Ejercicio grupal M4_AE5_ABPRO

Emily Quintana
Johana Torres
Linwi Vargas
Natalia Rodriguez

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 esperada:

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.

Requerimiento de la actividad:

Cada equipo debe investigar y responder en un documento las siguientes preguntas:

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

R. Las excepciones son eventos o situaciones inesperadas que interrumpen el flujo normal de ejecución de un programa. Si no se maneja, la excepción detendrá el programa Y puede causar una mala experiencia al usuario y la posible pérdida de datos.

Python nos ofrece una forma estructurada de capturar y manejar estas excepciones para no permitir que el programa se cierre abruptamente,lo que ayuda a garantizar que nuestro programa continúe funcionando de manera controlada.

2) **¿Cuáles son los tipos de excepciones más comunes?**

R. Las excepciones más comunes son:

ZeroDivisionError: Ocurre al intentar dividir un número por cero. Ejemplo: resultado = 10 / 0

ValueError: Se genera cuando una función recibe un argumento del tipo correcto pero un valor inadecuado. Ejemplo: numero = int("hola")

TypeError: Se genera cuando se aplica una operación o función a un objeto de tipo inadecuado. Ejemplo: resultado = 1 + "2"

FileNotFoundError: Aparece cuando el archivo que se intenta abrir no se encuentra por errores de ruta o tipográficos, no se tienen los permisos para acceder o no existe. Ejemplo: archivo.txt variable = open("archiBo.txt", "r")

IndexError: Aparece cuando se accede a un índice en una lista o tupla que no existe. Ejemplo: list = [a,b,c] print (list[5])

3) ¿Cómo funciona la sentencia try/except y cuándo se debe utilizar?

R. Es una herramienta en Python para manejar errores y excepciones de una manera controlada. En lugar de que tu programa se detenga abruptamente cuando ocurre un problema, puedes "atrapar" ese error y decidir qué hacer al respecto, además hace del programa más amigable para el usuario, ya que en lugar de que el programa falle sin explicación, se puede mostrar un mensaje de error claro o tomar una acción alternativa.

La sentencia try funciona de la siguiente manera:

- Primero, se ejecuta la cláusula try (la(s) linea(s) entre las palabras reservadas try y la except).

- Si no ocurre ninguna excepción, la cláusula except se omite y la ejecución de la cláusula try finaliza.

- Si ocurre una excepción durante la ejecución de la cláusula try, se omite el resto de la cláusula. Luego, si su tipo coincide con la excepción nombrada después de la palabra clave except, se ejecuta la cláusula except, y luego la ejecución continúa después del bloque try/except.

- Si ocurre una excepción que no coincide con la excepción nombrada en la cláusula except, se pasa a instrucciones try externas; si no se encuentra ningún controlador, es una excepción no controlada y la ejecución se detiene con un mensaje de error.

¿Cuándo se debe utilizar Try?

R.Debes usar try/except en cualquier situación cuando se anticipe que un error podría ocurrir y quieres manejarlo de forma elegante en lugar de dejar que el programa falle.

Algunos escenarios comunes:

- Manipulación de la entrada del usuario: Cuando pides al usuario que ingrese datos, siempre existe la posibilidad de que no ingrese el formato esperado (un número en lugar de una cadena, un archivo que no existe, etc.).

- Operaciones con archivos: Al intentar abrir, leer o escribir en un archivo, es posible que el archivo no exista (FileNotFoundError) o que no tengas los permisos necesarios.

- Conexiones de red: Las conexiones a servidores o bases de datos pueden fallar por muchas razones (servidor caído, falta de conexión a internet, etc.).

- Conversiones de tipo de dato: Al intentar convertir una cadena a un número (int(), float()), si la cadena no tiene el formato correcto, se producirá un ValueError.

- Índices de listas o claves de diccionarios: Al acceder a un elemento en una lista o diccionario, si el índice o la clave no existe, se generará un error (IndexError o KeyError).

4) ¿Cómo se pueden capturar múltiples excepciones en un solo bloque de código?

Una declaración try puede tener más de una cláusula *excep*t, para especificar gestores para diferentes excepciones. Como máximo, se ejecutará un gestor. Los gestores solo manejan las excepciones que ocurren en la cláusula try correspondiente, no en otros gestores de la misma declaración try. Una cláusula *except* puede nombrar múltiples excepciones como una tupla entre paréntesis.

Puedes capturar múltiples excepciones de dos maneras principales, dependiendo de si se quieren manejar cada una de forma diferente o si se quieren tratar todas de la misma manera.

1. Capturar cada excepción con un *except* diferente

Esta es la forma más común y recomendada si necesitas tomar una acción específica para cada tipo de error. Por ejemplo, si un error de división por cero debe tener un mensaje diferente a un error de valor.

