📝 Apuntes Mejorados: Control de Errores y `try...except` en Python

El manejo de excepciones es el mecanismo que permite a tu programa **responder elegantemente a errores** (situaciones inesperadas) sin colapsar, asegurando una experiencia de usuario más robusta y amigable.

1. El Bloque `try...except`: El Corazón del Control de Errores

La estructura `try...except` intenta ejecutar un código potencialmente problemático (`try`) y, si falla, permite "atrapar" y gestionar el error (`except`).

Estructura y Flujo de Control

1. `try` **:** Contiene el código sensible o que puede lanzar una excepción.
2. `except TipoDeError:` **:** Captura el error específico. Si ocurre en el `try` , se ejecuta el código del `except` y el programa continúa. Si no se especifica el tipo (`except:`), captura cualquier error, lo cual es útil para casos genéricos, pero **desaconsejado** por dificultar la depuración.
3. **Flujo Visual:**

Análisis Detallado del Código

El código utiliza el bloque `try...except` dentro de un bucle `while` para forzar al usuario a ingresar una entrada válida antes de salir.

Excepción (Bloque `except`)	Significado y Causa	Mensaje Personalizado
`ZeroDivisionError`	**Error de Aritmética.** Se lanza si el usuario introduce `0` .	`"⚠️ Has intentado devidir un numero por '0' ⚠️"`
`ValueError`	**Error de Tipo/Valor.** Se lanza si la entrada (`input`) no puede ser convertida a entero por `int()` (ej. si el usuario escribe `"hola"`).	`"⚠️ Error, debemos meter una letra o un valor que no sea un numero entero ⚠️"`
`except:` (Captura Genérica)	**Captura CUALQUIER otro error**, incluyendo el `AssertionError` lanzado si el número es negativo.	`"⚠️ Execepcion, pero otra ⚠️"`

2. Bloques Opcionales: `else` y `finally`

Estos bloques añaden control de flujo avanzado y capacidades de limpieza.

A. Bloque `else`

- **Condición de Ejecución:** Se ejecuta **SOLO si el bloque** `try` **finaliza con éxito**, es decir, sin lanzar ninguna excepción.
- **Utilidad:** Ideal para colocar código que *depende* del éxito del `try` .

```
else: #* Se ejecuta si 'try' funciona sin errores
    print(resultado)
    correcto = True
```

B. Bloque `finally`

- **Condición de Ejecución:** Se ejecuta **SIEMPRE**, haya o no habido excepciones, y ocurra o no un `else`.
- **Utilidad:** Fundamental para tareas de limpieza que deben ocurrir sí o sí (ej. cerrar un archivo, liberar memoria o, en este caso, indicar que el proceso continúa).

```
finally: #* Se ejecuta siempre
    print("Seguimos adelante...")
```

3. Errores Personalizados: `raise` y `assert`

Estas sentencias permiten al programador lanzar excepciones basadas en la lógica de negocio.

A. La Sentencia `assert` **(Afirmaciones Lógicas)**

Se usa para verificar que una condición debe ser verdadera en un punto específico del código.

- **Comportamiento:** Si la **condición es falsa**, lanza un `AssertionError`.
- **Sintaxis:** `assert condición, "Mensaje de error"`

```
assert numero>=0,"No admito numero negativos"
```

- En este caso, si `numero` es menor que `0`, se lanza un `AssertionError`. Este error es capturado por el bloque genérico `except:`.

B. La Sentencia `raise` **(Lanzar Excepciones Explícitas)**

Permite detener el programa de forma controlada y lanzar una excepción específica (existente o personalizada) con un mensaje claro.

```
# Código de ejemplo (comentado):
"""if numero < 0:
    raise Exception("No es un numero positivo")""" #! Lanza una excepción genérica
```

4. Consejos para Mejorar la Programación y el Uso de Excepciones (Examen Pro-Tip)

Para demostrar dominio, no solo captures errores, ¡prevenlos y úsalos con sabiduría!

🎯 Principio 1: No Captures Errores sin Sentido (EAFP vs. LBYL)

- **Malo (LBYL - Look Before You Leap):** Revisar si algo funcionará antes de intentarlo. Puede ser propenso a errores de concurrencia.
- **Bueno (EAFP - Easier to Ask for Forgiveness than Permission):** Intentar la acción y capturar el error si ocurre. **Esto es lo que hace tu código.** Es la filosofía de Python.

🎯 Principio 2: Sé Específico en el `except`

- **MALO:** `except: ...` (Captura todo, incluyendo `SystemExit` o errores de programación, ocultando fallos graves).
- **BUENO:** `except (ZeroDivisionError, ValueError): ...` (Solo captura los errores que sabes cómo manejar).

🎯 Principio 3: Usar `else` para la Claridad

- No pongas código que se ejecuta correctamente en el `try`. El `try` debe ser lo más corto posible y contener solo la acción de alto riesgo.
- El bloque `else` es el lugar perfecto para el código que *depende* del éxito del `try` (como en tu código: `print(resultado)`).

🎯 Principio 4: Crea tus Propias Clases de Excepción

Para programas grandes, define tus propias excepciones (ej. `class NumeroNegativoError(Exception): pass`) que hereden de la clase base `Exception`. Esto hace que tu código sea más semántico y permite a otros módulos capturar solo los errores específicos de tu código.