

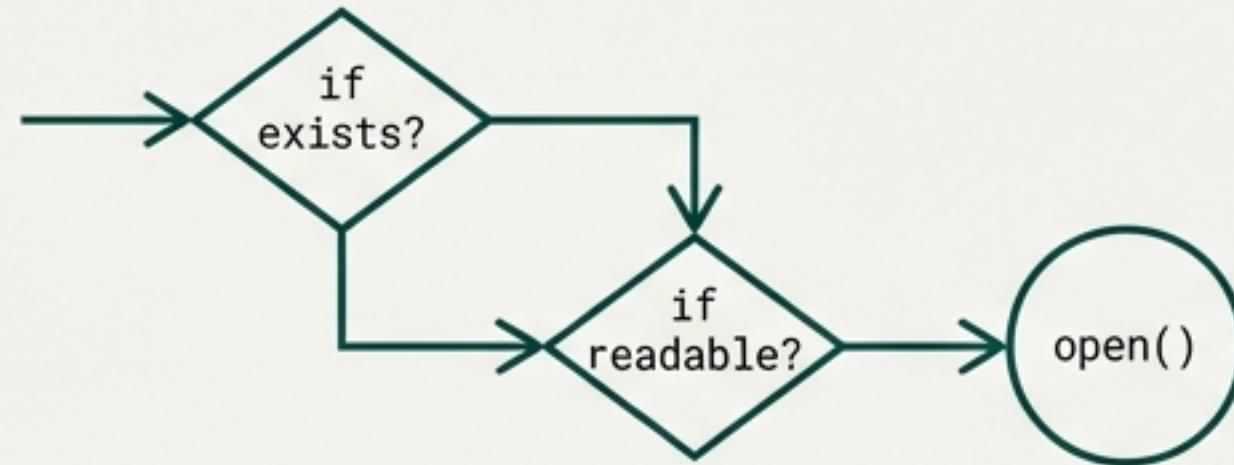


Blandskron

De Reaccionar a Diseñar: Dominando el Control de Flujo con Excepciones en Python

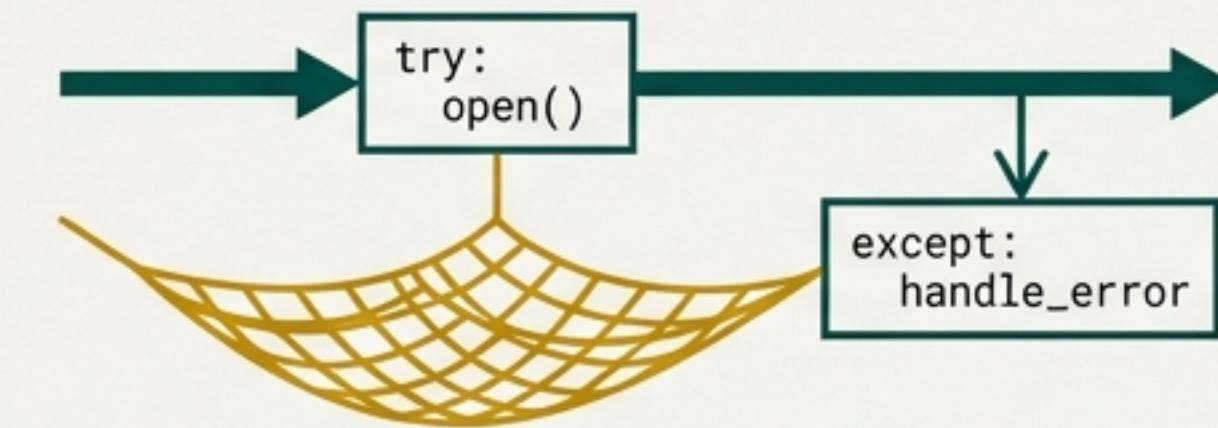
Una guía para escribir código robusto, resiliente y Pythónico.

La Filosofía Pythonica: Es Más Fácil Pedir Perdón que Permiso (EAFP)



LBYL (Look Before You Leap)

- 'Mira antes de saltar'.
- Común en C o Java.
- Dicta que se deben verificar todas las precondiciones antes de una operación (ej. `if os.path.exists(file)`).
- **Problema clave:** Vulnerable a condiciones de carrera en entornos concurrentes. El estado puede cambiar entre la verificación y la acción.



