M4_AE5_ABPRO-Ejercicio grupal

Integrantes:

- Jorge Cárdenas
- Hans Schiess
- Catalina Villegas
-

Link GitHub, Código visual Studio:

https://github.com/CatalinaMonserrat/Python_modulo_4/tree/main/ejercicio_BIKECITY

Contexto

Tres amigos han creado un emprendimiento llamado "BIKECITY", un servicio de renta de bicicletas urbanas. Los clientes deben llamar o enviar mensajes a los dueños para reservar una bicicleta, pero el sistema manual que utilizan ha generado los siguientes problemas:

- **Reservas duplicadas por falta de sincronización entre los socios.**
- **Clientes que no recogen las bicicletas reservadas, generando pérdidas.**
- **Cobros incorrectos debido a errores en los cálculos manuales.**
- **Falta de registro sobre las bicicletas disponibles y su estado.**

Solución

Se necesita desarrollar un sistema automatizado que permita:

- **Registrar bicicletas disponibles para renta.**
- **Gestionar reservas evitando conflictos de horario.**
- **Aplicar cobros correctos según la duración del uso.**
- **Controlar el estado de cada bicicleta para evitar pérdidas o mal uso.**

Paso 1: Conceptualización y Análisis
Cada equipo debe investigar y responder en un documento las siguientes preguntas:

- **¿Qué es una excepción en programación y por qué es importante manejarla correctamente?**

En cualquier lenguaje de programación, las excepciones son eventos o situaciones inesperadas que interrumpen el flujo normal de ejecución de un programa. En Python, una excepción es un objeto que describe un error que ocurrió durante la ejecución del código. Si no se maneja, la excepción detendrá el programa.

En lugar de permitir que el programa se cierre abruptamente, Python nos ofrece una forma estructurada de capturar y manejar estas excepciones, lo que ayuda a garantizar que nuestro programa continúe funcionando de manera controlada.

- **¿Cuáles son los tipos de excepciones más comunes?**

Existen varios tipos de excepciones predefinidas en Python. Algunas de las más comunes incluyen:

- **ZeroDivisionError**: Ocurre cuando intentas dividir un número por cero.
- **ValueError**: Se genera cuando una operación recibe un valor inapropiado.
- **TypeError**: Se lanza cuando una operación o función es aplicada a un objeto de tipo incorrecto.
- **FileNotFoundError**: Se lanza cuando se intenta abrir un archivo que no existe.
- **IndexError**: Aparece cuando se accede a un índice que no existe en una lista o tupla.
- **¿Cómo funciona la sentencia try/except y cuándo se debe utilizar?**

Python utiliza una estructura `try-except` para manejar excepciones. La estructura básica se ve así:

```
    try:
    # Bloque de código que puede causar una excepción
        x = 10 / 0
    except ZeroDivisionError:
    # Bloque de código que se ejecuta si ocurre la excepción
    print("No puedes dividir por cero")
```

En el ejemplo anterior:

try: contiene el código que podría generar una excepción.

except: maneja el error si ocurre, en este caso, una división por cero.

- **¿Cómo se pueden capturar múltiples excepciones en un solo bloque de código?**

Se pueden manejar múltiples excepciones en un solo bloque try-except utilizando varias cláusulas except. Esto permite que se capture una variedad de errores sin necesidad de escribir un bloque except para cada uno de ellos.

- **¿Qué es el uso de raise en Python y cómo se utiliza para generar excepciones en validaciones?**

En ocasiones, es necesario que un programa lance una excepción de manera explícita. Esto se puede hacer utilizando la palabra clave `raise`.

```
        def dividir(a, b):
    if b == 0:
        raise ZeroDivisionError("No se puede dividir por cero")
    return a / b
try:
    resultado = dividir(10, 0)
except ZeroDivisionError as e:
    print(e)
```

- **¿Cómo se pueden definir excepciones personalizadas y en qué casos sería útil?**

En Python, también es posible definir nuestras propias excepciones personalizadas. Para hacerlo, simplemente creamos una nueva clase que herede de la clase base `Exception`.

- **¿Cuál es la función de finally en el manejo de excepciones?**

Acciones de limpieza con `finally`

El bloque `finally` es útil para garantizar que ciertos códigos se ejecuten siempre, incluso si ocurre una excepción o no. Es ideal para realizar acciones de limpieza, como cerrar archivos, liberar recursos o terminar conexiones de red.

El bloque `finally` se ejecuta sin importar si la excepción se ha manejado correctamente o no. Esto lo convierte en un lugar adecuado para la liberación de recursos que deben cerrarse independientemente del resultado del bloque `try`.

- **¿Cuáles son algunas acciones de limpieza que deben ejecutarse después de un proceso que puede generar errores?**

Es importante señalar que el manejo de excepciones permite ejecutar medidas de control ante situaciones que potencialmente podrían generar ciertas inconsistencias en la ejecución de código. Algunas acciones de limpieza de uso común son:

Manejo de Excepciones*:** Antes que todo, para el manejo de excepciones, se puede utilizar la sintaxis ***try/***except***/***finally*** para definir un código con potencial de error, y su manejo a través de ***except***. Luego, ***finally*** se usa para acciones que se ejecutan cualquiera sea la salida del ***except*** anterior.

```
try:
   # Código que puede generar errores
except Exception as e:
   print(f"Ocurrió un error: {e}")
finally:
   # Código de limpieza
```

Cierre de Archivos: Luego de accesar a un archivo, a través del manejo de excepciones se puede automatiza el cerrado del documento:

```
try:
   file = open("archivo.txt", "r")
   # Operaciones con el archivo
finally:
   file.close()
```

Cabe señalar que otra estrategia para manejar el cierre de ficheros puede ser el uso de ***with***, el cual abre un archivo y ejecuta acciones. Luego de efectuadas las acciones dentro de la sección ***with***, el fichero accesado se cerrará por defecto:

```
with open("archivo.txt", "r") as file:
   # Operaciones con el archivo
# El archivo se cierra automáticamente al salir del bloque with
```

Liberar Recursos de Red: a través de recursos try/finally, se pueden cerrar conexiones de red:

```
import requests
try:
   response = requests.get("https://ejemplo.com")
   # Procesamiento de la respuesta
finally:
   response.close()
```

Cerrar conexiones a BBDD:

```
import sqlite3
conn = sqlite3.connect("base_de_datos.db")
try:
   cursor = conn.cursor()
   # Operaciones con la base de datos
finally:
   conn.close()
```

Paso 2: Implementación en Código
Cada equipo debe implementar un código en Python que simule el sistema de reservas y manejo de bicicletas, utilizando los conceptos de manejo de excepciones.

Tareas del equipo:

- **Crear clases para representar bicicletas y reservas.**
- **Aplicar try/except para manejar errores en el sistema.**
- **Capturar múltiples excepciones en el mismo bloque de código.**
- **Usar raise para generar excepciones cuando haya errores en las reservas.**
- **Definir una excepción personalizada para manejar casos específicos.**
- **Usar finally para acciones de limpieza, como cerrar conexiones o registrar información en logs.**