Unidad didáctica 04

Gestión de la persistencia con base de datos orientada a objetos

Fernando Usero Fuentes

Índice

UD 04: Gestión de la Persistencia con base de datos orientada a objetos

1. Introducción a ZODB

A lo largo de la evolución de las bases de datos, dos paradigmas principales han emergido: las bases de datos relacionales y las bases de datos orientadas a objetos. Las primeras, popularizadas por el uso del lenguaje SQL, han dominado el mercado debido a su capacidad para gestionar grandes volúmenes de datos estructurados de manera eficiente. Sin embargo, en algunos contextos, como el desarrollo de software orientado a objetos, las bases de datos relacionales presentan limitaciones, ya que requieren una transformación o "mapeo" de los objetos del mundo real a tablas y filas, lo que introduce una capa adicional de complejidad en el proceso de desarrollo.

Es aquí donde entra en juego ZODB (Zope Object Database), una base de datos orientada a objetos completamente integrada con Python, que permite almacenar objetos de manera nativa sin necesidad de realizar conversiones o mapeos a un formato relacional. ZODB es una base de datos que se ajusta perfectamente al paradigma de la programación orientada a objetos, donde las entidades del mundo real se representan como objetos con atributos y métodos.

¿Qué es ZODB?

ZODB es una base de datos orientada a objetos que permite almacenar, consultar y gestionar objetos de Python de manera directa. A diferencia de las bases de datos relacionales, no es necesario definir un esquema o realizar transformaciones para almacenar los datos; los objetos se guardan tal como se crean en Python. Esto elimina la necesidad de utilizar herramientas ORM (Object-Relational Mapping) para convertir los datos entre el mundo de objetos y las tablas relacionales.

Uno de los aspectos más poderosos de ZODB es su transparencia: para el programador, interactuar con ZODB se siente como trabajar con un diccionario gigante de Python que persiste en el tiempo. No es necesario preocuparse por la estructura subyacente de la base de datos, ya que ZODB se encarga de gestionar la complejidad de la persistencia de los objetos.

¿Cómo funciona ZODB?

ZODB utiliza un mecanismo de almacenamiento basado en archivos para almacenar los objetos. Cuando abres una conexión a ZODB, accedes a un archivo de almacenamiento donde se guardan los objetos. Todos los objetos almacenados son accesibles a través de un diccionario raíz (denominado `root`), que actúa como punto de entrada para navegar y manipular los objetos. Puedes pensar en este diccionario como un repositorio principal donde guardas tus objetos persistentes.

Una característica fundamental de ZODB es su soporte para transacciones. Al igual que en las bases de datos relacionales, ZODB utiliza transacciones para asegurar que los cambios en la base de datos se realicen de manera atómica. Si ocurre un error durante una operación, todos los cambios realizados hasta ese punto pueden ser revertidos mediante un rollback, asegurando que la base de datos permanezca en un estado consistente.

¿Por qué usar ZODB?

- Natural para los desarrolladores Python: Los objetos Python se pueden almacenar directamente sin la necesidad de realizar conversiones a otro formato. Esto simplifica el desarrollo de aplicaciones, ya que no es necesario utilizar un ORM ni definir un esquema de base de datos.
- Almacenamiento transparente: Los objetos se almacenan en la base de datos de la misma manera que se manejan en la memoria de Python. Esto permite que los desarrolladores se centren en la lógica de negocio en lugar de preocuparse por cómo se almacena la información en el disco.
- Gestión de estructuras de datos complejas: ZODB permite almacenar estructuras complejas de datos, como listas, diccionarios o instancias de clases con múltiples atributos. Esto es especialmente útil cuando se desarrollan aplicaciones que requieren gestionar relaciones complejas entre los datos.
- Soporte de transacciones: ZODB maneja las transacciones de manera robusta, garantizando que los cambios se confirmen correctamente o se deshagan por completo en caso de error, asegurando la integridad de los datos.

Casos de uso de ZODB

ZODB es una excelente opción para proyectos que necesitan manejar estructuras complejas de objetos y requieren una solución de persistencia que integre perfectamente con Python. Algunos ejemplos de aplicaciones que pueden beneficiarse de ZODB incluyen:

- Sistemas de gestión de contenidos (CMS): ZODB ha sido la base de Zope, un popular servidor de aplicaciones web y sistema de gestión de contenidos.
- Aplicaciones orientadas a objetos: Proyectos donde la lógica y los datos están organizados de manera natural en objetos de Python, como aplicaciones de software empresarial o sistemas de información.
- Prototipado rápido: ZODB es ideal para aplicaciones que requieren una fase rápida de prototipado sin la complejidad añadida de definir un esquema relacional.

Limitaciones de ZODB

Aunque ZODB ofrece múltiples ventajas, también tiene algunas limitaciones. Su uso no está tan extendido como el de las bases de datos relacionales, lo que puede hacer que el soporte y los recursos disponibles sean limitados. Además, para aplicaciones que manejan cantidades masivas de datos o requieren un alto grado de concurrencia, ZODB puede no ser la solución más adecuada en términos de rendimiento y escalabilidad.

2. Ventajas e Inconvenientes de las Bases de Datos que Almacenan Objetos

A medida que el mundo del desarrollo de software ha evolucionado, el paradigma orientado a objetos ha jugado un papel central en la forma en que se conciben y modelan los sistemas. Las bases de datos relacionales, que dominan el mundo de la gestión de datos desde hace décadas, fueron diseñadas inicialmente para gestionar datos estructurados, donde las entidades se organizan en tablas, filas y columnas. Este modelo es extremadamente eficiente para ciertos tipos de aplicaciones, como las financieras o de gestión de inventarios. Sin embargo, cuando hablamos de aplicaciones que modelan objetos del mundo real (con relaciones complejas entre ellos), las bases de datos relacionales presentan ciertas limitaciones.

Es aquí donde las bases de datos orientadas a objetos (como ZODB) cobran relevancia. Estas bases de datos almacenan los datos en su forma nativa como objetos de Python, lo que permite a los desarrolladores interactuar directamente con los datos de la misma forma en que los manejan en el código. No se requiere transformar los datos en un modelo relacional, lo que elimina la capa de complejidad asociada a la conversión de objetos en tablas y filas.

Ventajas de las Bases de Datos que Almacenan Objetos

1. Persistencia natural de los objetos:

La principal ventaja de las bases de datos orientadas a objetos es que no necesitan convertir los objetos del mundo real en un formato relacional para almacenarlos. Esto se traduce en una integración más fluida con el código Python, ya que los objetos se persisten de manera directa. En una base de datos como ZODB, puedes almacenar instancias de clases, listas, diccionarios y otros tipos de datos complejos de Python tal como son, sin necesidad de descomponerlos en múltiples tablas. Esto facilita la programación y reduce la cantidad de código de "pegamento" necesario para gestionar la conversión de datos.

