1. MANEJO DE ERRORES

Al programar en Python, puedes cometer varios tipos de errores. Aquí tienes una lista de los errores más comunes:

- Errores de sintaxis: Estos errores ocurren cuando el código no sigue la estructura y reglas de sintaxis de Python. Pueden incluir olvidar dos puntos (:) al definir un bucle o una función, olvidar cerrar paréntesis o comillas, entre otros.
- Errores de tiempo de ejecución: Estos errores ocurren durante la ejecución del programa y generalmente se deben a problemas lógicos o a datos inesperados. Algunos ejemplos comunes incluyen divisiones por cero, índices fuera de rango, errores de conversión de tipos, etc.
- Errores lógicos: Estos errores ocurren cuando la lógica del programa es incorrecta y no produce los resultados esperados. Puede deberse a una mala comprensión del problema, mal uso de condiciones, bucles o variables, o errores en algoritmos y cálculos.
- Errores de nombre de variable: Estos errores ocurren cuando intentas usar una variable que no ha sido definida previamente o cuando el nombre de la variable está mal escrito. Python tratará de interpretarla como una nueva variable y generará un error.
- Errores de importación: Estos errores ocurren cuando intentas importar un módulo o una función que no existe o que no está disponible en tu entorno de Python. También pueden ocurrir si hay errores en la estructura del código del módulo que intentas importar.
- Errores de archivo: Estos errores ocurren cuando hay problemas al abrir, leer o escribir archivos. Puede ser debido a que el archivo no existe, no tienes los permisos necesarios o hay problemas de formato de archivo.
- Errores de lógica de programación: Estos errores son más difíciles de detectar y pueden ocurrir cuando la lógica detrás del programa es incorrecta o incompleta. El programa puede ejecutarse sin generar errores, pero los resultados no serán los esperados.

2. EXCEPCIONES

En programación, una excepción es un evento que ocurre durante la ejecución de un programa y interrumpe el flujo normal de ejecución. Las excepciones generalmente se generan cuando ocurren situaciones

inesperadas o errores que el programa no puede manejar automáticamente.

Las excepciones son útiles porque permiten identificar y manejar errores de manera controlada. En lugar de que el programa se bloquee por completo cuando ocurre un error, las excepciones brindan una forma de detectar y responder a errores de manera específica, evitando que el programa se detenga abruptamente.

2.1. MANEJO DE EXCEPCIONES

En Python, el manejo de excepciones se realiza utilizando los bloques try, except, else y finally.

try y except: Estos bloques se utilizan en conjunto para capturar y manejar excepciones específicas. El código que potencialmente puede generar una excepción se coloca dentro del bloque try, y si ocurre una excepción, se captura en el bloque except.

Ejemplo:

```
try:
    numero = int(input("Ingrese un número: "))
    resultado = 10 / numero
    print("El resultado es:", resultado)
except ValueError:
    print("Error: Ingrese un número válido.")
except ZeroDivisionError:
    print("Error: No se puede dividir entre cero.")
```

En este ejemplo, el bloque try intenta realizar una división y captura las excepciones ValueError y ZeroDivisionError de manera específica. Si se ingresa un valor no numérico, se ejecutará el bloque except ValueError. Si se intenta dividir entre cero, se ejecutará el bloque except ZeroDivisionError.

else: Este bloque se ejecuta si no se produce ninguna excepción en el bloque try. Podés usarlo para ejecutar código adicional después de que se haya ejecutado el bloque try correctamente.

```
try:
    numero = int(input("Ingrese un número: "))
    resultado = 10 / numero
```

```
except ValueError:
    print("Error: Ingrese un número válido.")
except ZeroDivisionError:
    print("Error: No se puede dividir entre cero.")
else:
    print("El resultado es:", resultado)
```

En este caso, si se ingresa un valor no numérico o se intenta dividir entre cero, se ejecutarán los bloques except. Pero si no se produce ninguna excepción, se ejecutará el bloque else y se imprimirá el resultado.

finally: Este bloque se ejecuta siempre, independientemente de si se produce una excepción o no. Se utiliza para realizar tareas de limpieza o liberar recursos, como cerrar archivos o conexiones a bases de datos.

```
try:
    archivo = open("datos.txt", "r")
    # Realizar operaciones con el archivo
except FileNotFoundError:
    print("Error: Archivo no encontrado.")
finally:
    archivo.close()
```

En este ejemplo, el bloque finally se asegura de que el archivo se cierre correctamente, incluso si se produce una excepción o no. Esto evita dejar archivos abiertos accidentalmente y garantiza una liberación adecuada de recursos.