EAFP (Easier to Ask for Forgiveness than Permission)

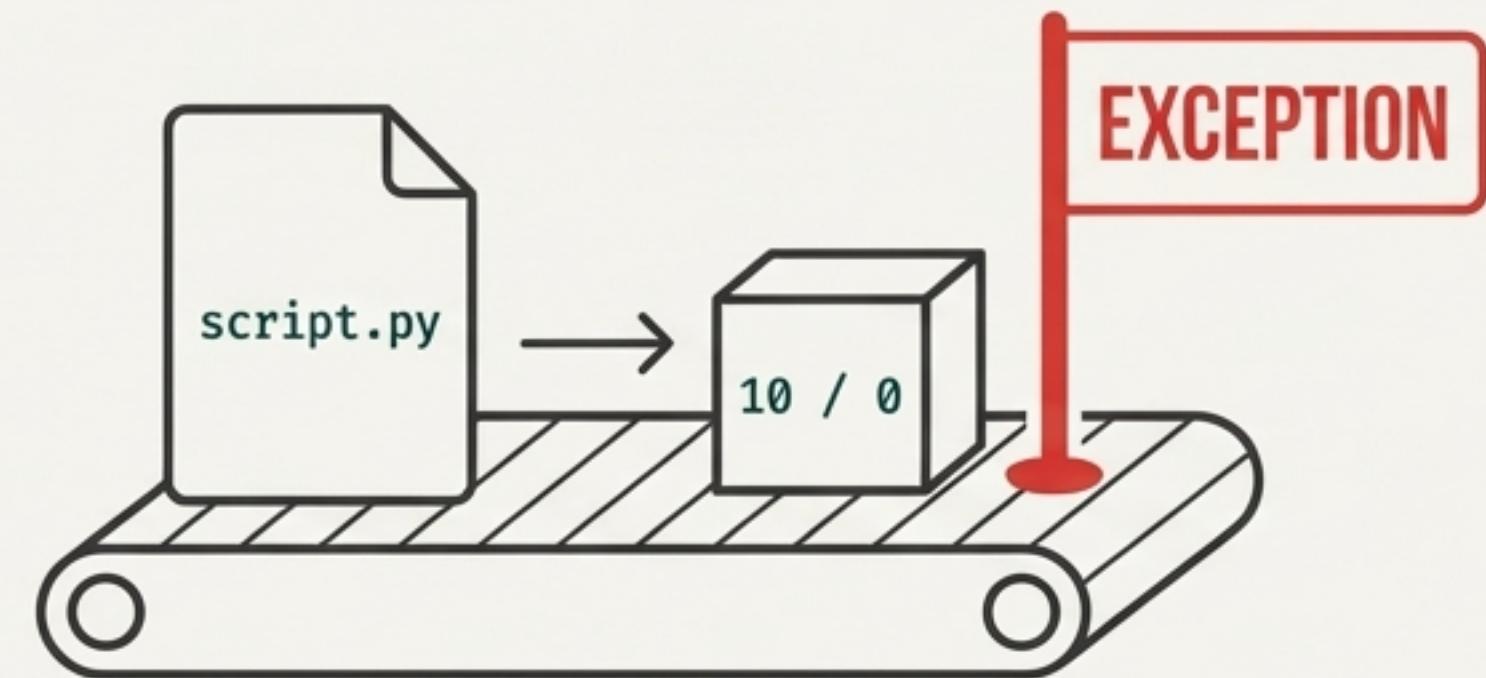
- El enfoque preferido en Python.
- Se asume que las operaciones son válidas y se procede directamente.
- Si la operación falla, se lanza una excepción que es capturada y manejada.
- **Ventajas:** Código más limpio, legible y atómico, evitando condiciones de carrera. Se enfoca en el 'camino feliz'.

"El dominio del manejo de excepciones en Python es, en esencia, el dominio del estilo EAFP."