Ejemplo:

Si tienes un objeto Herramienta con atributos como `nombre`, `tipo`, `material`, `uso` y `marca`, puedes persistir ese objeto directamente en ZODB. No necesitas definir tablas ni realizar mapeos complejos; simplemente guardas la instancia del objeto y ZODB se encarga del resto.

2. Soporte para estructuras de datos complejas:

Las bases de datos relacionales están diseñadas para manejar datos estructurados en tablas. Si necesitas representar relaciones complejas entre entidades o almacenar estructuras anidadas, como

listas o diccionarios dentro de objetos, la modelación en un sistema relacional puede volverse bastante engorrosa. En cambio, una base de datos orientada a objetos permite almacenar estas estructuras tal como son, sin necesidad de descomponerlas.

Ejemplo:

Supongamos que tienes una herramienta llamada Caja de Herramientas que contiene una lista de otras herramientas en su interior. En una base de datos relacional, tendrías que crear tablas separadas y usar claves foráneas para modelar esta relación. Con ZODB, simplemente puedes almacenar una lista de objetos Herramienta dentro de tu objeto Caja de Herramientas sin complicaciones adicionales.

3. Transacciones y persistencia automática:

Al igual que las bases de datos relacionales, ZODB soporta transacciones. Esto significa que cualquier conjunto de cambios que hagas en los objetos puede ser confirmado (commit) o revertido (rollback) como una unidad atómica. La gran ventaja aquí es que la persistencia de objetos en ZODB es automática: una vez que modificas un objeto almacenado, ZODB se encarga de mantener la consistencia y persistir los cambios cuando decides confirmar la transacción.

Ejemplo:

Imagina que actualizas una serie de herramientas en una aplicación de gestión de inventario. Si algo sale mal durante el proceso de actualización, puedes revertir los cambios y asegurar que la base de datos no quede en un estado inconsistente.

4. Eliminación de la necesidad de un ORM:

En las bases de datos relacionales, es común utilizar un ORM (Object-Relational Mapping) para mapear las clases y objetos del código a tablas y relaciones en la base de datos. Esto añade una capa adicional de complejidad al desarrollo. En ZODB, al ser una base de datos orientada a objetos, este paso no es necesario, ya que los objetos se almacenan directamente como tal. Esto no solo simplifica el desarrollo, sino que también mejora la transparencia del código.

Inconvenientes de las Bases de Datos que Almacenan Objetos

1. Menor soporte y adopción:

Aunque las bases de datos orientadas a objetos ofrecen grandes ventajas en términos de modelado de datos, su adopción es mucho más limitada en comparación con las bases de datos relacionales. La mayoría de los desarrolladores, herramientas y tecnologías están diseñados para trabajar con

bases de datos relacionales y SQL, lo que puede hacer que encontrar soporte o documentación para ZODB sea más difícil.

Ejemplo:

- Si en un equipo de desarrollo se decide usar ZODB, puede ser más complicado encontrar herramientas de gestión o monitoreo que estén diseñadas específicamente para bases de datos orientadas a objetos. Además, si los miembros del equipo están más acostumbrados a trabajar con SQL, la curva de aprendizaje podría ser mayor.

2. Sin consultas SQL:

Una de las grandes fortalezas de las bases de datos relacionales es la capacidad de realizar consultas ad hoc utilizando el lenguaje SQL. SQL es un estándar universalmente reconocido que permite realizar consultas complejas, unir datos de múltiples tablas y realizar análisis sobre los datos de forma eficiente. En ZODB, no se usa SQL, lo que significa que no tienes acceso a todas las herramientas y técnicas que SQL proporciona. Para hacer consultas en ZODB, accedes directamente a los objetos almacenados, pero esto puede ser menos eficiente para ciertas operaciones analíticas o de agregación.

Ejemplo:

- En una base de datos relacional, si necesitas encontrar todas las herramientas de un tipo específico o agrupar herramientas por su material, SQL proporciona una forma directa de hacer esto con un solo comando. En ZODB, tendrías que iterar manualmente sobre los objetos y aplicar la lógica de filtrado en tu código.

3. Escalabilidad limitada:

Las bases de datos orientadas a objetos como ZODB están diseñadas para gestionar estructuras de datos complejas en aplicaciones de tamaño mediano. Si tu aplicación crece exponencialmente en términos de cantidad de datos o número de usuarios concurrentes, es posible que ZODB no ofrezca el mismo nivel de escalabilidad y rendimiento que una base de datos relacional optimizada para operaciones masivas.

Ejemplo:

- Imagina que tienes una aplicación que inicialmente gestiona un inventario pequeño de herramientas, pero que eventualmente necesita gestionar millones de registros y ser accesible por cientos de usuarios al mismo tiempo. En estos casos, una base de datos relacional optimizada para consultas a gran escala podría ser más adecuada que una base de datos orientada a objetos como ZODB.

Conclusión

Las bases de datos orientadas a objetos como ZODB proporcionan una solución elegante para proyectos que necesitan manejar objetos y estructuras complejas de manera natural, sin la sobrecarga de convertir datos a un modelo relacional. Sin embargo, su menor adopción y escalabilidad, junto con la falta de un lenguaje universal como SQL, hacen que sean más adecuadas para proyectos medianos y con menos requisitos de concurrencia y operaciones masivas. A medida que los alumnos exploren las diferencias entre estos dos paradigmas de bases de datos, aprenderán a identificar cuál es la herramienta más adecuada para cada proyecto según sus necesidades y el tamaño de los datos a gestionar.

3. Establecer y Cerrar Conexiones. Persistencia de objetos simples.

Uno de los grandes beneficios de las bases de datos orientadas a objetos como ZODB es la capacidad de almacenar y gestionar objetos simples de Python de una manera completamente transparente y sin la necesidad de realizar ningún tipo de transformación o conversión a otro formato, como sería necesario en una base de datos relacional. En el contexto de una base de datos orientada a objetos, la "persistencia" se refiere a la capacidad de almacenar los objetos de manera que sobrevivan al cierre de la aplicación, es decir, los objetos permanecen almacenados en la base de datos y pueden ser recuperados en futuras sesiones.

Cuando hablamos de objetos simples, nos referimos a estructuras de datos de Python como listas, diccionarios, tuplas o incluso instancias de clases con atributos básicos. Estos objetos son ideales para ilustrar cómo funciona la persistencia en ZODB, ya que no requieren una estructura compleja de datos anidados o relaciones entre múltiples objetos.

