TRATAMIENTO DE ERRORES Y ROBUSTEZ EN PROGRAMACIÓN CON PYTHON

Introducción

En programación, los errores son inevitables. Sin embargo, un buen manejo de errores y la implementación de prácticas que promuevan la robustez del código pueden minimizar problemas y mejorar la experiencia del usuario. Este tutorial cubre cómo manejar errores en Python, cómo diseñar código robusto y las mejores prácticas para prevenir errores comunes.

1. Entendiendo los Errores en Python

Los errores en Python se dividen en dos categorías principales:

- **Errores de sintaxis** (SyntaxError): Ocurren cuando el código no sigue las reglas de sintaxis de Python (por ejemplo, olvidar un `:` después de un bloque `if`). Estos son detectados por el intérprete antes de ejecutar el programa.
- **Errores en tiempo de ejecución (RuntimeError):** que generan **Excepciones**. Son errores que ocurren durante la ejecución del programa, como intentar dividir por cero o acceder a un archivo inexistente.
- **Errores semánticos:** el programa se ejecuta pero no realiza correctamente lo esperado. Estos son más difíciles de detectar y requieren depuración.

Python proporciona un sistema robusto para manejar excepciones usando bloques `try-except`, lo que permite capturar y gestionar errores sin que el programa falle abruptamente.

2. Manejo de Excepciones con `try-except`

El bloque `try-except` es la herramienta principal para manejar excepciones en Python. Aquí está la estructura básica:

```
try:
    # Código que podría generar una excepción
    resultado = 10 / 0
except ZeroDivisionError:
    # Código que se ejecuta si ocurre la excepción
    print("Error: No se puede dividir por cero.")
```

- El código dentro de `try` se intenta ejecutar.
- Si ocurre una excepción que coincida con `except`, se ejecuta el código ahí dentro.
- Si no ocurre error, el bloque `except` se ignora.

Ejemplo Completo

En realidad podemos hacer un uso más complejo de las excepciones:

- `else`: Código que se ejecuta si **no hubo errores** en el bloque try.
- `finally`: Código que se ejecuta siempre, ocurra o no una excepción, útil para liberar recursos.

```
try:
    numero = int(input("Ingresa un número: "))
    resultado = 100 / numero
    print(f"El resultado es {resultado}")
except ValueError:
    print("Error: Ingresa un número válido.")
except ZeroDivisionError:
    print("Error: No se puede dividir por cero.")
else:
    print("Operación realizada con éxito.")
finally:
    print("Bloque finalizado.")
```

Explicación: - `try`: Contiene el código que puede generar una excepción. - `except`: Captura excepciones específicas (puedes manejar múltiples tipos de excepciones). - `else`: Se ejecuta si no ocurre ninguna excepción. - `finally`: Se ejecuta siempre, independientemente de si hubo una excepción o no (útil para limpieza, como cerrar archivos).

Captura de múltiples excepciones

Puedes manejar varias excepciones específicas:

```
try:
    # Código arriesgado
    pass
except (IOError, ValueError) as e:
    print(f"Ocurrió un error: {e}")
```

O múltiples bloques `except` separados para distintos tipos.

Mejores Prácticas para `try-except`

1. Especifica excepciones concretas: Evita usar `except` sin especificar el tipo de excepción, ya que capturar todas las excepciones (`except Exception`) puede ocultar errores inesperados.

```
# Mal
try:
   numero = int("abc")
except:
   print("Algo salió mal.") # No especifica qué error

# Bien
try:
   numero = int("abc")
except ValueError:
   print("Error: Entrada no válida, debe ser un número.")
```

2. Captura solo las excepciones que sabes manejar: evitar usar `except Exception` sin control específico para no ocultar errores inesperados.

3. Reporta o registra siempre el error: no ocultes información que pueda ayudar a entender qué falló.

4. Usa `raise` para propagar errores después de manejar si necesitas que el programa los maneje

más arriba.

5. Evita bloques `try` demasiado amplios: Incluye solo el código que puede generar una excepción específica para mantener la claridad.