Se pueden combinar los bloques except, else y finally según las necesidades y manejar diferentes tipos de excepciones de manera específica.

2.2. PROCESAMIENTO Y PROPAGACIÓN DE EXCEPCIONES

Cuando se genera una excepción en un bloque de código, el flujo normal de ejecución se interrumpe y el control se transfiere al bloque except correspondiente para manejar la excepción. Sin embargo, si no se encuentra un bloque except adecuado dentro del mismo bloque try, la excepción se propaga al bloque try exterior o a niveles superiores en la pila de llamadas hasta que se encuentre un bloque except que pueda manejarla o, en caso contrario, el programa se detiene y muestra un rastreo de la excepción.

Ejemplo:

```
def funcion_b():
    resultado = 10 / 0  # División por cero
def funcion_a():
    funcion_b()



try:
    funcion_a()
except ZeroDivisionError:
    print("Error: División por cero")
```

En este ejemplo, se llama a la función funcion_a(), que a su vez llama a funcion_b(). Dentro de funcion_b(), se realiza una división por cero, lo que genera una excepción ZeroDivisionError. Dado que no hay un bloque try-except dentro de funcion_b(), la excepción se propaga hacia el bloque try en funcion_a(), donde se encuentra un bloque except adecuado para manejarla. Como resultado, se imprime el mensaje "Error: División por cero".

El proceso de propagación de excepciones permite que las excepciones se manejen en diferentes niveles del programa, lo que proporciona una forma efectiva de controlar y responder a las situaciones de error en el código.

Es importante tener en cuenta que si una excepción se propaga hasta el nivel superior sin ser capturada, el programa se detendrá y mostrará un rastreo de la excepción, que incluye información sobre la excepción y la secuencia de llamadas de funciones que la llevaron a ser propagada. Este rastreo es útil para depurar y comprender qué causó la excepción.

Además, podés utilizar la palabra clave raise para generar manualmente una excepción en cualquier parte de tu código. Esto puede ser útil cuando deseas indicar una condición de error específica y controlar cómo se maneja.

Espero que esta explicación aclare cómo funciona el procesamiento y la propagación de excepciones en Python. Si tienes más preguntas, no dudes en hacerlas.

2.3. RAISE

La palabra clave raise en Python se utiliza para generar manualmente una excepción en un punto específico del código. Esto te permite indicar que se ha producido una condición de error o excepción personalizada y controlar cómo se maneja.

La sintaxis básica para usar raise es la siguiente:

raise TipoDeExcepcion("Mensaje de error")

Donde TipoDeExcepcion es el tipo de excepción que deseas generar y "Mensaje de error" es un mensaje descriptivo opcional que puedes incluir para proporcionar más información sobre la excepción.

Ejemplo:

```
def dividir(a, b):
    if b == 0:
        raise ZeroDivisionError("No se puede dividir entre cero")
    return a / b


try:
    resultado = dividir(10, 0)
except ZeroDivisionError as e:
    print("Error:", str(e))
```

En este ejemplo, la función dividir() comprueba si el divisor b es cero. Si es cero, se genera manualmente una excepción ZeroDivisionError utilizando la palabra clave raise y se proporciona un mensaje de error personalizado. Luego, se captura la excepción con el bloque except y se imprime el mensaje de error.

Podés utilizar raise para generar excepciones de cualquier tipo, ya sea las excepciones incorporadas en Python, como ValueError, TypeError, etc., o incluso puedes crear tus propias excepciones personalizadas creando una clase de excepción.

Ejemplo:

```
class MiExcepcion(Exception):
    pass
```

```
def mi_funcion():
    raise MiExcepcion("Ocurrió un error personalizado")


try:
    mi_funcion()
except MiExcepcion as e:
    print("Error:", str(e))
```

En este ejemplo, se crea una clase de excepción personalizada llamada MiExcepcion que hereda de la clase base Exception. Luego, dentro de la función mi_funcion(), se genera manualmente una instancia de MiExcepcion utilizando raise. Posteriormente, se captura la excepción con el bloque except y se imprime el mensaje de error.