Cómo Funciona la Persistencia en ZODB

En ZODB, el proceso de persistencia es extremadamente sencillo y directo. Al igual que en Python puedes almacenar y gestionar objetos en memoria utilizando un diccionario, en ZODB puedes hacer lo mismo, pero con la ventaja de que los objetos almacenados en este diccionario permanecen en la base de datos y pueden ser recuperados en cualquier momento, incluso después de cerrar y volver a abrir la base de datos.

El objeto principal a través del cual accedemos a los datos en ZODB es el diccionario raíz o `root`, que actúa como el punto de entrada a todos los objetos almacenados en la base de datos. El `root` es esencialmente un diccionario que persiste en el tiempo: puedes agregar, modificar o eliminar entradas de este diccionario como lo harías en cualquier diccionario de Python, pero con la ventaja de que los datos se almacenan de manera permanente en el sistema de archivos.

Librerías necesarias para trabajar con ZODB en Python

Sólo necesitarás instalar las siguientes 3 librerías:

1. ZODB: La base de datos principal.

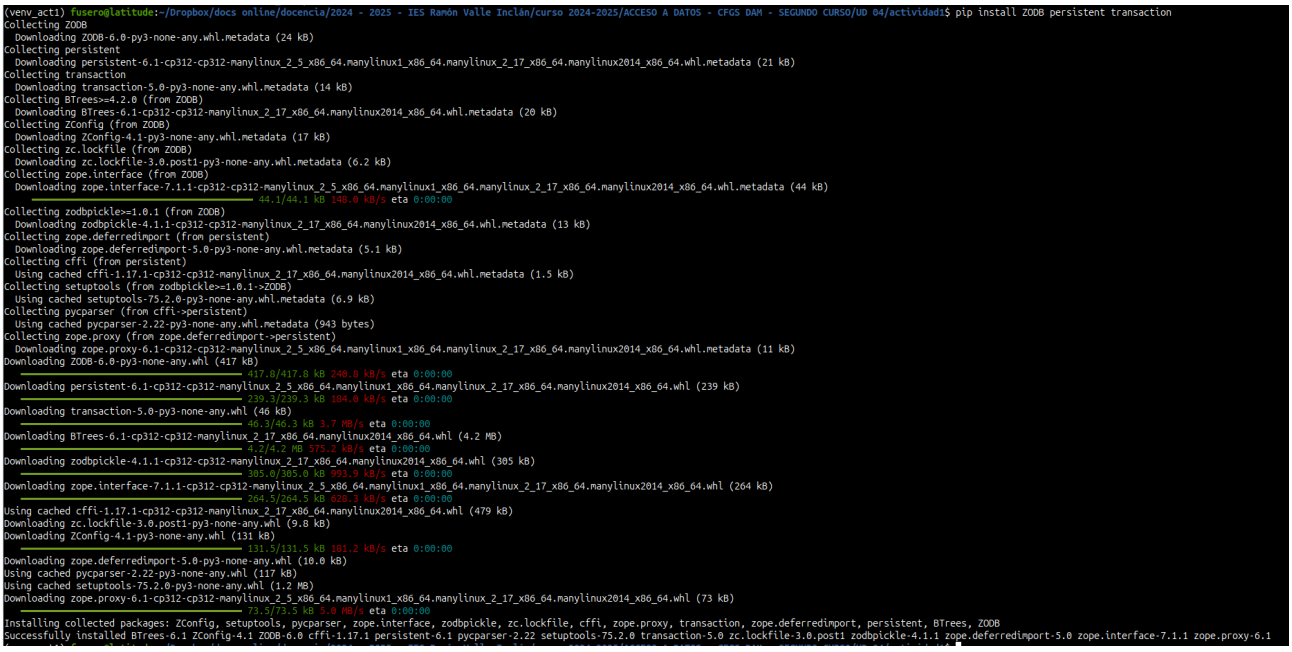
```
pip install ZODB
```

2. persistent: Necesaria para persistir objetos de clases personalizadas en ZODB. Todos los objetos que desees almacenar deben heredar de la clase Persistent.

```
pip install persistent
```

3. transaction: Gestiona las transacciones en ZODB. Aunque ZODB ya maneja las transacciones de forma interna, se utiliza esta librería para confirmar (commit) o revertir (rollback) los cambios explícitamente.

```
pip install transaction
```



Almacenamiento y Recuperación de Objetos Simples

Veamos cómo almacenar y recuperar una lista de herramientas en ZODB como un ejemplo de persistencia de objetos simples. En este caso, la lista se comporta exactamente como lo haría en memoria, pero gracias a ZODB, podemos persistirla en el disco y recuperarla en futuras sesiones.

Ejemplo: Almacenar y recuperar una lista de herramientas en ZODB

```
import ZODB, ZODB.FileStorage, transaction
# Establecer conexión a la base de datos ZODB
storage = ZODB.FileStorage.FileStorage('1dam.fs')  # Almacenamiento en archivo
db = ZODB.DB(storage)
connection = db.open()
root = connection.root()  # Diccionario raíz para acceder a los objetos almacenados
# Almacenar una lista simple en la base de datos
root['mi_lista'] = ['Martillo', 'Taladro', 'Destornillador']
transaction.commit()  # Confirmar los cambios para que sean persistentes
# Recuperar la lista almacenada y mostrarla
print(root['mi_lista'])  # Salida: ['Martillo', 'Taladro', 'Destornillador']
# Cerrar la conexión y la base de datos
connection.close()
db.close()
```

Explicación Detallada del Ejemplo:

- Establecer la conexión: Primero, se abre una conexión a la base de datos ZODB almacenada en un archivo llamado `1dam.fs`. Esto es similar a abrir una base de datos relacional, pero en este caso, el archivo actúa como el contenedor de los datos persistentes.
- `storage`: Define el archivo donde se almacenan los objetos.
- `db.open()`: Abre la base de datos.
- `root`: Es el diccionario principal que actúa como punto de entrada a todos los objetos almacenados en ZODB.
- Almacenar una lista: Luego, almacenamos una lista de herramientas en el diccionario raíz bajo la clave `'mi_lista'`. Esto es exactamente igual que agregar una entrada en un diccionario normal de Python. La diferencia es que, al realizar un `commit` mediante `transaction.commit()`, los datos se guardan de manera persistente en el archivo `1dam.fs`. El uso de `transaction.commit()` es importante porque ZODB utiliza un mecanismo transaccional. Esto significa que los cambios no se guardan permanentemente hasta que se realiza un commit. Si no ejecutáramos el commit, los cambios no se guardarían y, en caso de error, podríamos perder la información que intentábamos almacenar.
- Recuperar la lista almacenada: Finalmente, recuperamos la lista almacenada utilizando el mismo mecanismo que usaríamos en un diccionario estándar de Python. El objeto raíz `root` nos permite acceder a cualquier objeto almacenado en la base de datos.

