



Creación y manejo de modelos en Django

Aprendizaje Esperado 2 – Bootcamp Full Stack Python

¿Qué son los modelos en Django?

Un modelo en Django es la representación en Python de una tabla en la base de datos. Nos permite trabajar con los datos como objetos, sin escribir SQL directamente.

Django convierte las clases que definimos en estructuras de tablas mediante su ORM (Object-Relational Mapper). Esta abstracción poderosa elimina la necesidad de escribir consultas SQL manualmente, haciendo que el desarrollo sea más rápido y seguro.



Pregunta de inicio: ¿Qué tablas creen que tendría una aplicación de ventas de bicicletas?



¿Qué aprenderemos hoy?

En esta sesión exploraremos los fundamentos de los modelos en Django y cómo trabajar eficientemente con bases de datos relacionales.

1

Fundamentos de Modelos

Entender qué es un modelo en Django y cómo se relaciona con la base de datos mediante el ORM

2

Definición de Campos

Crear modelos y definir campos, tipos de datos y opciones de configuración avanzadas

3

Claves y Restricciones

Manejar claves primarias, campos únicos y restricciones de integridad

4

Operaciones CRUD

Realizar operaciones de Crear, Leer, Actualizar y Borrar registros

5

Práctica Real

Aplicar conceptos con un ejemplo completo del proyecto Bikeshop

El concepto de Modelo

Cada modelo es una clase de Python que hereda de `models.Model`. Esta herencia proporciona toda la funcionalidad necesaria para interactuar con la base de datos de forma automática.

Código Python

```
from django.db import models

class Bicicleta(models.Model):
    marca = models.CharField(
        max_length=50
    )
    modelo = models.CharField(
        max_length=50
    )
    precio = models.DecimalField(
        max_digits=10,
        decimal_places=2
    )
```

Resultado en la BD

Django traduce esta clase en una tabla SQL automáticamente, creando columnas para cada campo definido.



El nombre de la tabla será `nombre_app_bicicleta` siguiendo las convenciones de Django.

El ORM de Django

El ORM (Object-Relational Mapper) es el puente que convierte nuestras operaciones en Python en consultas SQL optimizadas. Esto significa que podemos trabajar con objetos Python en lugar de escribir SQL directamente.

En Python escribes:

```
Bicicleta.objects.all()
```

**Código orientado a objetos,
limpio y Pythonic**

El ORM ejecuta:

```
SELECT * FROM  
bicicletas_bicicleta;
```

**Consulta SQL optimizada
automáticamente**

✓ **Sin SQL manual**

No necesitas escribir
consultas SQL complejas

✓ **Código legible**

Sintaxis clara y fácil de
mantener

✓ **Multi-motor**

Compatible con MySQL, SQLite, PostgreSQL

```
ecalunten)  
nct_)  
est #)  
tote=  
BC1)
```

```
e==  
tes==  
D==)  
#LH-C#11  
ta( )  
==
```

```
etf#)  
-clounlc  
-cã te'  
t:enl  
)  
-(toJ-
```



Tipos de campos en modelos

Cada campo representa una columna en la tabla. Django ofrece múltiples tipos de campos que se mapean automáticamente a tipos de datos SQL apropiados.