La Anatomía de un Fallo: Errores de Sintaxis vs. Excepciones



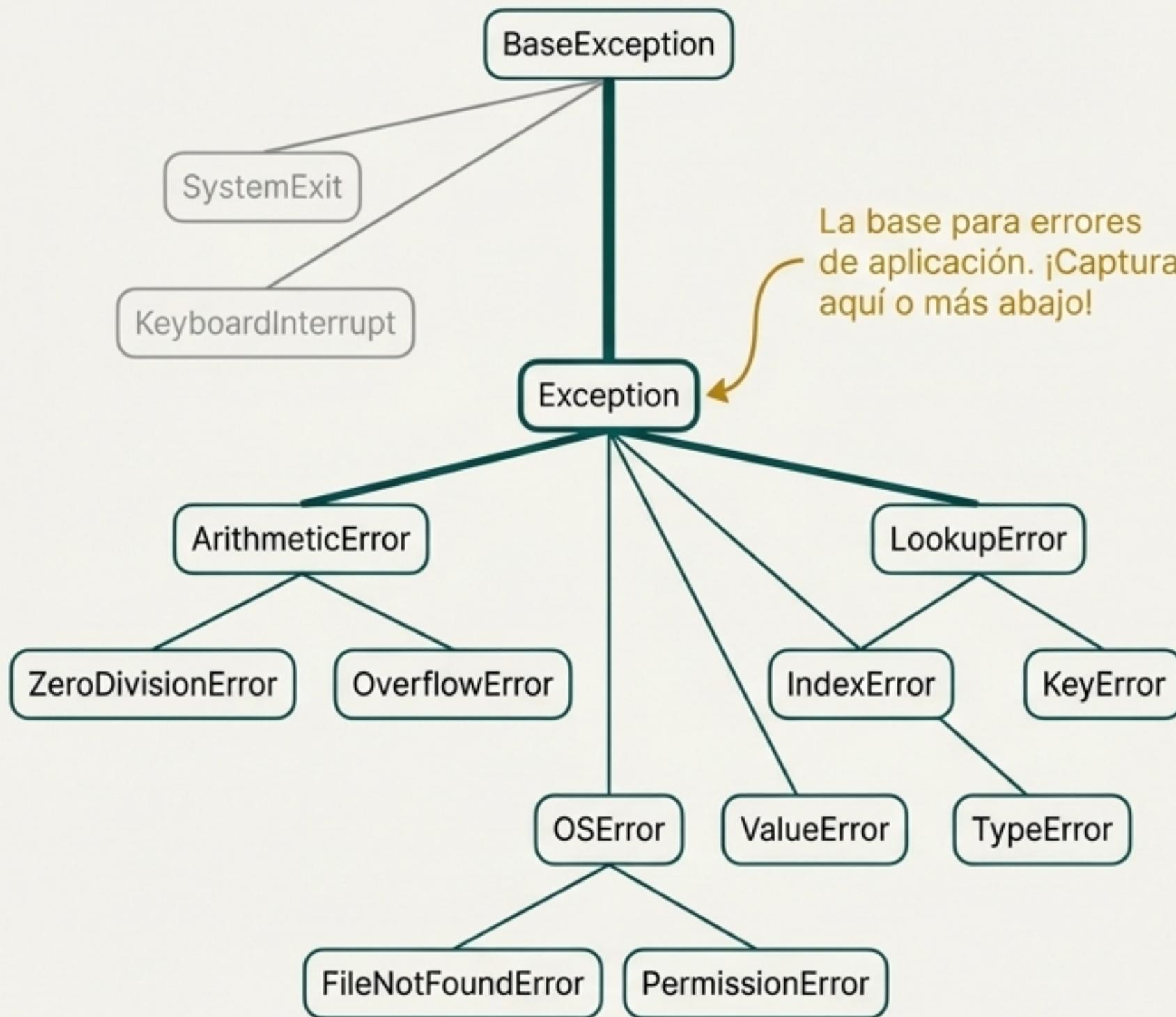
El programa nunca arranca.



El programa se detiene durante la ejecución.

Característica	Error de Sintaxis (SyntaxError)	Excepción (Exception)
Momento de Detección	Tiempo de análisis (antes de la ejecución). El programa nunca arranca.	Tiempo de ejecución (durante la ejecución).
Causa	Violación de las reglas gramaticales del lenguaje (ej. olvidar `::`, paréntesis desequilibrados).	Una operación sintácticamente válida falla (ej. `10 / 0`, `int("abc")`).
Manejo	Imposible de manejar con `try-except`. Requiere corrección del código fuente.	Diseñado para ser manejado con bloques `try-except`.
Ejemplos	<code>SyntaxError</code> , <code>IndentationError</code> , <code>TabError</code> .	<code>ZeroDivisionError</code> , <code>TypeError</code> , <code>ValueError</code> , <code>FileNotFoundException</code> .