- Cerrar la conexión: Es fundamental cerrar la conexión cuando terminamos de trabajar con la base de datos para asegurarnos de que todos los recursos se liberen adecuadamente.

¿Por Qué es Útil Almacenar Objetos Simples?

Almacenar objetos simples en ZODB puede ser muy útil en muchas aplicaciones que no requieren una estructura de datos compleja. Por ejemplo, si estás desarrollando una aplicación que necesita mantener una lista de herramientas o cualquier otro conjunto de datos simples (listas de nombres, inventarios básicos, configuraciones), ZODB proporciona una forma simple y directa de hacerlo sin tener que preocuparse por definir un esquema de base de datos o realizar mapeos de objetos a tablas relacionales.

El hecho de que ZODB se maneje como un diccionario gigante en el que puedes almacenar y recuperar objetos simples (como listas o diccionarios) lo convierte en una herramienta poderosa para prototipado rápido y desarrollo ágil. No necesitas preocuparte por los detalles de la estructura de la base de datos; simplemente almacenas tus objetos como lo harías en la memoria, pero con la garantía de que estarán disponibles la próxima vez que ejecutes la aplicación.

Actividad de clase 1

1. Objetivo: Almacenar y recuperar una lista de tres objetos en ZODB para aprender a gestionar la persistencia de objetos simples.

2. Instrucciones:

- Crea un script en Python que establezca una conexión con la base de datos 1dam.fs.
- Almacena una lista simple de tres nombres correspondientes a los objetos que te fueron asignados a principios de curso. Sólo almacenamos el nombre de los objetos (cadenas de caracteres), no objetos.
- Confirma los cambios con `transaction.commit()`.
- Luego, recupera la lista almacenada y muéstrala en la consola.

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 ZODB.

4. Gestión de la Persistencia de Objetos Estructurados

La persistencia de objetos estructurados en ZODB permite almacenar instancias de clases que tienen varios atributos y, potencialmente, relaciones con otros objetos. Este tipo de almacenamiento es especialmente útil para aplicaciones que manejan datos complejos y que necesitan almacenar objetos con múltiples atributos, relaciones o estructuras anidadas, como listas y diccionarios. A diferencia de las bases de datos relacionales, en ZODB no necesitas realizar conversiones de los objetos de Python a un formato relacional; los objetos se guardan y recuperan en su forma nativa.

Para almacenar un objeto estructurado en ZODB, el objeto debe ser una instancia de una clase que herede de Persistent (proporcionada por la librería persistent). Esto garantiza que el objeto se gestione de forma adecuada en ZODB, y que cualquier cambio realizado en el objeto se registre correctamente en la base de datos.

Ejemplo: Almacenamiento de un Objeto de la Clase Herramienta

Veamos un ejemplo de cómo almacenar y recuperar una instancia de la clase Herramienta en ZODB. En este ejemplo, Herramienta es una clase personalizada con varios atributos que representan las características de una herramienta.

```
import ZODB, ZODB.FileStorage, transaction
from persistent import Persistent

# Definir clase Herramienta
class Herramienta(Persistent):
    def __init__(self, nombre, tipo, material, uso, marca):
        self.nombre = nombre
        self.tipo = tipo
        self.material = material
        self.uso = uso
        self.marca = marca

# Establecer conexión
storage = ZODB.FileStorage.FileStorage('1dam.fs')
db = ZODB.DB(storage)
connection = db.open()
root = connection.root()
```

```
# Almacenar una herramienta
root['martillo'] = Herramienta('Martillo', 'Manual', 'Acero', 'Percusión', 'Truper')
transaction.commit()

# Recuperar la herramienta
herramienta = root['martillo']
print(f"Herramienta: {herramienta.nombre}, {herramienta.marca}")

# Cerrar conexión
connection.close()
db.close()
```

5. Desarrollo de Aplicaciones que Realizan Consultas

En ZODB, el sistema de consulta funciona de forma diferente a una base de datos relacional donde usamos SQL para filtrar, unir y ordenar los datos. Al tratarse de una base de datos orientada a objetos, en ZODB accedemos a los objetos directamente mediante claves, similar a un diccionario de Python. Esta estructura permite que las consultas simples se realicen de manera muy eficiente, aunque para operaciones más complejas puede ser necesario iterar sobre los objetos y aplicar filtros en el código.

5.1. Consultas simples

En este apartado, exploraremos cómo realizar consultas sencillas para acceder a los objetos almacenados en la base de datos ZODB y aplicar algunos filtros básicos. Los ejemplos ilustrarán cómo recuperar objetos específicos o filtrar por un atributo determinado en una colección de objetos.

Ejemplo de consulta simple:

Imaginemos que en nuestra base de datos tenemos varios objetos de tipo **Herramienta** almacenados con distintas claves. En este ejemplo, realizaremos consultas para acceder a una herramienta específica usando la clave con la que se almacenó en el diccionario raíz **root**.

```
import ZODB, ZODB.FileStorage
# Establecer conexión
storage = ZODB.FileStorage.FileStorage('1dam.fs')
db = ZODB.DB(storage)
connection = db.open()
root = connection.root()

# Realizar una consulta
herramienta = root.get('martillo')  # Recupera la herramienta 'martillo'
if herramienta:
    print(f"Herramienta: {herramienta.nombre}, Marca: {herramienta.marca}")
else:
    print("Herramienta no encontrada.")

# Cerrar conexión
connection.close()
```

```
db.close()
```

Explicación del Ejemplo

Acceso directo por clave:

Usamos el método get para acceder a un objeto específico mediante su clave en el diccionario raíz root. En este caso, intentamos recuperar la herramienta con la clave 'martillo'.

Si el objeto existe, mostramos todos sus atributos (nombre, tipo, material, uso, marca). Si no se encuentra, mostramos un mensaje indicando que no está en la base de datos.

Eficiencia:

Este tipo de consulta es rápida en ZODB, ya que accedemos al objeto directamente mediante su clave.

5.2. Consultas con Filtros Básicos

Para aplicar filtros en ZODB, debemos iterar sobre los objetos almacenados en el diccionario raíz root y usar condiciones en el código para filtrar los resultados. Esto permite realizar búsquedas por atributos específicos, aunque es menos eficiente que el acceso directo por clave.

