Checkpoint 6

¿Para qué usamos Clases en Python?

Las clases en Python son estructuras que permiten crear objetos con características y comportamientos similares. Son como moldes que se pueden usar para crear múltiples objetos que compartirán los mismos atributos.

```
class Cliente:
    def __init__(self, nombre, apellido):
        self.nombre = nombre
        self.apellido = apellido

    def saludar(self):
        print(f'Hola, {self.nombre} {self.apellido}')
```

```
Aqui se define la clase como Cliente, y la funcion  _init_ se llama
automáticamente
al crear una nueva instancia de la clase. Se utiliza para inicializar los
atributos de la instancia.
Y la funcion saludar se usa para imprimir imprimir en la cadena el nombre y
apellido del cliente.
```

Este ejemplo funcionaria para mandar correos masivos a clientes y que sean personalizados segun la base de datos. O tambien funcinaria al ingresar en un sitio donde estes registrado. Va a hacer el mismo proceso muchas veces con distintos datos.

Ventajas:

- **Reutilización de código:** Las clases te permiten evitar escribir código repetitivo.
- **Organización:** Las clases ayudan a organizar el código en bloques lógicos.
- **Mantenimiento:** Las clases facilitan la actualización y el mantenimiento del código.
- **Extensibilidad:** Las clases permiten crear nuevas funcionalidades a partir de las existentes.

Desventajas:

- **Curva de aprendizaje:** Las clases pueden ser un poco difíciles de entender al principio.
- **Complejidad:** Las clases pueden aumentar la complejidad del código si se usan de forma incorrecta.

Errores comunes en las clases de Python:

1. Errores de sintaxis:

- Falta de dos puntos (:) después del nombre de la clase.
- Paréntesis no válidos en la definición de la clase.
- Errores de indentación en el código de la clase.
- Falta de self como primer argumento en los métodos de la clase.

2. Errores de lógica:

- Atributos o métodos no definidos: Usar un atributo o método que no existe en la clase.
- Tipos de datos incorrectos: Asignar un valor de un tipo de dato incompatible a un atributo.
- Errores en la lógica de los métodos: Implementación incorrecta de la funcionalidad del método.

3. Errores de referencia:

- Referencias circulares: Dos o más clases se referencian entre sí de forma indirecta.
- Nombres de variables ambiguos: Usar el mismo nombre para diferentes variables en la clase.
- Falta de limpieza: No liberar recursos al eliminar una instancia de la clase.

4. Errores de diseño:

- Clases demasiado complejas: Intentar encapsular demasiada funcionalidad en una sola clase.
- Falta de modularidad: No dividir la funcionalidad en clases más pequeñas y manejables.
- Violación del principio de encapsulación: Exponer detalles internos de la clase que no deberían ser públicos.

¿Qué método se ejecuta automáticamente cuando se crea una instancia de una clase?

El metodo que se ejecuta automaticamente es la instancia `__init__`. Sirve para asignar valores a los atributos que se hayan establecido para la clase. En el ejemplo anterior se puede ver que al crear la clase cliente se ejecuta la instancia, con los atributos de nombre y apellido.

```python
class Cliente:
    def __init__(self, nombre, apellido):
        self.nombre = nombre
        self.apellido = apellido
```

El método `__init__` en Python se ejecuta automáticamente cada vez que se crea una nueva instancia de una clase. También se puede llamar explícitamente desde otro método de la clase. Aunque no es obligatorio, es una buena práctica tenerlo para inicializar los atributos de la instancia. Si no se define `__init__`, se crea un método `__init__` vacío por defecto.

Para usar `__init__`, se deben definir los argumentos del método para recibir los datos necesarios para la inicialización. Luego, se asignan los valores de los argumentos a los atributos de la instancia. Finalmente, se puede realizar cualquier otra configuración necesaria para la nueva instancia.

¿Cuáles son los tres verbos de API?

