Manejo de Errores

Validación de datos

Al trabajar con datos, es inevitable que se produzcan errores con su uso y manipulación. La detección y corrección de errores (depuración) es una parte muy importante en el proceso de desarrollo de software.

El ingreso de datos es uno de los puntos con mayor frecuencia de errores, y a su vez, es el lugar más "económico" para detectar y salvar errores. Siempre que exista un ingreso de datos, es posible que el usuario se equivoque, y debemos asumir que se equivocará.

El proceso mediante el cual se verifican los datos ingresados, y eventualmente, se rechazan, es conocido como **validación** de datos o validación de ingreso. Debemos validar los datos siempre que se pueda, y siempre que sea conveniente.

Por ejemplo, si se pide el ingreso de un número múltiplo de tres, deberíamos validar que efectivamente sea múltiplo de 3. O si se pide la edad de una persona, validar que sea un número entero y positivo.

Cuando detectamos un error de validación, lo ideal es informar de ello al usuario y darle otra oportunidad para que ingrese el dato, sin avanzar en el programa, hasta que se ingrese un dato válido.

Al realizar conversiones de datos (*casteos*), se producen **excepciones** si los datos no son compatibles. Por ejemplo, si se quiere hacer `int(entrada)` para convertir un string a entero, pero la variable `entrada` contiene una letra, lanzará una excepción.

Para evitar esto, antes de realizar la conversión se puede verificar si el string contiene lo que esperamos. Para esto existen métodos como `.isdigit()` y `.isnumeric()`.

Excepciones

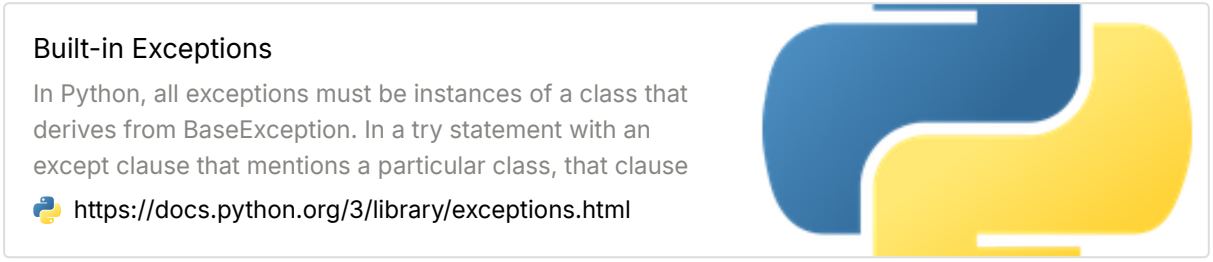
Una excepción es un error en la ejecución del programa, que interrumpe el flujo normal del mismo. Las excepciones son objetos, que contienen información: el tipo de error, el mensaje de error, y la línea del código donde se produjo.

Un ejemplo típico de excepción es `NameError`, que ocurre si intentamos acceder a una variable que no está definida.

Todas las excepciones extienden la clase `BaseException`. Algunas excepciones comunes son:

- **AssertionError**: fallo de assert
- **AttributeError**: falla la referencia de atributos (al intentar acceder a un atributo inexistente de un objeto)
- **EOFError**: ocurre al terminar de leer un archivo e intentar seguir leyendo del mismo
- **FloatingPointError**: error en una operación de punto flotante
- **ImportError**: se produce cuando se intenta importar un módulo que no se encuentra
- **IndexError**: se produce cuando se intenta acceder a un índice fuera del rango de una lista
- **KeyError**: se produce cuando se intenta acceder a una key inexistente de un diccionario
- **MemoryError**: ocurre cuando el programa se queda sin memoria
- **NameError**: se produce cuando se intenta acceder a una variable no definida
- **OSError**: error del sistema operativo. Por ejemplo, cuando se intenta leer un archivo que no existe.
- **ZeroDivisionError**: ocurre cuando se intenta dividir por cero.

La lista de todas las excepciones predeterminadas de Python junto con su significado, se puede encontrar en la documentación:



Manejo de excepciones con `try` / `except`

Mediante el uso de las sentencias `try` y `except`, podemos hacer que nuestro programa sea más robusto y resistente a errores.

```
try:
    # Código que podría producir una excepción
except:
    # Este bloque se ejecuta cuando se produce una excepción
```

Python ejecuta el código dentro del bloque `try`, y si no ocurre ningún error, el programa prosigue con normalidad. En caso de que se produzca un error, se ejecutará el bloque `except`, donde podemos realizar acciones en respuesta a dicho error.