Ejemplo: Consultar todas las herramientas de un tipo específico

Supongamos que queremos recuperar todas las herramientas de tipo Manual almacenadas en ZODB. Esto implica recorrer todos los objetos almacenados y seleccionar aquellos que cumplen la condición deseada.

```
import ZODB, ZODB.FileStorage
# Establecer conexión a la base de datos ZODB
storage = ZODB.FileStorage.FileStorage('1dam.fs')
db = ZODB.DB(storage)
connection = db.open()
root = connection.root()

# Filtrar herramientas por tipo
tipo_deseado = "Manual"
for clave, herramienta in root.items():
    if hasattr(herramienta, 'tipo') and herramienta.tipo == tipo_deseado:
         print(f"Nombre: {herramienta.nombre}, Tipo: {herramienta.tipo}, Material: {herramienta.
Material }, Uso: {herramienta.uso}, Marca: {herramienta.marca}")
```

```
# Cerrar la conexión
connection.close()
db.close()
```

Explicación del Ejemplo

A) Iteración y Filtrado:

Recorremos todos los objetos almacenados en root. Para cada objeto, verificamos si tiene el atributo tipo (esto evita errores si hay otros objetos sin este atributo) y si coincide con el tipo_deseado especificado.

Si el tipo de la herramienta coincide con el tipo deseado ("Manual"), se imprime la información de la herramienta.

B) Sintaxis pythoniana:

La sintaxis que estamos usando en el bucle para iterar sobre los elementos almacenados en el diccionario raíz de ZODB es la siguiente:

```
for clave, herramienta in root.items():
  # código para trabajar con cada herramienta
```

En Python, esta línea es una forma común de recorrer los pares clave-valor de un diccionario. Vamos a desglosar esto en detalle:

```
root.items():
```

items() es un método de los diccionarios en Python que devuelve una vista de todos los elementos del diccionario en forma de pares (clave, valor). Cada par representa una clave y su correspondiente valor en el diccionario.

En el caso de ZODB, el diccionario root contiene todas las claves (identificadores) y objetos almacenados en la base de datos, de modo que root.items() devolverá todos los pares clave-objeto.

El bucle desempaqueta cada par (clave, valor) devuelto por root.items() en dos variables: clave y herramienta. En cada iteración del bucle, clave toma el valor de la clave del par actual, mientras que herramienta se asigna al valor correspondiente a esa clave (en este caso, una instancia de la clase Herramienta).

C) Acceso a Atributos:

Dentro del bucle, puedes trabajar directamente con herramienta como un objeto, accediendo a sus atributos y métodos.

En el ejemplo, estamos usando hasattr(herramienta, 'tipo') para verificar que el objeto herramienta tiene un atributo tipo. **`hasattr`** es una **función incorporada de Python** (también conocida como built-in), por lo que puedes usarla directamente sin importar ninguna librería adicional.

Luego, aplicamos el filtro para comprobar que el valor de tipo coincide con el tipo deseado ("Manual").

Actividad de clase 2:

Objetivo:

Desarrollar una aplicación que almacene y consulte información sobre un tipo de objeto asignado (por ejemplo, Libro, Película, ObraDeArte o EdificioHistorico) en una base de datos ZODB.

Instrucciones:

1. Crear una clase:

Define una clase con el tipo de objeto que te fue asignado al inicio del curso (por ejemplo, Libro si te fue asignado trabajar con información sobre libros).

La clase debe incluir al menos cuatro atributos (por ejemplo, titulo, autor, genero, año_publicacion para un libro).

2. Almacenar varios objetos:

Crea y almacena tres instancias de la clase en ZODB, cada una con diferentes valores para los atributos.

Usa transaction.commit() para confirmar los cambios en la base de datos.

3. Consultar los objetos:

Recorre todos los objetos en la base de datos y utiliza hasattr para verificar si cada objeto tiene un atributo específico (por ejemplo, genero).

Filtra y muestra solo los objetos que tienen el atributo y cumplen con una condición específica (por ejemplo, género igual a "Novela").

4. Salida esperada:

La consola debe mostrar todos los objetos que cumplen con la condición, incluyendo todos sus atributos.

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 ZODB.

6. Modificación de Objetos Almacenados. Introducción a las transacciones

En ZODB, los objetos almacenados en la base de datos pueden ser modificados de forma similar a los objetos en memoria. Esto significa que puedes acceder a un objeto persistente, modificar sus atributos y luego confirmar esos cambios en la base de datos. ZODB maneja estos cambios automáticamente, pero es importante realizar un commit de la transacción para asegurarte de que los cambios se persistan.

Este apartado explora cómo recuperar un objeto de ZODB, modificar sus atributos y guardar los cambios en la base de datos. Este tipo de operación es esencial en aplicaciones que requieren mantener los datos actualizados, ya que permite realizar modificaciones sin la necesidad de eliminar o recrear los objetos.

Ejemplo: Modificar un Objeto de la Clase Herramienta

Imaginemos que tenemos una base de datos ZODB con varias herramientas almacenadas. En este ejemplo, actualizaremos un atributo de una herramienta específica (por ejemplo, el material de un martillo) y confirmaremos el cambio en la base de datos.

```
import ZODB, ZODB.FileStorage, transaction
from persistent import Persistent
# Definir la clase Herramienta
class Herramienta(Persistent):
    def __init__(self, nombre, tipo, material, uso, marca):
        self.nombre = nombre
        self.tipo = tipo
        self.material = material
        self.uso = uso
        self.marca = marca
# Establecer conexión a la base de datos ZODB
storage = ZODB.FileStorage.FileStorage('1dam.fs')
db = ZODB.DB(storage)
connection = db.open()
root = connection.root()
# Recuperar y modificar un objeto
herramienta = root.get('martillo')  # Recuperar la herramienta almacenada con la clave 'martillo'
if herramienta:
```

```
    print("Antes de la modificación:")
    print(f"Nombre: {herramienta.nombre}, Material: {herramienta.material}")

    # Modificar el atributo 'material'
    herramienta.material = 'Aleación de acero'
    transaction.commit()  # Confirmar los cambios en la base de datos

    print("Después de la modificación:")
    print(f"Nombre: {herramienta.nombre}, Material: {herramienta.material}")
else:
    print("La herramienta no se encontró en la base de datos.")

# Cerrar la conexión
connection.close()
db.close()
```

Explicación del Ejemplo

1. **Recuperación del objeto**:
    - Usamos **`root.get('martillo')`** para acceder a la herramienta almacenada con la clave `'martillo'`.
    - Si el objeto existe, mostramos su información antes de realizar la modificación.
2. **Modificación del atributo**:
    - Cambiamos el valor del atributo `material` a "**Aleación de acero**".
    - Realizamos un **commit** de la transacción con **`transaction.commit()`** para confirmar el cambio en la base de datos. Este paso es crucial, ya que sin el commit, el cambio solo se realizaría en memoria y no se guardaría en la base de datos. Ver siguiente apartado.
3. **Verificación del cambio**:
    - Volvemos a mostrar el atributo `material` de la herramienta para verificar que la modificación se ha realizado correctamente.

