Unidad didáctica 03

Gestión de la persistencia de datos con ORM

Índice

UD 03: Gestión de la Persistencia de los Datos con ORM

1. Introducción a los ORMs (Mapeo Objeto-Relacional)

El mapeo objeto-relacional (ORM) es una técnica que permite interactuar con bases de datos relacionales utilizando objetos y clases de un lenguaje de programación en lugar de escribir consultas SQL directamente. Con los ORMs, podemos trabajar de manera más cercana a la lógica de programación orientada a objetos, y al mismo tiempo, manejar la persistencia de los datos en una base de datos relacional.

ORMs como **Peewee** facilitan mucho el trabajo con bases de datos, ya que permiten realizar operaciones comunes, como la inserción, actualización y eliminación de datos, de una manera más intuitiva y con menos errores, ya que abstraen la mayoría de los detalles específicos de SQL.

Por ejemplo, en lugar de escribir consultas SQL como `INSERT INTO herramientas (nombre, tipo, marca) VALUES (...)`, podemos usar **Peewee** para escribir algo como `Herramienta.create( nombre = "Martillo" , tipo = "Manual" , marca= "Truper")`. Esto no solo simplifica el código, sino que también lo hace más legible y mantenible.

Además, los ORMs ofrecen una manera estándar de trabajar con diferentes bases de datos, lo que hace que cambiar de una base de datos a otra sea menos doloroso. En este tema trabajaremos con **Peewee**, un ORM ligero y flexible que es muy fácil de usar y se integra bien con **MySQL**.

Recordemos **la necesidad de trabajar con virtualenv** para cada una de las actividades de clase que se suceden a lo largo de la presente unidad didáctica.

2. Instalación y configuración de la Herramienta ORM

El primer paso para usar un ORM como Peewee es asegurarnos de que esté instalado junto con el conector de MySQL.

Para instalar Peewee y el conector de MySQL, ejecuta el siguiente comando dentro de tu entorno virtual de reciente creación:

```
pip install peewee pymysql
```

Nótese que la preferencia es usar mymysql como conector junto con peewee, dado que es el mejor soportado por este ORM para conectar con MySQL.

El siguiente paso es configurar el acceso a la base de datos. A continuación mostramos el código que configura una conexión a la base de datos **1dam** (puedes cambiar el nombre de la base de datos según tus necesidades) utilizando **Peewee**:

```
from peewee import MySQLDatabase

# Configurar la base de datos
db = MySQLDatabase(
    '1dam',  # Nombre de la base de datos
    user='usuario',  # Usuario de MySQL
    password='usuario',  # Contraseña de MySQL
    host='localhost',  # Host
    port=3306  # Puerto por defecto de MySQL
)

# Conectar a la base de datos
db.connect()
print("Conexión exitosa a la base de datos.")
```

Explicación:

- **MySQLDatabase**: Este es el objeto principal que representa la conexión a la base de datos. Le pasamos el nombre de la base de datos, usuario, contraseña, host y puerto.
- **db.connect()**: Este método establece la conexión con la base de datos.

3. Definición de Configuraciones de Mapeo

Para que Peewee pueda trabajar con la tabla Herramientas, primero debemos definir una clase que represente dicha tabla en Python. Cada columna de la tabla en la base de datos será representada como un atributo en la clase.

```
from peewee import Model, CharField
# Definir el mapeo de la tabla Herramientas
class Herramienta(Model):
    nombre = CharField()
    tipo = CharField()
    marca = CharField()
    class Meta:
        database = db  # Base de datos
        table_name = 'herramientas'  # Nombre de la tabla en la base de datos
# Crear la tabla si no existe
db.create_tables([Herramienta])
print("Tabla 'Herramientas' creada o ya existente.")
```

Explicación:

- Model: La clase Herramienta hereda de Model de Peewee, lo que significa que mapea a una tabla en la base de datos.
- CharField(): Representa una columna de tipo texto.
- Meta: Esta clase interna le dice a Peewee qué base de datos y qué tabla debe usar para esta clase. ¿Por qué usar una clase interna y no atributos directamente? La clase Meta agrupa todas las configuraciones relacionadas con el modelo en un solo lugar. Esto ayuda a mantener el código más organizado y fácil de entender.