Las APIs son como puentes que permiten a las aplicaciones comunicarse entre sí. Y viene del acrónimo “Application Programming Interface”. Existen dos tipos de APIs las cuales son: las generales, que permiten acceder a datos y funcionalidades de una aplicación de forma general. y las APIs REST que son el tipo más utilizado para la integración de datos. Facilitan la transferencia eficiente de datos entre aplicaciones.

Los verbos principales de las API REST

1. GET: Se utiliza para obtener información de un recurso. Es el verbo más utilizado y no debe modificar el estado del recurso.

Ejemplo:

- Obtener la lista de productos de una tienda online.
- Consultar el estado de un pedido.
- Buscar información sobre un usuario.

`GET /api/productos`

2. POST: Se utiliza para crear un nuevo recurso. La información del nuevo recurso se envía en el cuerpo de la solicitud.

Ejemplo:

- Crear un nuevo usuario en una red social.
- Publicar un comentario en un blog.
- Realizar una compra en una tienda online.

`POST /api/usuarios`

3. PUT: Se utiliza para actualizar un recurso existente. La información actualizada se envía en el cuerpo de la solicitud.

Ejemplo:

- Actualizar la información de un usuario.
- Modificar el contenido de un artículo.
- Cambiar la contraseña de una cuenta.

`PUT /api/usuarios/1`

¿Es MongoDB una base de datos SQL o NoSQL?

MongoDB se clasifica como una base de datos **NoSQL** por las siguientes razones:

- **Almacena los datos en documentos JSON:** Los documentos JSON son flexibles y no requieren un esquema predefinido, lo que los hace ideales para datos no estructurados.
- **Utiliza el lenguaje de consulta MQL:** MQL es un lenguaje de consulta específico para MongoDB que permite acceder y modificar los datos de forma eficiente.
- **Es ideal para aplicaciones que necesitan escalabilidad, rendimiento y flexibilidad:** MongoDB puede escalarse horizontalmente para añadir más capacidad a medida que la aplicación crece. También ofrece un alto rendimiento para las consultas y operaciones de escritura.

Ejemplos de aplicaciones que usan MongoDB:

- Tiendas online
- Redes sociales
- Aplicaciones móviles
- Análisis de datos

Errores comunes al usar MongoDB:

1. Errores de conexión:

- No se puede conectar al servidor: Asegúrate de que el servidor MongoDB esté funcionando y que la dirección IP y el puerto sean correctos.
- Error de autenticación: Verifica que el nombre de usuario y la contraseña sean correctos.
- Permisos insuficientes: Asegúrate de que el usuario tenga los permisos necesarios para acceder a la base de datos y la colección.

2. Errores de consulta:

- Sintaxis incorrecta: Revisa la sintaxis de la consulta MQL y asegúrate de que no haya errores.
- Nombre de campo o colección incorrecto: Verifica que el nombre del campo o la colección sean correctos.
- Filtros mal formados: Asegúrate de que los filtros de la consulta estén bien formados y sean válidos.

3. Errores de datos:

- Formato de datos incorrecto: Asegúrate de que los datos que se insertan o actualizan tengan el formato correcto.
- Valores nulos o vacíos: Verifica que los campos obligatorios no sean nulos o vacíos.
- Duplicación de claves: Si la colección tiene una clave única, asegúrate de que no se inserten documentos con la misma clave.

4. Errores de rendimiento:

- Consultas ineficientes: Revisa el índice de la colección y optimiza las consultas para mejorar el rendimiento.
- Carga excesiva del servidor: Si el servidor está sobrecargado, puede ser necesario aumentar la capacidad o distribuir la carga entre varios servidores.
- Problemas de hardware: Asegúrate de que el hardware del servidor sea adecuado para la carga de trabajo de MongoDB.

¿Qué es una API?

Una API, o Interfaz de Programación de Aplicaciones, es un conjunto de definiciones y protocolos que permiten que dos programas se comuniquen entre sí. La API define cómo se intercambian los datos y las funcionalidades entre los dos programas. Imagina que tienes dos aplicaciones: una que guarda recetas y otra que crea listas de compras.