6. Usa `finally` para recursos críticos: Por ejemplo, cerrar archivos o conexiones a bases de datos.

```
try:
   archivo = open("datos.txt", "r")
   contenido = archivo.read()
except FileNotFoundError:
   print("Error: El archivo no existe.")
finally:
   archivo.close()
```

3. Creando Excepciones Personalizadas

Puedes definir tus propias excepciones para manejar casos específicos en tu aplicación. Esto mejora la legibilidad y permite un control más granular.

```
class ErrorValorNegativo(Exception):
    """Excepción personalizada para valores negativos."""
    pass

def calcular_raiz_cuadrada(numero):
    if numero < 0:
        raise ErrorValorNegativo("No se puede calcular la raíz cuadrada de un
número negativo.")
    return numero ** 0.5

try:
    resultado = calcular_raiz_cuadrada(-4)
except ErrorValorNegativo as e:
    print(f"Error: {e}")
else:
    print(f"La raíz cuadrada es {resultado}")
```

Buena práctica: Usa nombres descriptivos para excepciones personalizadas y proporciona mensajes de error claros.

4. Buenas Prácticas para Evitar Errores

Las **buenas prácticas para prevenir errores en Python** durante la programación se basan en escribir código limpio, claro y seguro, además de utilizar herramientas y enfoques que minimicen la aparición de fallos. Estas son las más relevantes:

Validación de Entradas

Es fundamental **validar y sanear la entrada de datos**, especialmente la proveniente de usuarios o fuentes externas, para evitar errores y vulnerabilidades como inyecciones de código. Implementa listas blancas o filtros estrictos sobre lo que aceptas como válido.

```
def dividir_numeros(a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        raise ValueError("Los argumentos deben ser números.")
    if b == 0:
        raise ZeroDivisionError("El divisor no puede ser cero.")
    return a / b
```

Uso de nombres descriptivos y consistentes

Emplea **nombres claros y representativos para variables, funciones y módulos**. Esto facilita la lectura, comprensión y mantenimiento del código, permitiendo detectar errores con mayor rapidez.

Uso de Tipos y Anotaciones

Las anotaciones de tipo (type hints) mejoran la legibilidad y permiten a herramientas como `mypy` detectar errores antes de la ejecución:

```
def sumar(a: float, b: float) -> float:
    return a + b
```

Utiliza un buen editor con resaltado y linters

Edición con herramientas que soporten **resaltado de sintaxis y comprobación automática de errores (linters)** ayuda a detectar fallos rápidamente antes de ejecutar el programa.

Maneja correctamente excepciones y errores

No ignores las excepciones: usa bloques **try-except** para controlar errores esperados y evitar que el programa termine abruptamente. Además, utiliza bloques **finally** o el contexto `with` para asegurar el correcto cierre de recursos como archivos.

Organiza el código en funciones y módulos

Divide el código en piezas pequeñas, reutilizables y fáciles de probar mediante funciones y módulos. Las funciones deben ser pequeñas y cumplir una única tarea para reducir errores.

Documentación Clara

Documenta las secciones complejas o importantes con comentarios y docstrings claros. Esto ayuda a entender el propósito y funcionamiento, facilitando la detección y corrección de errores futuros.

Usa docstrings para documentar funciones, indicando posibles excepciones:

```
def abrir_archivo(nombre_archivo: str) -> str:
    """
    Abre y lee un archivo de texto.

    Args:
        nombre_archivo (str): Ruta del archivo.

    Returns:
```

```
        str: Contenido del archivo.

    Raises:
        FileNotFoundError: Si el archivo no existe.
        PermissionError: Si no hay permisos para leer el archivo.
    """
    with open(nombre_archivo, 'r') as archivo:
        return archivo.read()
```

Manejo de Recursos con Context Managers

Usa `with` para manejar recursos como archivos o conexiones a bases de datos, ya que asegura que se cierren correctamente incluso si ocurre una excepción:

```
with open("ejemplo.txt", "w") as archivo:
    archivo.write("Hola, mundo!")
# El archivo se cierra automáticamente, incluso si hay un error
```

Pruebas Unitarias

Implementa **pruebas automatizadas** para verificar que cada componente funciona correctamente antes de integrarlo en el programa. Esto minimiza errores en el desarrollo y mantenimiento.

Escribe pruebas unitarias con `unittest` o `pytest` para verificar el comportamiento de tu código en casos extremos:

```
import unittest

class TestDivision(unittest.TestCase):
    def test_division_por_cero(self):
        with self.assertRaises(ZeroDivisionError):
            dividir_numeros(10, 0)

    def test_division_valida(self):
        self.assertEqual(dividir_numeros(10, 2), 5.0)

if __name__ == "__main__":
    unittest.main()
```

Evita Variables Globales

Las variables globales pueden introducir errores difíciles de rastrear. Prefiere pasar parámetros explícitamente.