En Peewee, además de CharField(), que representa una columna de tipo texto en la base de datos, hay una variedad de tipos de columnas que puedes usar para mapear diferentes tipos de datos en tus tablas. Estos tipos de columnas corresponden a los tipos de datos más comunes que encontrarás en las bases de datos relacionales.

Aquí tienes un listado exhaustivo de los tipos de columnas más utilizados en Peewee:

- CharField(): Representa una columna de texto. Se utiliza para almacenar cadenas de texto.
- TextField():Similar a CharField(), pero sin límite en la cantidad de caracteres. Se usa para almacenar grandes cantidades de texto.
- IntegerField(): Representa una columna de números enteros.

- BigIntegerField(): Similar a IntegerField(), pero para números enteros más grandes.
- SmallIntegerField(): Representa una columna de enteros pequeños. Ideal para números enteros con un rango más pequeño.
- FloatField(): Representa una columna de números decimales (flotantes).
- DoubleField(): Similar a FloatField(), pero con mayor precisión para números decimales.
- DecimalField(): Utilizado para almacenar números decimales precisos, como montos monetarios. Tiene dos parámetros: max_digits (máximo de dígitos) y decimal_places (cantidad de decimales). Ejemplo: `precio_preciso = DecimalField(max_digits=10, decimal_places=2)`
- BooleanField(): Representa una columna de valor booleano (True o False). Ejemplo: `disponible = BooleanField(default=True)`
- DateField(): Representa una columna de fechas (sin hora).
- DateTimeField(): Similar a DateField(), pero almacena tanto la fecha como la hora.
- TimeField(): Representa una columna de tiempo (solo horas, minutos, y segundos).
- AutoField(): Es una columna entera autoincremental que generalmente se utiliza para claves primarias.
- PrimaryKeyField(): Similar a AutoField(), pero se utiliza explícitamente para indicar la clave primaria de una tabla.
- ForeignKeyField(): Representa una clave foránea que apunta a otra tabla (modelo). Ejemplo: `categoria = ForeignKeyField(Categoria, backref='productos')`
- UUIDField(): Representa un campo que almacena un UUID (Universally Unique Identifier).
- BlobField(): Se utiliza para almacenar datos binarios grandes (BLOB - Binary Large Object), como imágenes o archivos.
- BitField(): Representa una columna que almacena un valor como bits.
- JSONField(): Almacena JSON en una columna de la base de datos.
- ArrayField() (Para Postgres): Almacena una matriz de valores. Este tipo de campo es compatible solo con PostgreSQL. `Ejemplo: etiquetas = ArrayField(CharField)`
- IPField(): Almacena direcciones IP en la base de datos (IPv4 o IPv6).
- CompositeKey(): Se utiliza para crear una clave compuesta de múltiples columnas. Ejemplo:

```
class Meta:
    primary_key = CompositeKey('campo1', 'campo2')
```

4. Mecanismos de Persistencia

Una vez que tenemos la clase que mapea la tabla Herramientas, podemos insertar nuevos registros en la base de datos utilizando la clase Python.

Ejemplo de Inserción de Herramientas en la Base de Datos:

```
# Insertar varias herramientas
Herramienta.create(nombre='Martillo',tipo='Manual',material='Acero',uso='percusión', marca='Truper')
Herramienta.create(nombre='Taladro',tipo='Eléctrico',material='Plástico',uso='Perforación', marca='Bosch')
Herramienta.create(nombre='Sierra', tipo='Manual', material='Acero', uso='Corte', marca='Stanley')
print("Herramientas insertadas en la base de datos.")
```

Explicación:

create(): Este método crea una nueva instancia de la clase y automáticamente la inserta en la base de datos.