CharField

```
CharField(max_length=50)
```

Para texto corto como nombres, títulos o códigos

1

IntegerField

```
IntegerField()
```

Números enteros, ideal para cantidades o años

10

DecimalField

```
DecimalField(max_digits=10, decimal_places=2)
```

Números decimales precisos para precios o medidas



BooleanField

```
BooleanField(default=True)
```

Valores verdadero/falso para estados o banderas



DateTimeField

```
DateTimeField(auto_now_add=True)
```

Fechas y horas, con opciones de autocompletado



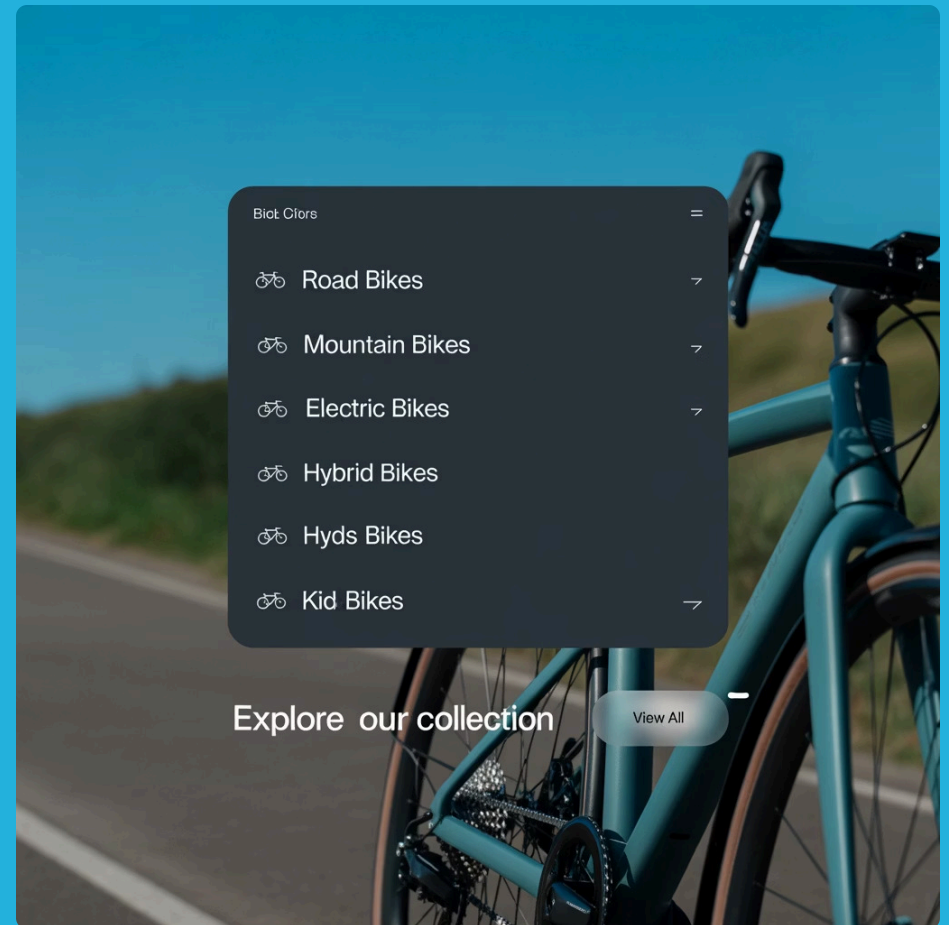
Ejercicio guiado: En VS Code, define 3 campos nuevos en el modelo Bicicleta: **tipo**, **año** y **disponible**.

Opciones en los campos

Los campos de Django aceptan múltiples opciones que controlan su comportamiento, validación y presentación. Estas opciones nos permiten crear modelos robustos con reglas de negocio incorporadas.

Ejemplo práctico:

```
tipo = models.CharField(  
    max_length=20,  
    choices=[  
        ('mtb', 'Montaña'),  
        ('ruta', 'Ruta'),  
        ('urbana', 'Urbana')  
    ],  
    default='mtb'  
)
```



max_length

Define la longitud máxima para campos de texto



default=valor

Establece un valor predeterminado cuando no se especifica



null=True

Permite valores NULL en la base de datos



choices=[(key, label)]

Limita las opciones disponibles a una lista predefinida



Claves primarias

Cada tabla necesita una clave primaria (Primary Key) que identifica de forma única cada registro. Esta es fundamental para mantener la integridad referencial de la base de datos.

Comportamiento predeterminado

Por defecto, Django crea automáticamente un campo `id` que funciona como clave primaria autoincremental. Este campo se genera sin necesidad de declararlo explícitamente.

```
id = models.AutoField(  
    primary_key=True  
)
```

Clave primaria personalizada

También puedes definir tu propia clave primaria usando cualquier campo único:

```
codigo = models.CharField(  
    max_length=10,  
    primary_key=True  
)
```



Reflexión activa: ¿Por qué es importante tener una clave única en cada registro? ¿Qué problemas podría causar no tenerla?

Campos únicos

Para asegurar que ciertos valores no se repitan en la base de datos, utilizamos la opción `unique=True`. Esto es especialmente útil para emails, nombres de usuario, códigos de productos, o cualquier dato que deba ser exclusivo.