Sobre transaction.commit()

La librería transaction en Python, utilizada por ZODB, permite agrupar varias operaciones en una única transacción. Esto significa que todas las modificaciones que realices en los objetos persistentes de la base de datos (objetos que heredan de Persistent) están incluidas en la transacción actual. Al ejecutar transaction.commit(), se confirma o "persisten" todos los cambios en ZODB.

La magia de transaction en ZODB radica en el uso del modelo de persistencia automática de ZODB. Cuando se realizan cambios en objetos de ZODB (instancias de clases que heredan de Persistent), ZODB registra automáticamente estos cambios en la transacción actual. No necesitas especificar qué operaciones incluir, ya que ZODB detecta internamente las modificaciones en sus objetos persistentes:

A. Registro automático:

- Cada vez que modificas un objeto que es persistente en ZODB, ese cambio se añade automáticamente a la transacción actual.
- No necesitas agregar los cambios explícitamente; ZODB los rastrea y añade en la transacción actual.

B. Confirmación con transaction.commit():

- transaction.commit() guarda los cambios de la transacción actual en la base de datos, haciendo que sean permanentes.
- Si ocurre algún error antes de llamar a commit, los cambios no confirmados en esa transacción no se guardarán en la base de datos.

C. Revertir los cambios con transaction.abort():

- También puedes usar transaction.abort() para cancelar todos los cambios de la transacción actual, revirtiendo los objetos a su estado anterior.

Puedes usar `transaction.commit()` directamente después de importar `transaction`. Cuando haces `import transaction`, obtienes acceso a la transacción global, que automáticamente agrupa todas las operaciones de modificación realizadas en los objetos persistentes de ZODB en el contexto actual.

Actividad de clase 3:

1. **Objetivo**: Practicar la modificación de objetos persistentes en ZODB.
2. **Instrucciones**:
    - Crea varios objetos del tipo asignado al inicio del curso (por ejemplo, **Libro**, **Película**, **EdificioHistorico**).
    - Almacena al menos tres instancias en la base de datos ZODB.
    - Recupera uno de los objetos por su clave y modifica uno de sus atributos (por ejemplo, actualiza el `año_publicacion` de un **Libro** o el `material` de un **EdificioHistorico**).
    - Realiza un **commit** para guardar los cambios.
    - Verifica la modificación imprimiendo el valor del atributo antes y después del cambio

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 ZODB.

7. Gestión de Transacciones con control de excepciones

Vamos a ver en este apartado como gestionar las excepciones de forma que el control de la transacción tenga coherencia, esto es poner todo el código “transaccionable” dentro de un “try” para que en el catch lancemos el rollback en caso de que haya una excepción.

Este enfoque asegura que la base de datos mantenga la coherencia de los datos incluso si ocurre un error a mitad de una operación.

Ejemplo de Gestión de Transacciones con excepciones en ZODB

Supongamos que estamos trabajando con un objeto de clase `Herramienta` en una base de datos ZODB para gestionar inventarios de herramientas. Queremos añadir varias herramientas en una sola transacción. Si todo se inserta correctamente, confirmaremos la transacción; si ocurre un error, la revertiremos.

```
from ZODB import DB, FileStorage
from persistent import Persistent
import ZODB
import transaction


# Clase Herramienta para el ejemplo
class Herramienta(Persistent):
    def __init__(self, nombre, tipo, marca, uso, material):
        self.nombre = nombre
        self.tipo = tipo
        self.marca = marca
        self.uso = uso
        self.material = material
# Conectar a la base de datos ZODB
storage = FileStorage.FileStorage('1dam.fs')
db = DB(storage)
connection = db.open()
root = connection.root()
# Función para gestionar la inserción de varias herramientas con transacción
def agregar_herramientas():
```

```
    try:
        print("Iniciando la transacción para agregar herramientas...")
        # Verificar y crear 'herramientas' en root si no existe
        if 'herramientas' not in root:
            root['herramientas'] = {}  # Inicializar una colección de herramientas si no existe
            transaction.commit()  # Confirmar la creación en la base de datos
        # Crear y añadir nuevas herramientas
        herramienta1 = Herramienta("Martillo", "Manual", "Truper", "Percusión", "Acero")
        herramienta2 = Herramienta("Taladro", "Eléctrico", "Bosch", "Perforación", "Plástico")
        herramienta3 = Herramienta("Sierra", "Manual", "Stanley", "Corte", "Acero")
        # Añadir herramientas a la colección en la raíz de ZODB
        root['herramientas']["Martillo"] = herramienta1
        root['herramientas']["Taladro"] = herramienta2
        root['herramientas']["Sierra"] = herramienta3
        # Confirmar la transacción
        transaction.commit()
        print("Transacción completada: Herramientas añadidas correctamente.")
    except Exception as e:
        # Si ocurre un error, revertimos la transacción
        transaction.abort()
        print(f"Error durante la transacción: {e}. Transacción revertida.")

# Llamar a la función para añadir herramientas
agregar_herramientas()
# Cerrar la conexión a la base de datos ZODB
connection.close()
db.close()
```

Explicación del Código

1. Iniciar Transacción e Insertar Herramientas:

- Creamos un diccionario de herramientas en la raíz de la base de datos (root[‘herramientas’]), si aún no existe. Las líneas:

```
if 'herramientas' not in root:
    root['herramientas'] = {}  # Inicializar una colección de herramientas si no existe
    transaction.commit()  # Confirmar la creación en la base de datos
```

Esta técnica es común para evitar errores de acceso a claves inexistentes y para inicializar estructuras de datos si es la primera vez que se utilizan en el programa. Confirmamos inmediatamente la creación de `root['herramientas']` en el primer `if`, lo que garantiza que la colección esté disponible para la transacción de añadir herramientas.

- Creamos tres instancias de la clase `Herramienta`, cada una con sus atributos específicos.
- Asignamos cada herramienta al diccionario de herramientas en `root` con una clave única (por ejemplo, el nombre de la herramienta).

2. Confirmar o Revertir la Transacción:

- Si la inserción es exitosa, confirmamos la transacción utilizando `transaction.commit()`.

- Si ocurre algún error durante el proceso, la excepción es capturada y la transacción es revertida mediante `transaction.abort()`. Esto asegura que no se guarden cambios parciales en caso de fallo.

3. Usar root como almacén de pares clave-valor. Dos enfoques distintos

En ZODB, podemos organizar los datos en forma de pares clave-valor de distintas maneras. A continuación, exploramos dos enfoques para almacenar herramientas en root, explicando cómo se estructuran y qué beneficios ofrecen.