5. Ejecución de sentencias SQL directamente con el ORM

A veces el ORM se “nos queda corto” y nos interesa ejecutar una sentencia SQL de forma directa, sin hacer uso de las características de orientación a objeto que nos proporciona el ORM. También puede ser que nos interese cuando buscamos mejoras en rendimiento.

El siguiente ejemplo muestra la creación de una función que detecta mediante SQL si una tabla existe. Luego, si existe la borramos y si no existe sólo se informa.

```
 def tabla_existe(nombre_tabla):

          consulta = "SELECT COUNT(*) FROM information_schema.tables WHERE table_schema = %s AND table_name = %s"

        cursor = db.execute_sql(consulta, ('1dam', nombre_tabla))

        resultado = cursor.fetchone()

        return resultado[0] > 0


    # Eliminamos la tabla si ya existe para empezar a trabajar desde cero

    if tabla_existe(Herramienta._meta.table_name):

        print(f"La tabla '{Herramienta._meta.table_name}' existe.")

        db.drop_tables([Herramienta], cascade=True)

        print(f"Tabla '{Herramienta._meta.table_name}' eliminada con éxito.")

    else:

        print(f"La tabla '{Herramienta._meta.table_name}' no existe.")
```

Algunas Explicaciones:

- En Peewee, no se accede a la clase interna Meta directamente. En su lugar, puedes acceder a las propiedades de la tabla usando _meta.<atributo>, que es parte del sistema de metadatos del modelo. `Herramienta._meta.table_name` es la forma correcta de obtener el nombre de la tabla asociada al modelo.
- `execute_sql()` te permite ejecutar consultas SQL manualmente cuando necesitas flexibilidad adicional o cuando la funcionalidad ORM no cubre tus necesidades. Devuelve un cursor de base de datos que puedes usar para obtener los resultados, y puede ser útil para ejecutar cualquier tipo de consulta SQL. El cursor es similar al de las bibliotecas de conexión a bases de datos como mysql-connector-python o sqlite3. Este cursor te permite manejar los resultados utilizando métodos como fetchall(), fetchone(), fetchmany(), etc.

Actividad de clase 1:

En esta actividad, deberás crear un programa en Python que cumpla con los siguientes requisitos:

1. Conectarse a una base de datos MySQL usando el ORM Peewee y el controlador PyMySQL.

2. Definir un modelo que mapee una tabla en la base de datos, asignada específicamente para cada alumn@. Esta tabla debe tener las siguientes características mínimas:

   - Debe tener al menos 4 campos que representen información relevante de los elementos de la tabla.
   - Debes asegurarte de que los nombres y tipos de columnas se correspondan con las especificaciones de tu tabla.

3. Verificar si la tabla existe y, de ser así, eliminarla para comenzar desde cero.

4. Crear la tabla en la base de datos utilizando Peewee.

5. Insertar al menos cinco registros en la tabla.

6. Mostrar mensajes informativos para confirmar la ejecución exitosa de cada etapa del programa.

7. Asegúrate de manejar correctamente las conexiones

Has de adjuntar en un fichero zip denominado actividad1.zip:

1. **Entrega una captura de pantalla que se centre en la ejecución del script. La captura de ha denominarse: actividad_de_clase1.jpg (también puede ser PNG)**
2. **Script de python que se denomine: actividad1.py.**

Motivos que harán que esta entrega no sea tenida en cuenta a efectos de evaluación ni de calificación:

1. Usar un formato que no sea ZIP.

2. Usar un formato de imagen que no sea JPG o PNG.

3. No respetar escrupulosamente los nombres indicados.

4. No entregar uno de los ficheros solicitados.

Has de tener en cuenta que tu script va a ser ejecutado por el profesor en un entorno virtual que tiene instalado Peewee y Pymysql.

6. Modificación y Recuperación de Objetos Persistentes

En esta sección, aprenderás a modificar registros existentes, recuperar los registros almacenados, buscar registros que cumplan una condición específica y eliminar registros de la base de datos utilizando el ORM Peewee.