No Todos los Errores son Iguales: La Jerarquía de Excepciones



- La cúspide de la pirámide. No se debe capturar directamente en código de aplicación, ya que incluye señales de salida como `'SystemExit'` y `'KeyboardInterrupt'`.
- **Esta es la que debes usar como base para tus propias excepciones y para capturas genéricas (con precaución).**

Grupos Funcionales Comunes:

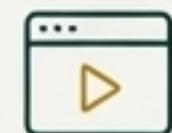
- `ArithmeticError`: `ZeroDivisionError`, `OverflowError`.
- `LookupError`: `IndexError` (secuencias), `KeyError` (diccionarios).
- `OSSError`: `FileNotFoundException`, `PermissionError`.

Una distinción clave:

- `ValueError` vs. `TypeError`: "ValueError para tipo correcto pero valor inapropiado (ej. `int('abc')`). `TypeError` para tipo incorrecto (ej. `len(123)`)".

La Red de Seguridad: Implementando `try` y `except`

Flujo de Ejecución Explicado



1. El código dentro del bloque `try` se ejecuta.

✓ → 2. **Sin excepción:** El bloque `except` se ignora. La ejecución continúa después.



3. **Con excepción:** La ejecución del `try` se detiene. El intérprete busca un bloque `except` que coincida. Si lo encuentra, ejecuta ese bloque y continúa. Si no, el programa termina.

Ejemplo de Código 1: Captura Específica

```
# Anotación: Protegiendo una operación propensa a fallos
try:
    valor = int(input("Ingrese un entero: "))
    print(f"Número válido ingresado: {valor}")
# Anotación: Manejando un tipo de error específico
except ValueError as e:
    print(f"Entrada no válida. Error: {e}")
```

Ejemplo de Código 2: Captura Múltiple

```
# Anotación: Agrupando errores con una respuesta similar
try:
    # Este código está diseñado para fallar
    x = int("abc") / 0
except (ValueError, TypeError) as e:
    print(f"Error de tipo o valor: {e}")
```

La Estructura Completa: `else` para el Éxito, `finally` para la Garantía



La Cláusula `else`: El Camino del Éxito

Se ejecuta **únicamente** si el bloque `try` se completa sin lanzar ninguna excepción.

¿Por qué usarla? Para minimizar el código dentro del `try`. Protege solo la línea que puede fallar y coloca el código de procesamiento exitoso en el `else`, evitando capturar accidentalmente errores de lógica posteriores.

```
try:  
    f = open('archivo.txt', 'r')  
except OSError:  
    print('No se pudo abrir el archivo')  
else:  
    # Anotación: Este código solo se ejecuta si open() fue exitoso  
    contenido = f.read()  
    print("Archivo leído con éxito.")  
    f.close()
```



La Cláusula `finally`: Garantía de Ejecución

Se ejecuta **siempre**, sin importar si hubo una excepción, si fue manejada o no, o incluso si hay un `return` en el `try` o `except`.

Uso Canónico: Liberar recursos externos (cerrar archivos, conexiones de base de datos, liberar locks).

Advertencia Crítica: Evitar usar `return` dentro de un bloque `finally`, ya que puede suprimir excepciones activas silenciosamente.

La Máxima Elegancia: Gestión de Recursos con `with` y Gestores de Contexto

La sentencia `with` automatiza la gestión de recursos que necesitan acciones de configuración y limpieza. Utiliza el protocolo de "Gestor de Contexto" (`__enter__` y `__exit__`), garantizando la limpieza incluso si ocurren excepciones.

Antes (con `try/finally`)

```
f = open('datos.txt', 'r')
try:
    datos = f.read()
    # ... procesar datos
finally:
    ↳ f.close() # Anotación: Cierre manual y propenso
                a olvidos
```

Ahora (con `with`)

```
# Anotación: Limpio, seguro y autodocumentado
with open('datos.txt', 'r') as f:
    datos = f.read()
    # ... procesar datos
# Anotación: f.close() se llama automáticamente aquí
```