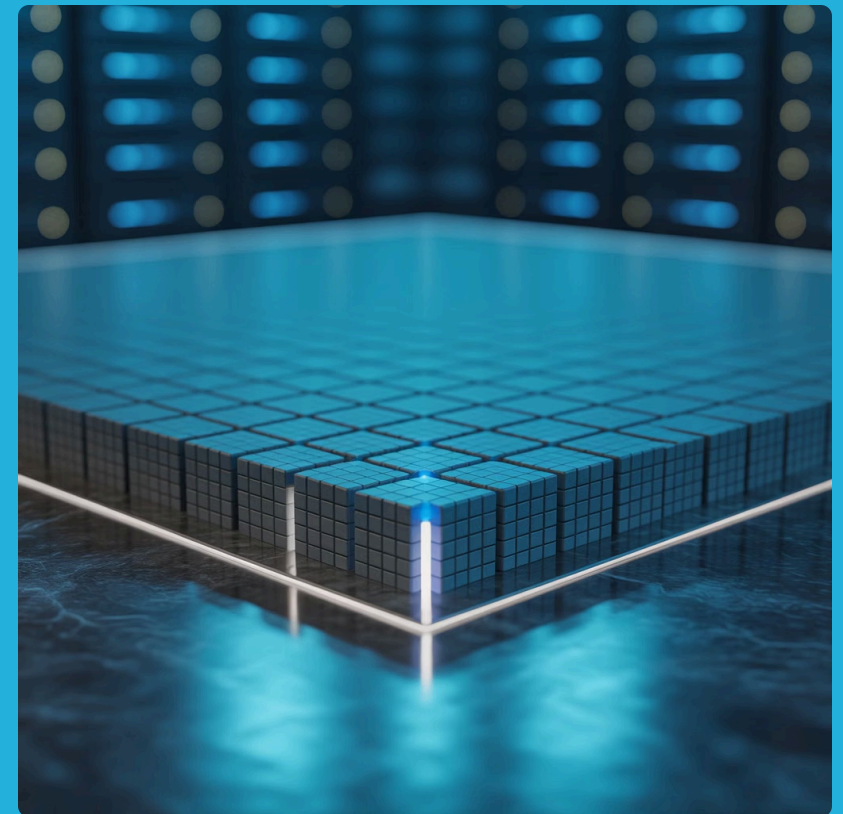
Ejemplo de implementación:

```
email = models.EmailField(  
    unique=True,  
    error_messages={  
        'unique': 'Este email ya está registrado'  
    }  
)
```

Esto garantiza que no haya dos usuarios con el mismo correo electrónico. Django automáticamente valida esta restricción tanto a nivel de aplicación como de base de datos.



Actividad: Haz que el campo `modelo` de la Bicicleta sea único, y ejecuta las migraciones correspondientes.



Claves compuestas con unique_together

Cuando necesitamos que la combinación de dos o más campos sea única, utilizamos `unique_together` en la clase Meta del modelo. Esta restricción es ideal para evitar duplicados en relaciones complejas.

```
class Venta(models.Model):
    bicicleta = models.ForeignKey(
        'Bicicleta',
        on_delete=models.CASCADE
    )
    cliente = models.CharField(
        max_length=50
    )
    fecha = models.DateField(
        auto_now_add=True
    )

    class Meta:
        unique_together = (
            'bicicleta',
            'cliente'
        )
```

Con esta configuración, un mismo cliente no puede tener dos ventas de la misma bicicleta, pero puede comprar diferentes modelos.

Esta restricción previene ventas duplicadas accidentales y mantiene la coherencia de los datos en el sistema.





Migraciones

Las migraciones son el mecanismo de Django para propagar cambios en los modelos a la estructura de la base de datos. Son archivos Python que describen cómo modificar el esquema de manera controlada y reversible.



1. makemigrations

Crea archivos de migración basados en cambios en `models.py`

```
python manage.py makemigrations
```



2. migrate

Aplica las migraciones pendientes a la base de datos

```
python manage.py migrate
```




3. Verificar

Confirma los cambios en la base de datos

```
python manage.py dbshell
```



 **Ejercicio:** Crea la migración para el modelo `Bicicleta` y observa cómo se genera la tabla en MySQL.

Crear registros (C de CRUD)

La creación de registros en Django se realiza de forma intuitiva utilizando el método `create()` del manager de objetos. Este método inserta un nuevo registro en la base de datos sin necesidad de escribir SQL.