6.1 Modificar un Registro Existente

Una vez que tienes registros insertados en tu tabla, es posible que necesites modificar la información de un registro específico. En Peewee, puedes hacerlo de la siguiente manera:

```
# Modificar la marca de la herramienta 'Martillo'
martillo = Herramienta.get(Herramienta.nombre == 'Martillo')
martillo.marca = 'Decker'
martillo.save()  # Guardar los cambios
print("Herramienta 'Martillo' actualizada.")
```

Explicación:

get(): Busca un registro específico en la base de datos. En este caso, se busca un registro donde el nombre sea 'Martillo'.

Modificar atributo: Cambiamos el valor del campo marca del registro.

save(): Guarda los cambios realizados en el registro.

6.2 Recuperar Todos los Registros

Además de modificar registros, a menudo necesitarás recuperar todos los registros de la tabla para trabajar con ellos. En Peewee, puedes hacer esto de manera sencilla usando el método select():

```
# Recuperar todas las herramientas de la base de datos
herramientas = Herramienta.select()
for herramienta in herramientas:
    print(f"Nombre: {herramienta.nombre}, Tipo: {herramienta.tipo}, Marca: {herramienta.marca}")
```

Explicación:

select(): Recupera todos los registros de la tabla especificada.

Iterar y mostrar: Recorremos todos los registros devueltos por select() y mostramos los datos de cada herramienta.

6.3 Buscar Registros que Cumplan una Condición Específica

Puedes realizar consultas para buscar registros que satisfagan ciertas condiciones. Por ejemplo, puedes buscar todas las herramientas de un tipo específico o de una marca particular.

```
# Buscar todas las herramientas de tipo 'Manual'
herramientas_manuales = Herramienta.select().where(Herramienta.tipo == 'Manual')
for herramienta in herramientas_manuales:
    print(f"Nombre: {herramienta.nombre}, Marca: {herramienta.marca}, Tipo: {herramienta.tipo}")
```

Explicación:

where(): Filtra los registros que cumplen con la condición especificada. En este caso, se seleccionan las herramientas cuyo tipo es 'Manual'.

6.4 Eliminar un Registro Específico

En caso de que necesites eliminar un registro específico, puedes hacerlo de la siguiente manera utilizando Peewee:

```
# Eliminar una herramienta específica por nombre
martillo = Herramienta.get(Herramienta.nombre == 'Martillo')
martillo.delete_instance()
print("Herramienta 'Martillo' eliminada.")
```

Explicación:

get(): Busca el registro con el nombre 'Martillo'.

delete_instance(): Elimina el registro de la base de datos.

6.5 Eliminar Múltiples Registros que Cumplan una Condición

También puedes eliminar varios registros que cumplan con una condición específica. Por ejemplo, eliminar todas las herramientas de una marca específica.

```
# Eliminar todas las herramientas de la marca 'Truper'
Herramienta.delete().where(Herramienta.marca == 'Truper').execute()
print("Herramientas de la marca 'Truper' eliminadas.")
```

Explicación:

delete(): Crea una consulta de eliminación.

where(): Filtra los registros que cumplen con la condición especificada (en este caso, aquellas herramientas cuya marca es 'Truper').

execute(): Ejecuta la consulta de eliminación.

6.6. Get() y where(): diferencias y similitudes

En Peewee, tanto get() como where() aceptan expresiones condicionales, pero la manera de utilizarlas puede diferir ligeramente en su enfoque. Ambas utilizan una sintaxis de expresiones similares basada en operadores relacionales. Sin embargo, hay ciertas diferencias en su propósito y uso.