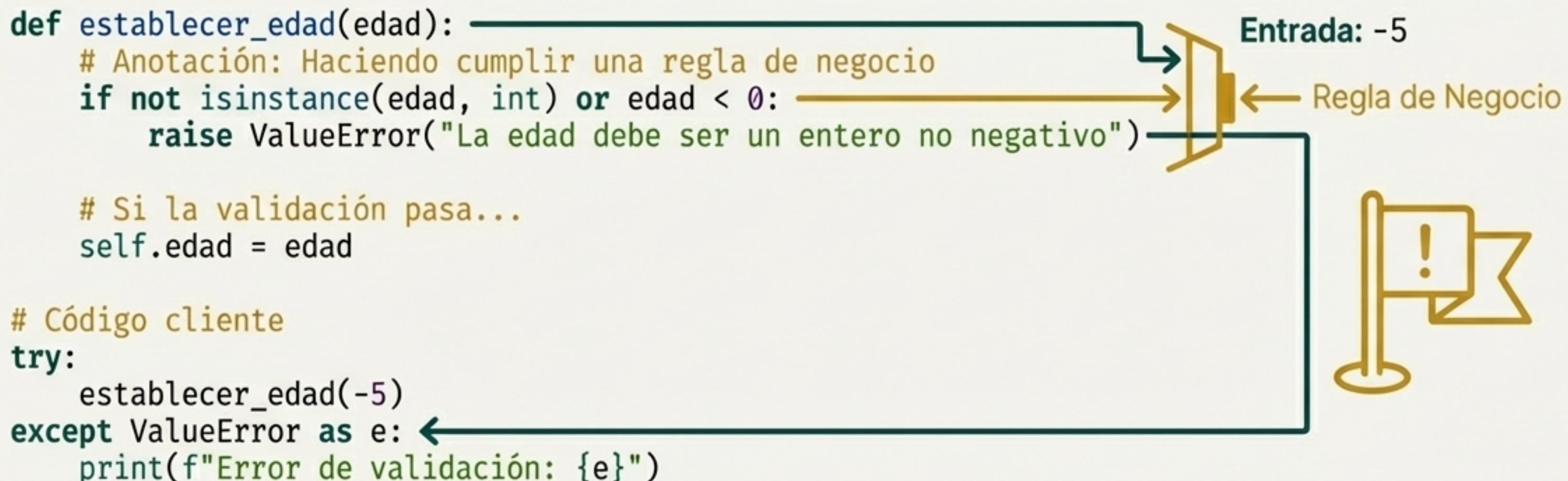
Prefiera siempre `with` sobre `try/finally` para la gestión de recursos estándar.

Tomando el Control: Señalización Proactiva con `raise`

La sentencia `raise` permite al programador forzar la ocurrencia de una excepción. Es fundamental para validar argumentos de una función, señalar estados inválidos en la lógica de negocio y detener la propagación de datos incorrectos en su origen.

```
def establecer_edad(edad): —————
    # Anotación: Haciendo cumplir una regla de negocio
    if not isinstance(edad, int) or edad < 0: —————→ Entrada: -5
        raise ValueError("La edad debe ser un entero no negativo") ← Regla de Negocio
    # Si la validación pasa...
    self.edad = edad

    # Código cliente
try:
    establecer_edad(-5)
except ValueError as e: ←
    print(f"Error de validación: {e}")
```



The diagram shows a flow from an input value '-5' through a 'Business Rule' component to an exception being raised, and finally being caught and handled in the client code.

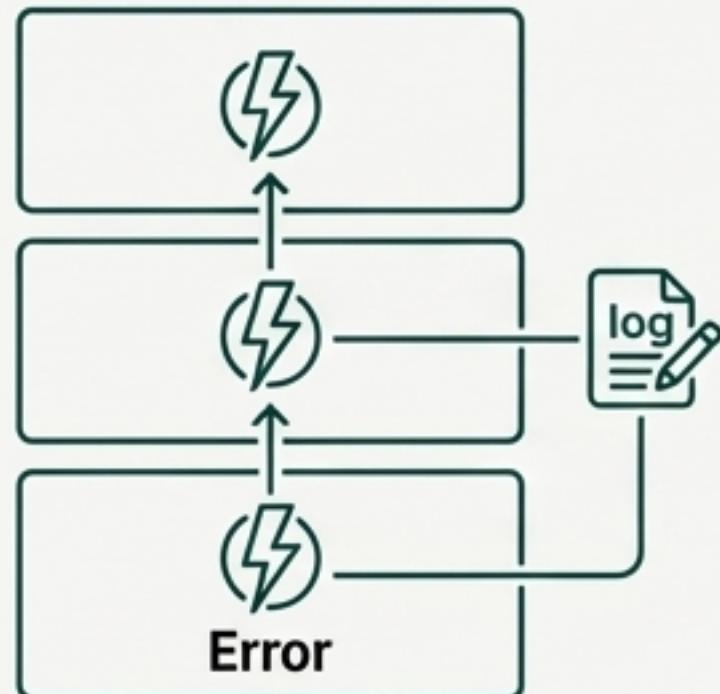


“Al ejecutar `raise`, la función termina inmediatamente. No devuelve ningún valor. El control pasa al primer manejador compatible en la pila de llamadas.”