La palabra clave raise te brinda flexibilidad para generar y controlar excepciones de manera explícita en tu código, permitiéndote señalar condiciones de error específicas y manejarlas de acuerdo a tus necesidades.

3. VALIDACIONES

En Python, las validaciones son mecanismos utilizados para verificar si ciertas condiciones se cumplen en un programa. Se utilizan para garantizar que los datos o las suposiciones sean correctos antes de continuar con la ejecución del código.

Una forma común de realizar validaciones en Python es mediante la declaración if. Puedes usar condiciones if para verificar una expresión y ejecutar un bloque de código si la condición es verdadera. Ejemplo:

```
def dividir(a, b):
    if b != 0:
        resultado = a / b
        return resultado
    else:
        print("Error: No se puede dividir entre cero")
```

```
resultado = dividir(10, 5)

if resultado is not None:
    print("El resultado de la división es:", resultado)
```

En este ejemplo, la función dividir() verifica si el divisor b es diferente de cero antes de realizar la división. Si b es cero, se imprime un mensaje de error. Después de llamar a la función dividir(), se verifica si el resultado es diferente de None antes de imprimirlo.

Otra forma de hacer validaciones en Python es utilizando la declaración assert. La declaración assert se utiliza para verificar que una expresión sea verdadera y, si no lo es, genera una excepción AssertionError. Ejemplo:

```
def dividir(a, b):
    assert b != 0, "Error: No se puede dividir entre cero"
    resultado = a / b
    return resultado


resultado = dividir(10, 5)
print("El resultado de la división es:", resultado)
```

En este ejemplo, la expresión b != 0 se verifica utilizando la declaración assert. Si la expresión es falsa, se genera una excepción AssertionError con el mensaje de error especificado. Si la expresión es verdadera, la ejecución continúa normalmente y se realiza la división.

Es importante tener en cuenta que las declaraciones assert se utilizan principalmente para detectar errores de programación y no para manejar errores esperados o condiciones excepcionales. Por lo general, se utilizan durante el desarrollo y las pruebas, y se recomienda desactivarlas en entornos de producción.

En resumen, las validaciones en Python se realizan utilizando declaraciones if para verificar condiciones y ejecutar código en consecuencia. La declaración assert se utiliza para verificar expresiones y generar una excepción si la expresión es falsa.

3.1. VALIDACIONES ENTRADA y TIPOS DE DATOS

La validación de datos de entrada del usuario y de tipos es una parte importante de cualquier programa para garantizar que los datos

ingresados sean correctos y cumplan con los requisitos esperados. Aquí te presento algunas técnicas comunes para validar los datos de entrada en Python:

Validación de tipos de datos:

type(): Podés utilizar la función type() para verificar el tipo de dato de una variable. Por ejemplo:

```
edad = input("Ingrese su edad: ")

if type(edad) != int:

    print("Error: La edad debe ser un número entero")
```

isinstance(): La función isinstance() permite verificar si un objeto es una instancia de una clase o de una clase derivada. Por ejemplo:

```
numero = input("Ingrese un número: ")

if not isinstance(numero, int):

    print("Error: Debe ingresar un número entero")
```

Validación de rangos:

Podés verificar si un valor se encuentra dentro de un rango específico utilizando operadores de comparación, como > (mayor que), < (menor que), >= (mayor o igual que), <= (menor o igual que), etc. Por ejemplo:

```
edad = int(input("Ingrese su edad: "))

if edad < 0 or edad > 120:

    print("Error: La edad debe estar entre 0 y 120")
```

Validación de formatos:

Podés utilizar expresiones regulares para validar cadenas de texto según un formato específico. El módulo re de Python proporciona funciones para trabajar con expresiones regulares. Por ejemplo, para validar un número de teléfono en un formato determinado:

```
import re

telefono = input("Ingrese su número de teléfono: ")

patron = r"\d{3}-\d{3}-\d{4}"

if not re.match(patron, telefono):

    print("Error: El número de teléfono debe tener el formato
XXX-XXX-XXXX")
```

Estos son solo algunos ejemplos básicos de validación de datos en Python. La elección de la técnica de validación depende de los requisitos específicos de tu programa y del tipo de datos que estés validando.

Recordá que la validación de datos es una capa adicional de seguridad y control, pero no es suficiente por sí sola. También es importante considerar otras formas de manejo de errores, como el manejo de excepciones, para garantizar un comportamiento adecuado del programa en situaciones inesperadas.