Sintaxis de las Condiciones en get() y where():

get(): Se usa para obtener un único registro que cumpla con una condición específica. Lanza una excepción si no se encuentra ningún registro o si hay más de un registro que cumple con la condición.

```
# Ejemplo de uso de get()
registro = Modelo.get(Modelo.campo == valor)
```

where(): Se utiliza para filtrar registros y suele combinarse con métodos como select(), update() o delete(). Devuelve una consulta que puede devolver múltiples registros.

```
# Ejemplo de uso de where() con select()
registros = Modelo.select().where(Modelo.campo == valor)
```

Comparación y Ejemplos:

1. Usando get():

Propósito: Obtener un único registro que cumpla con la condición dada.

```
# Buscar un único registro con el nombre 'Martillo'
martillo = Herramienta.get(Herramienta.nombre == 'Martillo')
```

2. Usando where():

Propósito: Filtrar registros en una consulta. Puedes usarlo con métodos como select(), update(), delete(), etc.

```
# Buscar todos los registros que tengan el tipo 'Manual'
herramientas_manuales = Herramienta.select().where(Herramienta.tipo == 'Manual')
```

Diferencias clave entre get() y where():

Número de resultados:

get(): Espera obtener un único registro que cumpla con la condición especificada. Si hay más de un registro, o si no se encuentra ninguno, se lanzará una excepción (DoesNotExist o MultipleObjectsReturned).

where(): Se utiliza para filtrar un conjunto de registros y puede devolver múltiples resultados. Se combina con métodos como select(), update(), o delete().

Contexto de uso:

get() se usa cuando sabes que la condición debería coincidir con un único registro.

where() se usa para construir consultas más flexibles y generales que pueden devolver varios registros.

Ejemplos detallados:

1. Uso de get() para obtener un único registro:

```
# Obtener un único registro cuyo nombre sea 'Martillo'
try:
    martillo = Herramienta.get(Herramienta.nombre == 'Martillo')
    print(f"Herramienta encontrada: {martillo.nombre}, Marca: {martillo.marca}")
except Herramienta.DoesNotExist:
    print("No se encontró la herramienta 'Martillo'.")
```

2. Uso de where() para filtrar varios registros:

```
# Buscar todas las herramientas de tipo 'Manual'
herramientas_manuales = Herramienta.select().where(Herramienta.tipo == 'Manual')
for herramienta in herramientas_manuales:
    print(f"Nombre: {herramienta.nombre}, Marca: {herramienta.marca}, Tipo: {herramienta.tipo}")
```

Síntaxis de Condiciones:

Ambas funciones utilizan la misma sintaxis de comparación, basada en los operadores de Peewee. Por ejemplo:

- Igualdad: Modelo.campo == valor
- Desigualdad: Modelo.campo != valor
- Mayor que: Modelo.campo > valor
- Menor que: Modelo.campo < valor
- Mayor o igual que: Modelo.campo >= valor
- Menor o igual que: Modelo.campo <= valor

Ambas utilizan la misma sintaxis de condiciones basada en operadores relacionales, por lo que el conocimiento de uno es aplicable al otro.

Actividad de clase 2

1. TAREA1: Imprime por pantalla “Tarea1…” y recupera todos los objetos (los asignados a tí: libros, películas, etc.) que sean de un tipo específico (por ejemplo en Herramientas, tipo 'Manual') y muestra dos atributos de cada uno de ellos.

2. TAREA2: Imprime por pantalla “Tarea2…” y elimina un sólo registro específico en base a dos atributos (investiga cómo se hace dado que no se explica en el presente documento) y muestra los registros restantes para verificar la eliminación.
3. TAREA3: Imprime por pantalla “Tarea3…” y elimina todos los registros que cumplan una condición y muestra los registros restantes para confirmar la eliminación.

Has de adjuntar en un fichero zip denominado actividad2.zip:

1. **Entrega una captura de pantalla que se centre en la ejecución del script. La captura de ha denominarse: actividad_de_clase2.jpg (también puede ser PNG)**
2. **Script de python que se denomine: actividad2.py.**

Motivos que harán que esta entrega no sea tenida en cuenta a efectos de evaluación ni de calificación:

1. Usar un formato que no sea ZIP.

2. Usar un formato de imagen que no sea JPG o PNG.

3. No respetar escrupulosamente los nombres indicados.

4. No entregar uno de los ficheros solicitados.

Has de tener en cuenta que tu script va a ser ejecutado por el profesor en un entorno virtual que tiene instalado Peewee y Mymysql. Además este script ha de correr perfectamente tras haber ejecutado el primero.