Técnicas Avanzadas de Señalización: Relanzar vs. Encadenar

Relanzamiento (Re-raising)

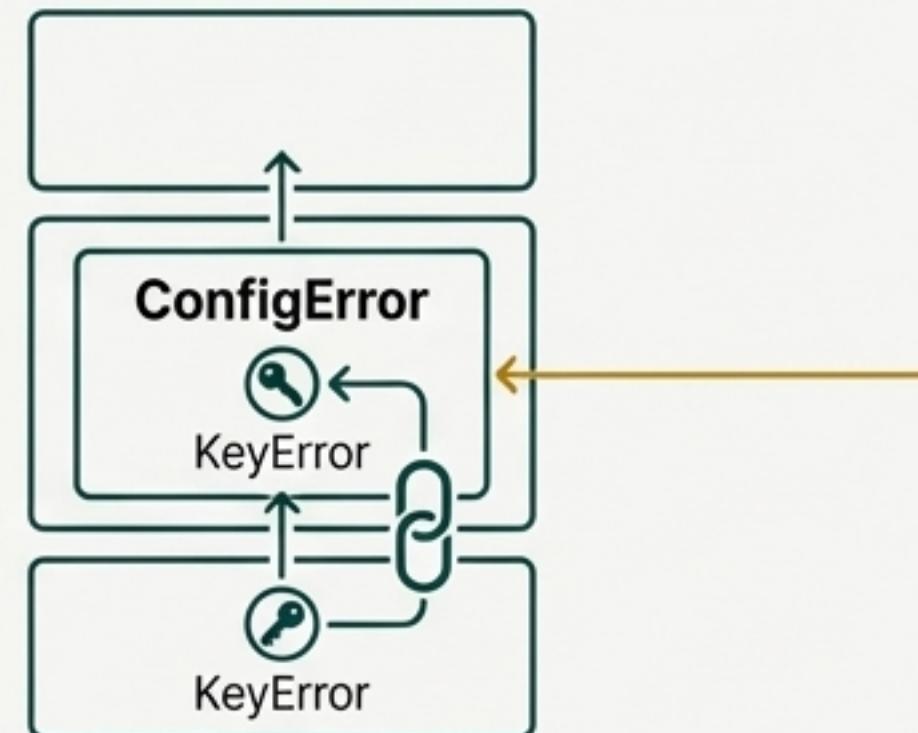
Cuando necesitas realizar una acción (ej. logging) pero **no puedes resolver el error**, por lo que debe continuar su propagación.



```
try:  
    procesar_pago()  
except PaymentError as e:  
    logger.error(f"Falló de pago: {e}")  
    raise # Anotación: Relanza la misma excepción,  
          preservando el traceback original
```

Encadenamiento Explícito ('from')

Para **envolver** una excepción de **bajo nivel** en una de alto nivel, más semántica, **sin perder la causa raíz**.



```
try:  
    val = config_dict[clave]  
except KeyError as e:  
    raise ConfigError("Clave de config faltante") from e
```

Creando tu Propio Vocabulario: Excepciones Personalizadas

¿Por qué crearlas?

- Los errores integrados (`ValueError`) describen fallos de programación, no de lógica de negocio.
- Las excepciones personalizadas (ej. `'SaldoInsuficienteError'`) hacen el código autodocumentado y semánticamente más rico.

¿Cómo diseñarlas?

1. Crea una clase base para tu aplicación que herede de `'Exception'`.
2. Crea excepciones específicas que hereden de tu clase base.
3. Sobrescribe `'__init__'` para que tus excepciones puedan transportar datos estructurados.