Con una API, la aplicación de recetas puede enviar un mensaje a la aplicación de lista de compras con la información de los ingredientes. La aplicación de lista de compras recibe el mensaje y utiliza la información para crear una lista de compras.

En resumen, una API es una herramienta que permite a las aplicaciones compartir información y funcionalidades entre sí.

¿Qué es Postman?

Es una plataforma integral que te permite desarrollar, probar, documentar y compartir APIs de forma intuitiva y eficiente. Imagina un laboratorio completo al alcance de tu mano, donde puedes dar vida a tus ideas, asegurarte de su correcto funcionamiento y compartirlas con el mundo. las cosas que puedes hacer con **Postman** son:

- **Crea peticiones HTTP:** Diseña peticiones a cualquier API, seleccionando el método adecuado (GET, POST, PUT, DELETE), la URL precisa, los encabezados necesarios y el cuerpo de la petición.
- **Envía y recibe datos:** Observa cómo tus peticiones cobran vida y recibe las respuestas de la API en diferentes formatos, como JSON, XML y HTML. Postman te permite visualizar la información de forma clara y organizada.
- **Pruebas exhaustivas:** Postman te brinda las herramientas para realizar pruebas a fondo y verificar que tu API funciona a la perfección. Experimenta con diferentes escenarios, valores de entrada y tipos de peticiones para asegurarte de su robustez.
- **Documentación impecable:** Genera documentación completa y profesional para tu API, incluyendo la descripción detallada de los endpoints, los parámetros y las respuestas. Comparte tu conocimiento con facilidad y facilita el uso de tu API por parte de otros.
- **Colaboración sin fronteras:** Trabaja en equipo de forma eficiente. Comparte tus colecciones de peticiones con otros miembros del equipo, agilizando el desarrollo y la colaboración.

¿Qué es el polimorfismo?

El polimorfismo es una de las características más fascinantes de la programación orientada a objetos. Permite que diferentes objetos respondan de forma diferente al mismo mensaje.

Imagina que tienes una clase Animal con lo que come. Los diferentes animales, como perros, gatos y pájaros, comerán de diferentes alimentos. El polimorfismo te permite definir el tipo de alimento que come cada animal, sin necesidad de modificar la clase Animal.

```
class Animal:
  def __init__(self, nombre):
    self.nombre = nombre

  def comer(self):
    raise NotImplementedError

class Perro(Animal):
  def comer(self):
    print(f"{self.nombre} come croquetas")

class Gato(Animal):
  def comer(self):
    print(f"{self.nombre} come atún")

class Pajaro(Animal):
  def comer(self):
    print(f"{self.nombre} come semillas")


perro = Perro("Mini")
gato = Gato("Jorge")
pajaro = Pajaro("Piolin")

perro.comer()
gato.comer()
pajaro.comer()
```

En este ejemplo vemos como la clase `Animal` define un método abstracto `comer`. Y las clases `Perro, Gato y Pajaro` heredan de la clase `Animal` y redefinen `comer` de forma específica para cada animal. Se crea un objeto de cada clase: `perro, gato y pajaro`. Se invoca el método `comer` en cada objeto. Se observa cómo cada objeto responde al mensaje comer de forma diferente.

Errores comunes en el uso del polimorfismo:

1. Errores en la definición de métodos:

- No definir el método abstracto en la clase padre.

- No redefinir el método abstracto en las clases hijas.
- Usar un nombre diferente para el método en la clase padre y en las clases hijas.
- No pasar los argumentos correctos al método en las clases hijas.

2. Errores en el uso de métodos:

- Llamar a un método que no está definido en la clase del objeto.
- Llamar a un método con argumentos incorrectos.
- No usar el tipo de dato correcto para el valor de retorno del método.

3. Errores en la comprensión del "duck typing":

- Asumir que todos los objetos tienen un método con el mismo nombre.
- No verificar si un objeto tiene un método antes de llamarlo.
- No usar el tipo de dato correcto para el valor de retorno del método.