7. Gestión de Transacciones

En esta sección, aprenderás a gestionar transacciones en Peewee para realizar operaciones seguras sobre la base de datos. Las transacciones son útiles para asegurar que una serie de operaciones se realicen de manera atómica, es decir, que se ejecuten todas o ninguna. Si ocurre un error en alguna de las operaciones, las transacciones permiten deshacer (o "rollback") los cambios realizados, manteniendo la integridad de la base de datos.

En Peewee, se puede gestionar una transacción usando el contexto db.atomic()`. Esto asegura que todas las operaciones dentro del contexto se ejecuten de manera atómica, es decir, todas se confirman al final, o se deshacen si ocurre algún error.

A continuación, se muestra un ejemplo de cómo insertar varias herramientas en la base de datos de manera atómica. Si ocurre algún error durante la inserción, se realiza un rollback para deshacer todos los cambios.

```
from peewee import IntegrityError

try:

     # Iniciar una transacción utilizando db.atomic()

     with db.atomic():

             # Insertar varias herramientas dentro de la transacción

        Herramienta.create(nombre='Martillo',   tipo='Manual',   material='Acero',   uso='Percusión',
marca='Truper')

        Heramienta.create(nombre='Taladro',          tipo='Eléctrico',          material='Plástico',
uso='Perforación', marca='Bosch')

        Herramienta.create(nombre='Destornillador',  tipo='Manual',  material='Acero',  uso='Ajuste',
marca='Stanley')

        print("Herramientas insertadas correctamente.")

except IntegrityError as e:

        print(f"Error al insertar herramientas: {e}")
```

Explicación:

- db.atomic(): Es un contexto de Peewee que asegura que todas las operaciones realizadas dentro de él se ejecuten de manera atómica. Si ocurre un error, se deshacen los cambios.

- IntegrityErro`: Es una excepción que captura errores relacionados con la integridad de la base de datos, como violaciones de restricciones únicas o claves primarias duplicadas.

Actividad de clase 3

Implementa de nuevo la actividad 1, pero con una transacción para insertar los cinco registros en la tabla que tienes asignada. Si ocurre algún error durante la inserción, los cambios deben deshacerse.

Has de adjuntar en un fichero zip denominado actividad3.zip:

1. **Entrega una captura de pantalla que se centre en la ejecución del script. La captura de ha denominarse: actividad_de_clase3.jpg (también puede ser PNG)**
2. **Script de python que se denomine: actividad3.py.**

Motivos que harán que esta entrega no sea tenida en cuenta a efectos de evaluación ni de calificación:

1. Usar un formato que no sea ZIP.

2. Usar un formato de imagen que no sea JPG o PNG.

3. No respetar escrupulosamente los nombres indicados.

4. No entregar uno de los ficheros solicitados.

Has de tener en cuenta que tu script va a ser ejecutado por el profesor en un entorno virtual que tiene instalado Peewee y Mymysql.

Actividad de clase 4

Implementa de nuevo la actividad 2, pero con transacciones para dar seguridad a aquellas partes del programa que provoquen cambios en la base de datos. Si ocurre algún error durante la inserción, los cambios deben deshacerse.

Has de adjuntar en un fichero zip denominado actividad4.zip:

1. **Entrega una captura de pantalla que se centre en la ejecución del script. La captura de ha denominarse: actividad_de_clase4.jpg (también puede ser PNG)**
2. **Script de python que se denomine: actividad4.py.**

Motivos que harán que esta entrega no sea tenida en cuenta a efectos de evaluación ni de calificación:

1. Usar un formato que no sea ZIP.

2. Usar un formato de imagen que no sea JPG o PNG.

3. No respetar escrupulosamente los nombres indicados.

4. No entregar uno de los ficheros solicitados.

Has de tener en cuenta que tu script va a ser ejecutado por el profesor en un entorno virtual que tiene instalado Peewee y Mymysql. Además este script ha de correr perfectamente tras haber ejecutado el de la actividad 3.