Método 1: `create()`

```
from bicicletas.models import
Bicicleta

Bicicleta.objects.create(
    marca="Trek",
    modelo="Marlin 7",
    tipo="mtb",
    precio=799.99,
    disponible=True,
    año=2023
)
```

Este método crea y guarda el objeto en un solo paso, retornando la instancia creada.

Método 2: `save()`

```
bici = Bicicleta(
    marca="Specialized",
    modelo="Rockhopper",
    tipo="mtb",
    precio=899.99
)
bici.save()
```

Primero creas la instancia y luego llamas a `save()` para persistirla.

Leer registros (R de CRUD)

Django ofrece una API rica y expresiva para consultar datos. El método `objects` proporciona múltiples opciones para recuperar información de la base de datos de forma eficiente.

`all()` – Todos los registros

```
Bicicleta.objects.all()
```

Retorna un QuerySet con todas las bicicletas

`filter()` – Filtrar

```
Bicicleta.objects.filter(  
    disponible=True,  
    tipo='mtb'  
)
```

Retorna registros que cumplen las condiciones

`get()` – Un registro específico


```
Bicicleta.objects.get(  
    id=1  
)
```

Retorna un único objeto (error si hay 0 o más de 1)

`order_by()` – Ordenar

```
Bicicleta.objects.order_by(  
    '-precio'  
)
```

Ordena por precio descendente (- indica DESC)

 **Ejercicio:** Obtén las bicicletas disponibles de tipo 'mtb' y ordénalas por año de más reciente a más antiguo.

Actualizar registros (U de CRUD)

Django proporciona dos enfoques principales para actualizar registros: modificar instancias individuales o actualizar múltiples registros simultáneamente usando `update()`.

Actualización de instancia única


```
bici = Bicicleta.objects.get(
    id=1
)
bici.precio = 850.00
bici.disponible = False
bici.save()
```

Este método primero recupera el objeto, lo modifica y luego lo guarda. Útil cuando necesitas procesar lógica adicional.

Actualización masiva

```
Bicicleta.objects.filter(
    tipo='mtb'
).update(
    precio=850,
    disponible=True
)
```

Este método actualiza múltiples registros de una vez, sin cargarlos en memoria. Más eficiente para operaciones masivas.

📝  **Actividad:** Modifica el precio de la bicicleta que creaste anteriormente, aumentándolo un 10%.



Eliminar registros (D de CRUD)

La eliminación de registros en Django debe hacerse con precaución. Django ofrece métodos seguros que permiten borrar tanto registros individuales como grupos de registros que cumplan ciertos criterios.

Eliminar un registro específico

```
bici = Bicicleta.objects.get(  
    id=2  
)  
bici.delete()
```

Retorna una tupla con el número de objetos eliminados y un diccionario con los detalles.

Eliminación múltiple con filtro

```
Bicicleta.objects.filter(  
    disponible=False,  
    año__lt=2020  
)delete()
```

Elimina todas las bicicletas no disponibles fabricadas antes de 2020.



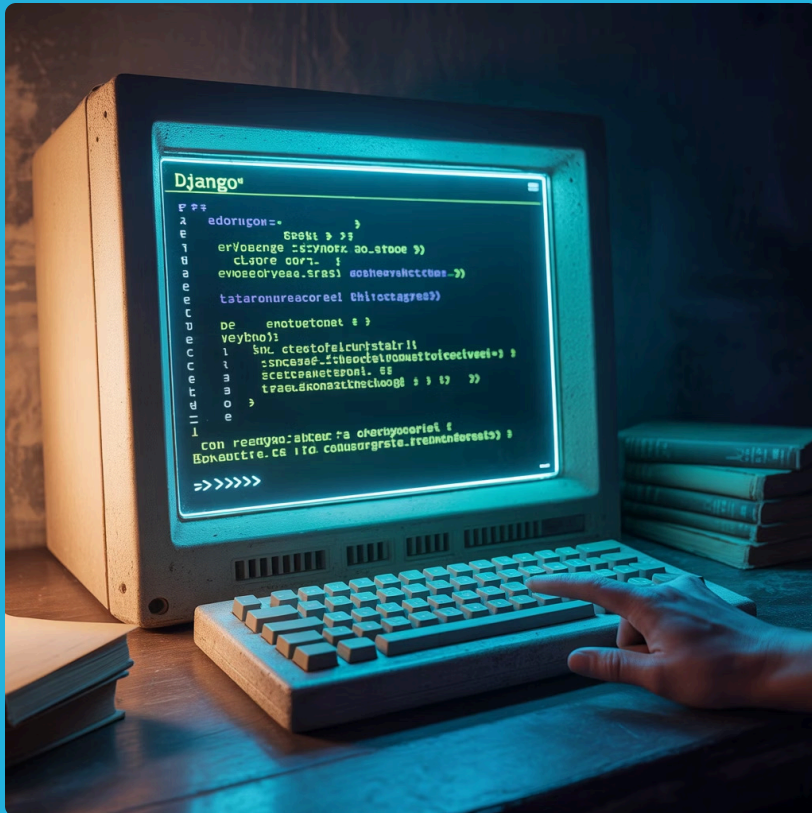
ⓘ ⚠ **Precaución:** La eliminación es permanente. Siempre verifica tus filtros antes de ejecutar `delete()`.