Ejemplo de uso en un programa que pide al usuario un número entero:

```
numero = None
while type(numero) is not int:
    try:
        entrada = input('Ingresa un número entero: ')
        numero = int(entrada) # Esto genera un ValueError si
        print(f"Tu número entero es: {numero}")
    except ValueError:
        print('El valor ingresado no es válido.')
```

En este caso, si el valor ingresado no se puede convertir a número entero (por ejemplo, si es un número decimal o contiene letras), se lanzará un `ValueError`, y en el bloque `except` respondemos a dicho error informándole al usuario que el valor ingresado no es válido. Mediante el bucle `while` de la segunda línea, volvemos a pedir el dato al usuario hasta que el número sea un entero válido.

Después de la palabra clave `except` podemos especificar el nombre del error que queremos capturar, o también podemos capturar múltiples tipos de error, separándolos por coma. Si no se especifica qué error(es), se capturará cualquier error que ocurra.

En ocasiones, puede ser útil ignorar ciertos tipos de excepciones. Esto puede ser necesario cuando sabemos que una operación puede generar un error,

pero no queremos que ese error detenga la ejecución del programa. En estos casos, Python nos proporciona la palabra clave `pass`.

Por ejemplo:

```
try:
    # Código que puede generar un error
except:
    pass # No hacer nada
```

Si se genera una excepción mientras se ejecuta este bloque de código, el programa no se detendrá. En cambio, se ejecutará el bloque `except`, que en este caso no hace nada.

De esta forma, se ignora la excepción y el programa continúa su ejecución. Esto difiere del comportamiento predeterminado de Python, donde si se genera una excepción que no se captura, el programa muestra un mensaje de error y se detiene.

Cláusulas `else` y `finally`

Adicional a `except`, podemos tener dos bloques más en el código:

- `else`: se ejecuta solo si el código del bloque `try` terminó sin ningún error (es decir, lo opuesto a `except`)
- `finally`: se ejecuta siempre, independientemente si el código del bloque `try` tuvo un error o no. Este bloque es útil para realizar operaciones de "limpieza", por ejemplo, si abrimos un archivo, vamos a querer cerrarlo, indistintamente de si hubo algún error al procesarlo o no.

```
try:
    # Código que puede fallar
except:
    # Responder al error (informar al usuario, reintentar, et
else:
    # No hubo ningún error
finally:
    # Limpieza - cerrar archivos, etc.
```

Lanzar excepciones con `raise`

Podemos usar la sentencia `raise` para lanzar nuestras propias excepciones en el código. Por ejemplo:

```
if x > 5:
    raise Exception(f"El valor no debe ser mayor a 5. El valo
```

Este código lanzará una excepción si el número en la variable `x` es mayor a 5.

La sentencia `raise` es particularmente útil dentro de una función, para pasarle el error a la función que la invoca, y manejarlo desde ahí. Por ejemplo:

```
def dividir(dividendo, divisor):
    if divisor == 0:
        raise Exception("No se puede dividir por cero")
    return dividendo / divisor
```

En este caso, el manejo de la excepción será responsabilidad del código que llame a la función `dividir`.

Ejemplo con `try` / `except` / `else` / `finally`:

```
# Programa que obtiene el cociente y el resto entre dos númer
dividendo = None
divisor = None

while type(dividendo) is not int or type(divisor) is not int:
  try:
    dividendo = int(input('Ingresa el dividendo: '))
    divisor = int(input('Ingresa el divisor: '))
  except ValueError:
    print('El valor ingresado no es válido.')

try:
  cociente = dividendo // divisor
  resto = dividendo % divisor
except ZeroDivisionError:
  print('No se puede dividir por cero.')
else:
  print(f'{dividendo} dividido por {divisor} es igual a {coci
```

```
finally:
  print('¡Hasta luego!')
```

- Si se ingresa en el dividendo o el divisor algo que no sea un número entero, se mostrará el mensaje “El valor ingresado no es válido” y se volverán a pedir los datos.
- Si se ingresan, por ejemplo, los números 73 y 4, la salida será:

```
73 dividido por 4 es igual a 18 y el resto es 1.
¡Hasta luego!
```

- Si se ingresan 10 y 0, la salida será:

```
No se puede dividir por cero.
¡Hasta luego!
```