Ejemplo de Código (Sistema Bancario):

```
class ErrorBanco(Exception):
    """Clase base para errores del módulo."""
    pass

class FondosInsuficientesError(ErrorBanco):
    """Lanzada cuando el retiro excede el saldo."""
    # Anotación: La excepción ahora lleva datos útiles
    def __init__(self, necesario, disponible):
        self.necesario = necesario
        self.disponible = disponible
        msg = f"Intento de retirar {necesario}, pero solo hay {disponible}"
        super().__init__(msg)

# En el código cliente:
try:
    # ... lógica de retiro ...
    pass
except FondosInsuficientesError as e:
    print(f"Faltan {e.necesario - e.disponible} para completar la operación.")
```

A yellow arrow points from the annotation "# Anotación: La excepción ahora lleva datos útiles" to the code where the `__init__` method is defined, specifically to the line where `necesario` and `disponible` are assigned. Another yellow arrow points from the line `print(f"Faltan {e.necesario - e.disponible} para completar la operación.")` back up to the annotation.

Caso de Estudio 1: Un Procesador de Archivos a Prueba de Fallos

Escenario – Un script que lee un archivo de configuración línea por línea, convierte cada una a número y realiza una división. Debe manejar los archivos inexistentes, datos corruptos y división por cero sin detenerse.

```
def procesar_configuracion(ruta_archivo):
    try: ←
        with open(ruta_archivo, 'r') as f:
            for linea in f:
                try: ←
                    valor = int(linea.strip())
                    resultado = 1000 / valor
                    print(f'Línea procesada: {resultado:.2f}')
                except ValueError:
                    print(f'Error: línea '{linea.strip()}' no es un número válido.')
                except ZeroDivisionError:
                    print(f'Error: línea '{linea.strip()}' causa división por cero.')
            except FileNotFoundError:
                print(f'Error Fatal: El archivo '{ruta_archivo}' no existe.')
            except PermissionError:
                print(f'Error Fatal: Sin permisos para leer '{ruta_archivo}'.')
        finally:
            print(f'--- Fin del procesamiento de {ruta_archivo} ---')

    Bloque externo para errores fatales del archivo
    Bloque interno para errores recuperables por línea
```

Análisis: Este código demuestra la anidación de bloques `try`. El interno maneja errores de datos (recuperables), permitiendo que el ciclo continúe. El externo maneja errores de infraestructura (fatales para ese archivo), deteniendo el procesamiento de ese archivo pero no necesariamente de todo el programa.