Django Shell: nuestro laboratorio

Django Shell es una consola interactiva de Python con acceso completo a tu proyecto Django. Es una herramienta invaluable para probar código, explorar datos y depurar problemas sin necesidad de crear vistas o templates.


Iniciar el shell

```
python manage.py shell
```



Ventajas del Shell

- Prueba CRUD rápidamente sin vistas ni plantillas
- Experimenta con consultas complejas antes de implementarlas
- Depura problemas con datos en tiempo real
- Importa y prueba funciones de tus modelos
- Ejecuta scripts de mantenimiento de datos

📝  **Actividad práctica:** Desde el shell, realiza el ciclo completo: crea una bicicleta, consúltala, actualiza su precio y finalmente bórrala. Observa el resultado de cada operación.

Validación y depuración

Antes de implementar vistas complejas, es crucial probar y validar la lógica de tus modelos en el shell. Esto te permite detectar errores temprano y comprender el comportamiento real de tus queries.



Contar registros

```
Bicicleta.objects.all().count()
```

Verifica cuántos registros existen sin cargarlos todos en memoria



Inspeccionar queries

```
print(Bicicleta.objects.filter(
    tipo='mtb'
).query)
```

Visualiza el SQL generado por el ORM para optimizar consultas

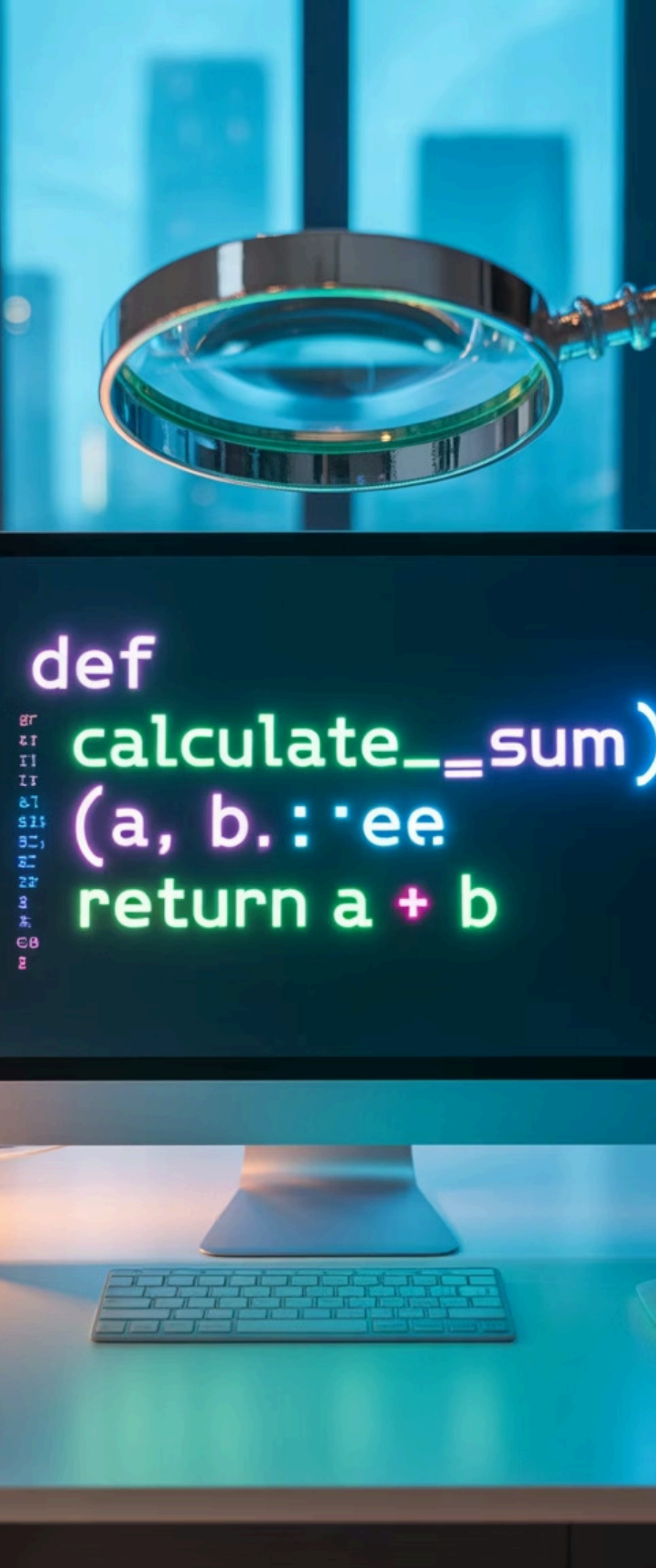


Método `__str__()`

```
def __str__(self):
    return f"{self.marca} {self.modelo}"
```

Define una representación legible de tus objetos para depuración

Usa `print()` generosamente durante el desarrollo para entender el flujo de datos y detectar problemas antes de que lleguen a producción.



Actividad práctica colaborativa

Taller Bikeshop

En esta actividad aplicaremos todo lo aprendido trabajando en equipos. Seguirán un flujo completo de desarrollo desde la modificación del modelo hasta la manipulación de datos.

01	02	03
Agregar campo nuevo Añadan al modelo Bicicleta un nuevo campo llamado <code>color</code> de tipo CharField con <code>max_length=30</code>	Ejecutar migraciones Creen y apliquen las migraciones necesarias para actualizar la base de datos	Insertar registros Desde el shell, inserten al menos 3 bicicletas con colores diferentes (rojo, azul, negro)