Ejemplo:

```
try:
    # Código que podría causar errores
    numerador = int(input("Ingresa el numerador: "))
    denominador = int(input("Ingresa el denominador: "))

    resultado = numerador / denominador
    print(f"El resultado es: {resultado}")

except ValueError:
    print("Error: Por favor, ingresa solo números enteros.")
```

```
except ZeroDivisionError:
    print("Error: No puedes dividir por cero.")
```

En este caso, si el usuario ingresa "hola" en lugar de un número, el bloque except ValueError se ejecuta. Si ingresa 5 y 0, se ejecuta el bloque except ZeroDivisionError.

2. Capturar múltiples excepciones en un solo *except*

Si quieres ejecutar el mismo código sin importar el tipo de excepción, puedes agrupar las excepciones en una tupla dentro del mismo bloque *except*.

Ejemplo:

```
try:
    # Código que podría causar errores
    numerador = int(input("Ingresa el numerador: "))
    denominador = int(input("Ingresa el denominador: "))

    resultado = numerador / denominador
    print(f"El resultado es: {resultado}")

except (ValueError, ZeroDivisionError):
    print("Ha ocurrido un error. Asegúrate de ingresar números válidos y de que el denominador no sea cero.")
```

Aquí, si ocurre un ValueError o un ZeroDivisionError, el mismo mensaje genérico se imprimirá en ambos casos. Esto es útil cuando el manejo del error es el mismo para todos los posibles problemas.

Usar *except* separados para cada tipo de excepción, permite ser más específico y ofrecer mensajes de error más útiles para el usuario. La opción de la tupla es ideal cuando sabes que el manejo de las excepciones es idéntico y no necesitas diferenciar entre ellas.

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

R. El uso de raise en Python sirve para generar una excepción de forma manual cuando se detecta una situación que no cumple con ciertas condiciones dentro del programa. Es muy útil al hacer validaciones, ya que permite detener el flujo del programa de manera controlada cuando algo no está bien. Por ejemplo, si un valor ingresado no es válido o si se presenta un caso que podría causar un error, se puede usar raise para lanzar una advertencia específica. Esto ayuda a que el programa sea más seguro, claro y predecible, ya que se anticipa a posibles errores antes de que ocurran.

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

R. Las excepciones personalizadas se utilizan cuando se necesita identificar y manejar errores muy específicos que no están contemplados por las excepciones comunes que ofrece Python. Para definir una excepción personalizada, se crea una nueva clase que debe

basarse en la clase general de excepciones de Python. A esta clase se le puede agregar un mensaje o comportamiento específico, dependiendo de lo que se quiera controlar. Este tipo de excepciones es útil cuando el programa tiene ciertas reglas propias o situaciones particulares que no encajan en los errores típicos del lenguaje. Por ejemplo, se puede usar una excepción personalizada para advertir cuando se rompe una regla de negocio en una aplicación, como intentar registrar un usuario con un correo ya existente, o realizar una acción no permitida en determinado contexto. Definir excepciones personalizadas permite que el código sea más claro, organizado y fácil de mantener, ya que se pueden identificar rápidamente los errores según su tipo y actuar en consecuencia.

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

R. La función de finally en el manejo de excepciones consiste en mantener el funcionamiento del programa, ya que siempre ejecutará el bloque de código exista una excepción o no, su uso es encargarse de cerrar archivos, liberar recursos o terminar conexiones de red, este se ejecutará sin importar si la excepción fue manejada por try. Esto lo convierte en la función adecuada para la liberación de recursos que deben cerrarse independientemente del resultado del manejo de la excepción.

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

R. Después de ejecutar un proceso que puede generar errores, es importante realizar ciertas acciones de limpieza para asegurar que los recursos utilizados se liberen correctamente y el sistema no quede en un estado inestable. Algunas de estas acciones incluyen:

- Cerrar archivos abiertos una vez que ya no se necesitan, para evitar problemas de bloqueo o pérdida de datos.
- Cerrar conexiones a bases de datos o redes, para liberar recursos del sistema y evitar saturaciones o errores futuros.
- Liberar memoria o recursos temporales que hayan sido utilizados durante el proceso.
- Restablecer configuraciones que hayan sido modificadas durante la ejecución del programa.
- Eliminar archivos temporales que se hayan creado como parte del proceso.

Estas acciones se deben ejecutar siempre, incluso si ocurrió un error, para mantener el programa funcionando de forma estable y segura. Realizarlas garantiza que el sistema no quede con procesos abiertos o datos en mal estado, lo que también mejora la experiencia del usuario y la calidad del software.Es recomendable realizar siempre acciones de limpieza después de manipular archivos, como cerrar el archivo o liberar recursos. Usar finally o el contexto with ayuda a gestionar estos aspectos automáticamente.

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.

Paso 3: Documentación y Entrega

El equipo debe entregar un archivo comprimido (.RAR o .ZIP) que contenga:

Documento con las respuestas del análisis.

Código Python con el sistema de reservas y manejo de excepciones.