persona)
```

4.11. Ordenar los registros devueltos.

Método: ***order_by('atributo').*** Si el orden es inverso se utiliza ***-atributo***. Puede ordenarse por varios atributos. El orden se aplica a un ***QuerySet.***

```
1   import os
2   import django
3
4   os.environ.setdefault('DJANGO_SETTINGS_MODULE',
5                         'test_django_orm.settings')
6   django.setup()
7
8   from persona.models import Persona
9
10  # Ordenar por nombre
11  personas= Persona.objects.all().order_by('nombre')
12
13  for persona in personas:
14      print(persona)
```

4.12. Eliminar registros.

Método: ***delete().*** Se puede aplicar a una instancia del modelo o al modelo completo, lo que daría lugar a eliminar todas las filas de la tabla. Mediante ***modelo.objects.all().delete()*** elimina todos los objetos/registros de la tabla asociada al modelo.

Es conveniente solicitar confirmación antes de llevar a cabo eliminaciones.

```
1   import os
2   import django
3
4   os.environ.setdefault('DJANGO_SETTINGS_MODULE',
5                         'test_django_orm.settings')
6   django.setup()
7
8   from persona.models import Persona
9
10  # Buscar por id y eliminar
11  persona = Persona.objects.get(id=1)
12  persona.delete()
13  print("Registro eliminado")
```

4.13. Uniones.

Método: ***union()***. Mediante este método se obtiene un ***QuerySet*** como resultado de la unión de varios. Este resultado elimina las repeticiones. Añadiendo ***all=True***, se mantienen los objetos repetidos en el ***QuerySet*** final.

```
import os
import django

os.environ.setdefault('DJANGO_SETTINGS_MODULE',
                      'test_django_orm.settings')
django.setup()

from persona.models import Persona

# Uniones
personas= Persona.objects.filter(nombre='Luis')
personas_gmail = Persona.objects.filter(correo__contains='gmail')

result = personas.union(personas_gmail)

for persona in result:
    print(persona)
```

4.14. Funciones de grupo.

Métodos: Avg(), Max(), Min(), Sum(), Count(). Para usar estos métodos es necesario realizar la importación ***from django.db.models import Avg, Count, Max, Min, Sum.*** Se persigue aplicar estas funciones a una columna concreta.

Estos métodos retornan un diccionario. Estudia la imagen con el resultado de la salida.

```
import os
import django
from django.db.models import Avg, Count, Max, Min, Sum

os.environ.setdefault('DJANGO_SETTINGS_MODULE',
                      'test_django_orm.settings')
django.setup()

from persona.models import Persona

# Funciones agregado
mayor_id = Persona.objects.aggregate(Max('id'))
menor_id = Persona.objects.aggregate(Min('id'))
contar = Persona.objects.filter(correo__contains = 'gmail').aggregate(Count('id'))
promedio = Persona.objects.aggregate(Avg('id'))

print(f'Mayor id: {mayor_id}')
print(f'Menor id: {menor_id}')
print(f'Contar: {contar}')
print(f'Promedio: {promedio}')
```

```
Mayor id: {'id__max': 5}
Menor id: {'id__min': 2}
Contar: {'id__count': 1}
Promedio: {'id__avg': 3.5}
```

4.15. Obtener una serie de atributos de un modelo.

Método: values_list() . Permite obtener una serie de atributos del modelo y no el modelo completo. Los nombres de los atributos se pasan como argumentos a este método. Este método retorna un QuerySet formado por tuplas con los valores de los datos.

Ejemplo: Persona.objects.values_list(nombre, 'telefono').

El método values() realiza la misma operación que values_list(), salvo que retorna los datos como un diccionario.

El método distinct() suele acompañar a values_list()/values() para eliminar repeticiones en un QuerySet.

Ejemplo: Persona.objects.values_list(nombre).distinct().

```python
import os
import django

os.environ.setdefault('DJANGO_SETTINGS_MODULE',
                      'test_django_orm.settings')
django.setup()

from persona.models import Persona

# Obtener solo algunos atributos de las instancias de la clase Persona
personas = Persona.objects.values_list('nombre').distinct()

# Mostrar las instancias de la clase Persona
for persona in personas:
    print(persona)
```

(module) test_django_orm

4.16. Consultas con agrupamiento y totales.

Con estas consultas se crean grupos de datos con una información en común y a estos grupos se les aplican las funciones de grupo (max, min, avg, sum, count).

En el ejemplo, se busca contar cuántas veces se repite cada nombre. Con value_list('nombre') se extrae solo el atributo nombre y mediante el método annotate se aplica la función de grupo Count a ese grupo de atributos.

```python
import os
import django

from django.db.models import Count

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "test_django_orm.settings")
django.setup()

from personas.models import Persona

print("Agrupar y contar")

result = Persona.objects.values_list('nombre').annotate(total=Count('nombre'))

for tupla in result:
    print(tupla)
```

5. Enlace de interés.

- https://medium.com/@simeon.emanuilov/orm-vs-raw-sql-in-python-frameworks-1bb067d49fc8
- https://docs.djangoproject.com/en/5.1/topics/db/models/
- https://docs.djangoproject.com/en/5.1/ref/models/fields/#field-types.
- https://docs.djangoproject.com/en/5.1/topics/db/queries/
- https://docs.djangoproject.com/en/5.1/topics/db/queries/#field-lookups
- https://docs.djangoproject.com/en/5.1/topics/db/examples/