Primer Enfoque: Herramientas Directas en root

root

├── 'Martillo' : Objeto Herramienta

├── 'Taladro' : Objeto Herramienta

└── 'Sierra'   : Objeto Herramienta

Descripción:

- En este primer enfoque, cada herramienta (`'Martillo'`, `'Taladro'`, `'Sierra'`) se almacena directamente como un par clave-valor en `root`.
- Cada clave es el nombre de la herramienta, y el valor asociado es un objeto de la clase `Herramienta` que contiene los detalles de esa herramienta.

Ventajas:

- **Acceso directo**: Las herramientas están en `root` sin una colección intermedia, lo que permite acceder rápidamente a cada herramienta.
- **Simplicidad**: Este enfoque es ideal cuando tenemos solo unas pocas herramientas, ya que cada elemento está en el nivel superior sin estructura adicional.

Ejemplo de código:

```
# Almacenar herramientas directamente en root
root['Martillo'] = Herramienta("Martillo", "Manual", "Truper", "Percusión", "Acero")
root['Taladro'] = Herramienta("Taladro", "Eléctrico", "Bosch", "Perforación", "Plástico")
```

```
root['Sierra'] = Herramienta("Sierra", "Manual", "Stanley", "Corte", "Acero")
```

Segundo Enfoque: Uso de una Colección `root['herramientas']`

```
root
 └── 'herramientas'
      ├── 'Martillo' : Objeto Herramienta
      ├── 'Taladro'  : Objeto Herramienta
      └── 'Sierra'   : Objeto Herramienta
```

Descripción:

- En este segundo enfoque, almacenamos todas las herramientas dentro de una colección llamada `root['herramientas']`.
- `root['herramientas']` es un diccionario que agrupa las herramientas bajo una misma clave en `root`.
- Dentro de `root['herramientas']`, cada clave es el nombre de la herramienta (`'Martillo'`, `'Taladro'`, `'Sierra'`), y el valor es un objeto `Herramienta`.

Ventajas:

- **Organización centralizada**: Este enfoque es útil cuando queremos agrupar y gestionar todas las herramientas desde una misma colección.
- **Estructura de inventario**: Ideal si esperamos agregar o eliminar herramientas frecuentemente, ya que todas están en un espacio común.

Ejemplo de código:

```
# Crear una colección de herramientas en root y añadir herramientas a ella
if 'herramientas' not in root:
    root['herramientas'] = {}

root['herramientas']['Martillo'] = Herramienta("Martillo", "Manual", "Truper", "Percusión", "Acero")
root['herramientas']['Taladro']  =  Herramienta("Taladro",  "Eléctrico",  "Bosch",  "Perforación",
"Plástico")
root['herramientas']['Sierra'] = Herramienta("Sierra", "Manual", "Stanley", "Corte", "Acero")
```

Actividad de clase 4

Actividad: Implementa una transacción en la que se añadan al menos tres objetos diferentes de la clase asignada (por ejemplo, `Libro`, `Película` o `Edificio Histórico`) a la base de datos ZODB. Si la transacción se completa sin errores, confirma los cambios. Si ocurre un error, revierte la transacción. Utiliza el segundo enfoque.

Finalmente, muestra por consola todos los al menos 3 elementos persistidos en bases de datos. Ojo a esta parte, que el código de ejemplos anteriores requiere un mínimo cambio debido a que estamos usando el segundo enfoque.

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 ZODB.

8. Gestión de dos tablas relacionadas con una Foreign Key

En este apartado, exploraremos cómo manejar datos en dos tablas, **Herramientas** y **Proveedores**, en una base de datos orientada a objetos con ZODB. Cada herramienta tendrá un campo `id_proveedor` que hace referencia al proveedor del cual fue adquirida, lo que permite gestionar relaciones entre ambas tablas.

Estructura de Datos

1. **Tabla Herramientas**
    - Almacena datos de las herramientas disponibles.
    - Cada herramienta tiene los atributos `nombre`, `tipo`, `marca`, `uso`, `material` y **`id_proveedor`** (ID del proveedor que suministró la herramienta).
    - Se almacenará en la colección `root['herramientas']` para facilitar el acceso y la organización.
2. **Tabla Proveedores**
    - Almacena información de los proveedores de herramientas.
    - Cada proveedor tiene los atributos `nombre_proveedor`, `direccion` y `telefono`.
    - Se almacenará en la colección `root['proveedores']` y estará relacionado con las herramientas que suministra.

Ejemplo de Estructura en ZODB

```
root
 ├── 'herramientas'
 │    ├── 'Martillo' : Objeto Herramienta (id_proveedor='Proveedor1')
 │    ├── 'Taladro'  : Objeto Herramienta (id_proveedor='Proveedor1')
 │    └── 'Sierra'   : Objeto Herramienta (id_proveedor='Proveedor2')
 └── 'proveedores'
      ├── 'Proveedor1' : Objeto Proveedor
      └── 'Proveedor2' : Objeto Proveedor
```

Código de Ejemplo

Aquí tienes un ejemplo de cómo podríamos definir e insertar objetos en estas tablas en ZODB, teniendo en cuenta el campo `id_proveedor` en `Herramienta`:

```
from persistent import Persistent
```

```
import transaction

# Clases para Herramientas y Proveedores
class Herramienta(Persistent):
    def __init__(self, nombre, tipo, marca, uso, material, id_proveedor):
        self.nombre = nombre
        self.tipo = tipo
        self.marca = marca
        self.uso = uso
        self.material = material
        self.id_proveedor = id_proveedor  # ID del proveedor

class Proveedor(Persistent):
    def __init__(self, nombre_proveedor, direccion, telefono):
        self.nombre_proveedor = nombre_proveedor
        self.direccion = direccion
        self.telefono = telefono

# Verificar y crear colecciones si no existen
if 'herramientas' not in root:
    root['herramientas'] = {}

if 'proveedores' not in root:
    root['proveedores'] = {}

# Insertar datos en Proveedores
root['proveedores']['Proveedor1'] = Proveedor("Proveedor1", "Calle Falsa 123", "123-456-789")
root['proveedores']['Proveedor2'] = Proveedor("Proveedor2", "Avenida Real 456", "987-654-321")

# Insertar datos en Herramientas, incluyendo id_proveedor
root['herramientas']['Martillo'] = Herramienta("Martillo", "Manual", "Truper", "Percusión", "Acero",
"Proveedor1")
root['herramientas']['Taladro']  =  Herramienta("Taladro",  "Eléctrico",  "Bosch",  "Perforación",
"Plástico", "Proveedor1")
root['herramientas']['Sierra']  =  Herramienta("Sierra",  "Manual",  "Stanley",  "Corte",  "Acero",
"Proveedor2")

transaction.commit()
```

¿Qué ocurre si borro un proveedor?