04	05	
Consultar por color Filtren todas las bicicletas de un color específico y muestren los resultados	Actualizar precios Aumenten el precio de todas las bicicletas rojas en un 15%	

Tiempo estimado: 20-25 minutos | Modalidad: Equipos de 2-3 personas

Buenas prácticas

Seguir buenas prácticas desde el inicio te ahorrará horas de refactorización y te ayudará a crear aplicaciones más mantenibles y escalables.

✓ Nombres descriptivos

Usa nombres claros para campos y modelos que reflejen su propósito. Prefiere `fecha_compra` sobre `fecha` cuando haya múltiples fechas.

✓ Valores por defecto

Define defaults cuando sea posible para evitar errores y simplificar la creación de objetos.

Ejemplo:

```
activo=models.BooleanField(default=True)
```

✓ Evita duplicación

No repitas información que puede derivarse o calcularse. Usa propiedades o métodos en lugar de campos redundantes.

✓ Usa el ORM

Aprovecha el ORM en lugar de SQL directo para mantener portabilidad entre bases de datos y aprovechar las optimizaciones de Django.

✓ Verifica migraciones

Siempre revisa los archivos de migración generados antes de aplicarlos. Usa `sqlmigrate` para ver el SQL que se ejecutará.

✓ Documenta modelos

Incluye docstrings en tus modelos explicando su propósito y relaciones importantes.

Conclusiones

Hemos recorrido los conceptos fundamentales de los modelos en Django, la columna vertebral del desarrollo de aplicaciones robustas y escalables.

Modelos = Corazón

Los modelos son el corazón del acceso a datos en Django, definiendo la estructura y comportamiento de tu información

ORM Simplifica



El ORM simplifica la interacción con la base de datos, traduciendo Python a SQL optimizado automáticamente

Migraciones

Las migraciones mantienen sincronizados tus modelos con el esquema de la base de datos de forma controlada

CRUD Fácil

El CRUD permite manipular datos fácilmente desde Python con una sintaxis intuitiva y expresiva

  Reflexión final: ¿Cómo cambia tu forma de trabajar con bases de datos después de conocer el ORM?
¿Qué ventajas ves en este enfoque orientado a objetos?