Anexo A. Diferencia entre “import” y “from...import”

En Python, tanto import` como from ... import` se utilizan para importar módulos o elementos de un módulo, pero tienen diferencias en cómo se usa cada una y qué efectos tienen en el código. Vamos a verlas en detalle.

1. `import`

La declaración import` importa el módulo completo. Esto significa que, una vez importado, tienes que usar el nombre del módulo seguido de un punto (.) para acceder a cualquier función, clase o variable dentro del módulo.

Ejemplo:

```
import math  # Importa todo el módulo math
```

Usamos el nombre del módulo seguido de un punto para acceder a funciones

```
resultado = math.sqrt(16)
print(resultado)  # Imprime: 4.0
```

Ventajas de `import`:

- Claridad: Siempre se sabe de qué módulo proviene la función o clase, ya que el nombre del módulo está presente.
- Menos conflictos: Si dos módulos tienen funciones con el mismo nombre, no hay conflictos porque siempre se hace referencia al módulo (por ejemplo, `math.sqrt` y `numpy.sqrt` pueden coexistir).

Desventajas de `import`:

- Puede ser más “verbose” (más largo de escribir) cuando necesitas acceder a muchas funciones o clases del módulo.

2. `from ... import`

La declaración `from ... import` importa elementos específicos de un módulo. En lugar de importar el módulo completo, seleccionas solo las funciones, clases o variables que deseas usar directamente.

Ejemplo:

```
from math import sqrt  # Importa solo la función sqrt del módulo math
```

Podemos usar la función directamente sin referenciar el módulo

```
resultado = sqrt(16)
print(resultado)  # Imprime: 4.0
```

Ventajas de `from ... import`:

- Es más conciso cuando necesitas acceder a muchas funciones o clases de un módulo.
- Reduce el uso de la notación de puntos, lo que puede hacer el código más limpio en casos donde usas una función muchas veces.

Desventajas de `from ... import`:

- Conflictos de nombres: Si importas muchas funciones o clases de diferentes módulos con el mismo nombre, puede haber colisiones, y Python solo utilizará la última importación. Por ejemplo:

  ```
  from math import sqrt
  from numpy import sqrt  # Esto sobrescribe la función sqrt de math
  ```

  Esto puede ser confuso cuando no se tiene claro de dónde proviene una función.
- Menor claridad: A veces, no está claro de dónde proviene una función si importas muchas cosas de varios módulos.

Comparación:

Usando import`:

import math

print(math.sqrt(16)) # Debes usar el nombre del módulo

Usando from ... import`:

from math import sqrt

print(sqrt(16)) # No es necesario usar el nombre del módulo

Importaciones múltiples con `from ... import`:

Puedes importar múltiples elementos de un módulo en una sola línea:

```
from math import sqrt, pi
print(sqrt(16))  # Imprime 4.0
print(pi)  # Imprime 3.141592653589793
```

Cuándo usar uno u otro:

1. Usa `import` cuando:

- Quieres mantener claridad en el código, asegurando que sea evidente de dónde provienen las funciones o clases.

- Estás usando muchas funciones de diferentes módulos que podrían tener nombres similares.

- Prefieres evitar conflictos de nombres.

2. Usa `from ... import` cuando:

- Solo necesitas algunas funciones o clases de un módulo y no quieres importar el módulo completo.

- Quieres hacer tu código más conciso y evitar escribir repetidamente el nombre del módulo.

Ejemplo combinando ambos:

A veces, puedes combinar ambos enfoques. Por ejemplo, si necesitas solo una función de un módulo, pero prefieres importar el resto del módulo completo.

```
import math
from math import sqrt
print(math.pi)  # Usamos el módulo completo para acceder a pi
print(sqrt(16))  # Usamos directamente sqrt, sin 'math.'
```

Conclusión:

- import`: Importa todo el módulo y accedes a sus funciones o clases utilizando el nombre del módulo.

- from ... import`: Importa solo partes específicas de un módulo, permitiéndote acceder a ellas directamente sin usar el nombre del módulo.