¿Qué es un método dunder?

Los métodos dunber son métodos de Python que definen cómo se comportan los objetos de Python cuando se realizan operaciones comunes sobre ellos. Estos métodos se definen claramente con guiones bajos dobles antes y después del nombre del método. Un método común de dunder que quizás ya haya encontrado es el `__init__()` método que se utiliza para definir constructores de clases. Los métodos dunder no deben llamarse directamente en su código; más bien, el intérprete los llamará mientras el programa se está ejecutando.

Son útiles en la Programación Orientada a Objetos. Se especifica el comportamiento de sus tipos de datos personalizados cuando se usan con operaciones integradas comunes. Estas operaciones incluyen:

- Operaciones aritméticas
- Operaciones de comparación
- Operaciones de ciclo de vida
- Operaciones de representación

```
class Usuario:
  def __init__(self, nombre, correo):
    self.nombre = nombre
    self.correo = correo
```

```
  def __eq__(self, other):
    return self.nombre == other.nombre and self.correo == other.correo

usuario1 = Usuario("Robinson", "rbsnes@gmail.com")
usuario2 = Usuario("Robinson", "rbsnes@gmail.com")

print(usuario1 == usuario2)  # Salida: True
```

El código define una clase Usuario con dos métodos: `__init__` y `__eq__`. Mientras que `__init__` inicializa los atributos del objeto. El Dunder `__eq__` compara dos objetos y devuelve *True* si son iguales. Asi que este código crea dos objetos con los mismos atributos y los compara para ver si son iguales, si devuelve *True* indica que son iguales y por lo tanto el usuario puede estar repetido o esta en multiples bases de datos.

¿Qué es un decorador de python?

Un decorador de Python es una función que toma otra función como argumento y devuelve una nueva función. En otras palabras, es una forma de modificar el comportamiento de una función sin tener que modificar su código fuente.

Los decoradores se utilizan colocando la función decoradora antes de la función que se desea modificar, utilizando el símbolo @.

```
def decorador_acceso(funcion):
  def nueva_funcion(usuario):
    if usuario == "admin":
      return funcion()
    else:
      raise ValueError("Acceso denegado")
  return nueva_funcion

@decorador_acceso
def funcion_privada():
  return "Bienvenido Admin"

try:
  print(funcion_privada("usuario"))  # Acceso denegado
except ValueError as error:
  print(error)

print(funcion_privada("admin"))  # Bienvenido Admin
```

En este codigo el decorador `decorador_acceso` permite controlar el acceso a una función. El decorador comprueba si el usuario tiene permiso para acceder a la función. Si el usuario no tiene permiso, el decorador lanza una excepción `ValueError`.

Los decoradores se pueden utilizar para:

- **Añadir funcionalidad:** Se puede usar un decorador para agregar funcionalidad a una función sin tener que modificar su código fuente. Por ejemplo, se puede usar un decorador para registrar el tiempo que tarda una función en ejecutarse.
- **Validar la entrada:** Se puede usar un decorador para validar la entrada de una función y evitar que se ejecute si la entrada no es válida.
- **Controlar el acceso:** Se puede usar un decorador para controlar qué usuarios tienen acceso a una función.

Ventajas de los decoradores:

- **Hacen que el código sea más modular y reutilizable:** Se puede separar la lógica de la función de la lógica del decorador, lo que hace que el código sea más fácil de entender y mantener.
- **Permiten escribir código más elegante y conciso:** Se puede evitar la duplicación de código utilizando decoradores.
- **Facilitan la prueba de código:** Se pueden probar los decoradores de forma independiente a las funciones que modifican.

Desventajas de los decoradores:

- **Pueden hacer que el código sea más difícil de entender:** Si se usan demasiados decoradores, el código puede ser difícil de seguir.
- **Pueden dificultar la depuración de código:** Si hay un error en un decorador, puede ser difícil identificar la causa del error.