Usa estructuras de datos adecuadas

Aprovecha las estructuras de datos propias de Python como listas, diccionarios o conjuntos para manejar datos eficazmente y evitar errores de lógica.

Usa Log en lugar de `print`

Usa el módulo `logging` para registrar errores y eventos, lo que facilita la depuración:

```
import logging

logging.basicConfig(level=logging.INFO, filename="app.log")

try:
    resultado = 10 / 0
except ZeroDivisionError as e:
    logging.error("Se intentó dividir por cero", exc_info=True)
```

Puedes definir fácilmente el nivel de un mensaje o filtrar los que se muestran o no y

Actualiza siempre Python y librerías

Mantén el intérprete y las bibliotecas que usas actualizados a las versiones estables más recientes para beneficiarte de correcciones de errores y mejoras en seguridad.

Fomenta la revisión de código y la programación en parejas

La colaboración mediante **revisiones de código** y programación en parejas ayuda a identificar posibles errores y mejorar la calidad general del código.

5. Diseño de Código Robusto

Principio de Menor Sorpresa

Escribe código predecible. Por ejemplo, una función que calcula un promedio no debería modificar los datos de entrada.

Manejo de Casos Extremos

Considera casos límite, como entradas vacías, valores nulos o tamaños de datos grandes:

```
def promedio(valores):
    if not valores:
        return 0.0  # Manejo de lista vacía
    return sum(valores) / len(valores)
```

Evita Código Duplicado

Usa funciones o clases para reutilizar código, reduciendo la probabilidad de errores:

```
def procesar_datos(datos):
    if not datos:
        return []
    return [d * 2 for d in datos]
```

Modularidad

Divide tu código en módulos y funciones pequeñas con responsabilidades claras. Esto facilita el manejo de errores y la depuración.

6. Herramientas para Mejorar la Robustez

- **Linters**: Usa herramientas como `flake8` o `pylint` para detectar errores de estilo y posibles bugs.
- **Type Checkers**: Usa `mypy` para verificar tipos estáticos.
- **Formatters**: Usa `black` o `isort` para mantener el código consistente.
- **Depuración**: Usa `pdb` o entornos como VS Code para depurar errores.

7. Ejemplo Completo: Programa Robusto

Aquí hay un ejemplo que combina varias prácticas para crear un programa robusto:

```
import logging
from typing import List, Union

logging.basicConfig(level=logging.INFO, filename="calculadora.log")

def dividir_lista(numeros: List[Union[int, float]], divisor: Union[int, float]) ->
List[float]:
    """
    Divide cada número de una lista por un divisor.

    Args:
        numeros: Lista de números a dividir.
        divisor: Número por el cual dividir.

    Returns:
        Lista con los resultados de las divisiones.

    Raises:
        ZeroDivisionError: Si el divisor es cero.
        TypeError: Si los argumentos no son válidos.
    """
    if not isinstance(divisor, (int, float)):
        raise TypeError("El divisor debe ser un número.")
    if divisor == 0:
        raise ZeroDivisionError("El divisor no puede ser cero.")
    if not all(isinstance(n, (int, float)) for n in numeros):
        raise TypeError("Todos los elementos de la lista deben ser números.")

    resultados = []
    for num in numeros:
        try:
            resultados.append(num / divisor)
        except Exception as e:
            logging.error(f"Error al dividir {num}: {e}", exc_info=True)
```

```
            resultados.append(None)
    return resultados

def main():
    try:
        datos = [10, 20, "30", 40]
        divisor = 2
        resultado = dividir_lista(datos, divisor)
        print(f"Resultados: {resultado}")
    except (TypeError, ZeroDivisionError) as e:
        print(f"Error en la ejecución: {e}")
        logging.error(f"Error en main: {e}", exc_info=True)

if __name__ == "__main__":
    main()
```

Conclusión

El manejo adecuado de errores y la programación robusta son esenciales para crear aplicaciones confiables en Python. Al usar bloques `try-except`, excepciones personalizadas, validaciones, pruebas unitarias y herramientas de análisis, puedes reducir significativamente los errores y mejorar la calidad del código. Adopta estas prácticas desde el inicio de tus proyectos para ahorrar tiempo y esfuerzo en la depuración y mantenimiento.