Si eliminas un proveedor en esta estructura, ocurren dos situaciones que debes tener en cuenta:

Relaciones Huérfanas en Herramientas:

Las herramientas que tengan su id_proveedor asociado al proveedor eliminado conservarán ese ID, pero el proveedor ya no existirá en root['proveedores'].

Esto crea una "relación huérfana", en la que las herramientas tienen un id_proveedor que no apunta a un proveedor válido en la base de datos.

Impacto en la Integridad de Datos:

Si el sistema o los usuarios intentan acceder a los datos del proveedor desde la herramienta, se producirá un error o la información será inconsistente, ya que el proveedor ya no existe.

La integridad de datos se ve afectada al quedar herramientas con referencias a proveedores inexistentes.

Solución: Eliminar Referencias Antes de Borrar el Proveedor

Para evitar que queden relaciones huérfanas, puedes hacer lo siguiente antes de borrar un proveedor:

1. Buscar Herramientas Asociadas al Proveedor: Busca todas las herramientas cuyo id_proveedor coincida con el ID del proveedor que deseas eliminar.
2. Actualizar o Eliminar las Referencias: Decide si deseas actualizar el id_proveedor de estas herramientas a un valor predeterminado (por ejemplo, None) o eliminarlas por completo.
3. Eliminar el Proveedor: Una vez que todas las referencias se hayan actualizado, procede a eliminar el proveedor de root['proveedores'].

Actividad de clase 5:

Haz lo mismo que en este apartado pero con el objeto que te ha sido asignado y una tabla adicional que se te ocurra con la que tenga una relación 1 a n y que tenga sentido en la vida real. El programa debe insertar en base de datos al menos tres objetos de la clase que se te asignó a principios de curso y dos de la nueva clase.

Usaremos el enfoque 2, en el que los objetos de cada tipo están en distintas jerarquías colgando de root.

Además el programa mostrará por pantalla los objetos que cumplan una determinada condición de tu invención relacionada con la segunda tabla. Por ejemplo: “mostrar las herramientas que fueron adquiridas al proveedor 2”.

Has de adjuntar en un fichero zip denominado actividad5.zip:

1. **Entrega una captura de pantalla que se centre en la ejecución del script. La captura de ha denominarse: actividad_de_clase5.jpg (también puede ser PNG)**
2. **Script de python que se denomine: actividad5.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 ZODB.

Anexo A. Deepcopy

`deepcopy` es especialmente útil y conveniente para hacer copias de objetos complejos en Python, y ofrece ventajas significativas sobre crear una copia manualmente, especialmente cuando trabajas con estructuras de datos anidadas y referencias en una base de datos como ZODB.

Aquí tienes las principales ventajas de `deepcopy`:

1. Copia Completa de Estructuras Anidadas

Cuando haces una copia manual, solo copias los atributos y valores de nivel superior, lo cual puede ser suficiente en casos simples. Sin embargo, si el objeto tiene atributos que son a su vez otros objetos (por ejemplo, una herramienta que incluye otra clase como “Especificaciones”), `deepcopy` crea una copia independiente de cada nivel en esa estructura.

Ejemplo de estructura compleja:

```
class Especificaciones:
    def __init__(self, peso, dimensiones):
        self.peso = peso
        self.dimensiones = dimensiones

class Herramienta:
    def __init__(self, nombre, especificaciones):
        self.nombre = nombre
        self.especificaciones = especificaciones

herramienta_original = Herramienta("Taladro", Especificaciones("2 kg", "30x15x10 cm"))
herramienta_copia = copy.deepcopy(herramienta_original)
```

- Con `deepcopy`, tanto `herramienta_copia` como `herramienta_original` tendrán copias independientes de `Especificaciones`.

- Sin `deepcopy`, si intentas hacer una copia manual, es fácil que ambas `Herramienta` terminen compartiendo el mismo objeto `Especificaciones`, lo cual puede causar errores si luego intentas modificar la copia sin afectar al original.

2. Evitar Referencias Compartidas (Problema de Alias)

Al usar `deepcopy`, evitas referencias compartidas o aliasing en los atributos internos del objeto. Esto significa que si haces un cambio en el objeto copiado, el original permanecerá intacto, y viceversa. Esto es especialmente útil en sistemas de persistencia de datos como ZODB, donde los cambios en un objeto copiado pueden reflejarse de manera inesperada en el objeto original si no se hace una copia profunda.

Sin `deepcopy`, un cambio en un atributo compartido afectará a ambos objetos:

```
# Sin deepcopy
herramienta_copia = herramienta_original
herramienta_copia.especificaciones.peso = "1.5 kg"
print(herramienta_original.especificaciones.peso)  # Resultado: "1.5 kg" (modificación no deseada)
```

Con `deepcopy`, el original se mantiene intacto:

```
herramienta_copia = copy.deepcopy(herramienta_original)
herramienta_copia.especificaciones.peso = "1.5 kg"
print(herramienta_original.especificaciones.peso)  # Resultado: "2 kg" (el original no cambia)
```

3. Facilidad y Reducción de Errores

Hacer una copia profunda manualmente es posible, pero requiere que conozcas la estructura completa de los objetos y recuerdes copiar cada nivel y referencia, lo cual es propenso a errores y puede volverse complejo rápidamente. `deepcopy` hace esto automáticamente y maneja todos los niveles de profundidad en la copia, lo cual hace el código más seguro y fácil de mantener.

Resumen

En resumen, `deepcopy` es ideal cuando:

- Trabajas con estructuras complejas y anidadas.

- Quieres evitar efectos no deseados en el objeto original al modificar la copia.

- Prefieres una solución confiable y libre de errores manuales.

En casos simples, una copia manual puede funcionar, pero para estructuras más complejas y para asegurar una independencia completa, `deepcopy` es la mejor opción.

Actividad de clase 6:

Seguimos trabajando en el mismo contexto de la actividad 5. Tenemos la tabla que se nos asignó a principios de curso y la tabla relacionada con ella de forma 1:N.

Añadiremos código al de la actividad de clase 5 para que, usando deepcopy, creemos una copia completa de una instancia del objeto que se nos fue asignado (en el caso del profesor herramientas) que incluye una referencia a la tabla secundaria (en el caso del profesor,proveedor). Modificar la copia y verificar que cualquier cambio en la copia no afecta al original ni a su objeto referenciado.

Has de adjuntar en un fichero zip denominado actividad6.zip:

1. **Entrega una captura de pantalla que se centre en la ejecución del script. La captura de ha denominarse: actividad_de_clase6.jpg (también puede ser PNG)**
2. **Script de python que se denomine: actividad6.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 ZO