Caso de Estudio 2: Lógica de Negocio con Excepciones Bancarias

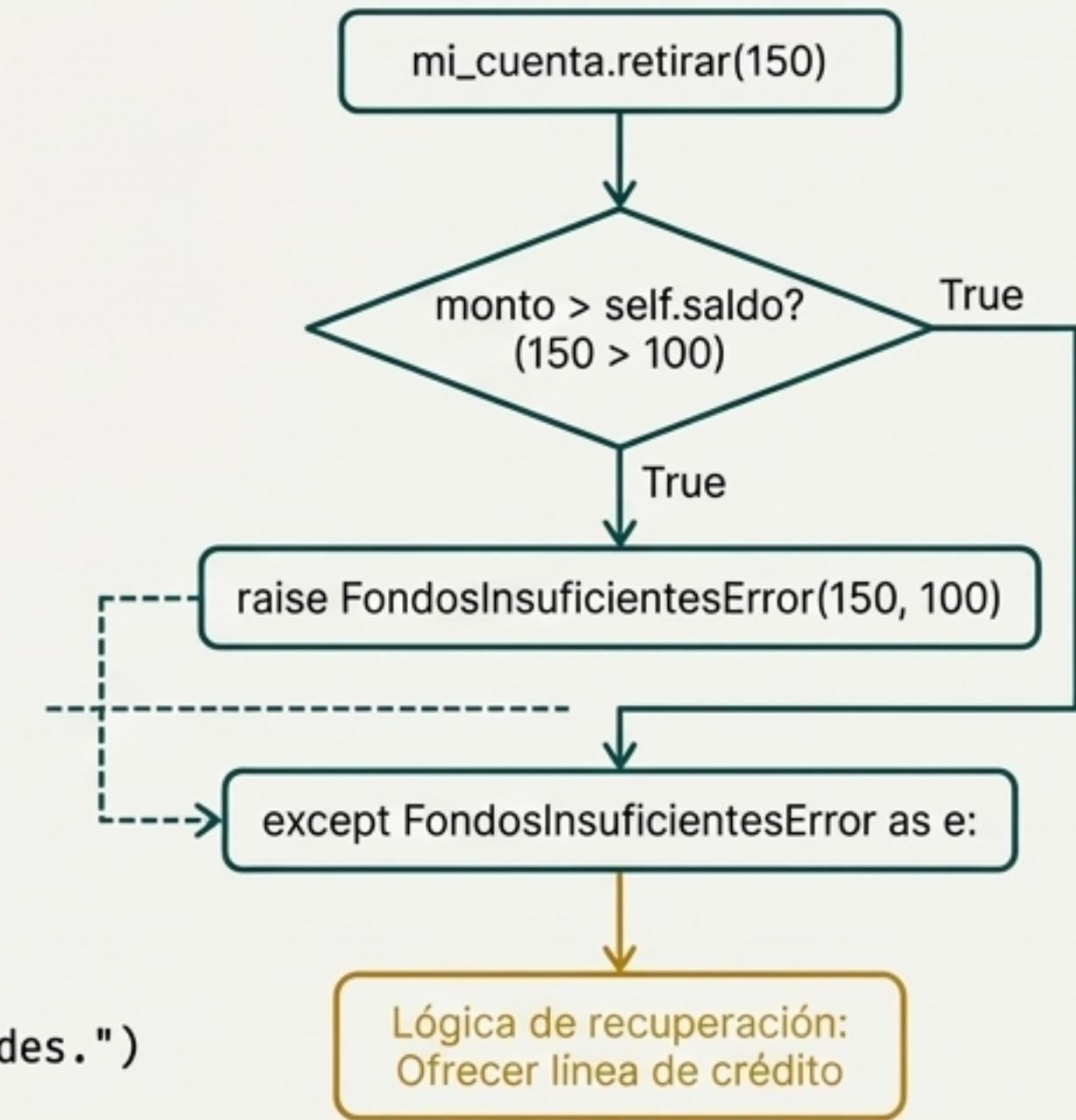
Escenario: Una clase `CuentaBancaria` que usa excepciones personalizadas para hacer cumplir reglas de negocio.

Código de la Lógica de Negocio

```
class CuentaBancaria:  
    # ... __init__ ...  
    def retirar(self, monto):  
        if monto <= 0:  
            raise MontoInvalidoError("El retiro debe ser positivo.")  
        if monto > self.saldo:  
            # Anotación: Lanzando una excepción rica en datos  
            raise FondosInsuficientesError(monto, self.saldo)  
        self.saldo -= monto
```

Código del Control de Flujo Cliente

```
mi_cuenta = CuentaBancaria("Juan Perez", 100)  
try:  
    mi_cuenta.retirar(150)  
except MontoInvalidoError as e:  
    print(f"Operación Rechazada: {e}")  
except FondosInsuficientesError as e:  
    # Anotación: El flujo de control puede tomar decisiones inteligentes  
    print(f"Fondos Insuficientes: Faltan {e.necesario - e.disponible} unidades.")  
    # Lógica potencial: ofrecer_linea_de_credito(...)
```



Análisis

En lugar de devolver códigos de error como `False` o `-1`, los métodos lanzan excepciones semánticas. El código cliente las captura para decidir el siguiente paso, demostrando un control de flujo rico que va más allá de la simple recuperación de errores.

El Camino a la Maestría: Resumen de Mejores Prácticas

- 1.**  **Sé Específico:** Captura siempre la excepción más específica posible. Evita `except Exception:` o `except:` desnudos, ya que ocultan bugs.
- 2.**  **Mantén la Atomicidad:** Los bloques try deben ser cortos. Protege solo la línea o el grupo mínimo de líneas que pueden fallar.
- 3.**  **Usa la Semántica:** Emplea excepciones personalizadas para errores lógica de negocio. Esto hace que el código sea autodocumentado.
- 4.**  **Preserva el Contexto:** Al relanzar o envolver excepciones, usa `raise (solo)` o `raise from` para mantener la cadena causal y facilitar la depuración.
- 5.**  **Limpia con Elegancia:** Prefiere `with` y gestores de contexto sobre `try/finally` para la gestión de recursos estándar.

El dominio de estas técnicas transforma el manejo de errores de una tarea defensiva a una estrategia integral de control de flujo, resultando en software de calidad profesional capaz de operar en el imprevisible mundo real.