**Módulo de 11**

**Manejo de excepciones:**

* try
* except
* else
* finally
* raiser

|  |  |
| --- | --- |
| En Python, el manejo de excepciones se realiza | |
| utilizando bloques   * + try   + except   + else   + finally   y la función   * + raiser | El bloque try se utiliza para encapsular el código que puede generar una excepción,  El bloque except se utiliza para manejar las excepciones capturadas.  El bloque else se utiliza tras un evento que no ingresa por excepción.  El bloque finally se utiliza como último bloque. Esto es muy importante en una función con return |

**Manejo de excepciones**

En un principio se trabajaba con manejo de errores. Estos fue mejorado hasta que no tengan que ser solo errores, sino que cualquier tipo de excepción o eventos que ocurren durante la ejecución y que interrumpen el flujo normal del programa.

El manejo de excepciones es un aspecto fundamental en la programación moderna, ya que permite que un programa responda adecuadamente a condiciones anómalas o errores sin detenerse abruptamente. Esto es crucial en aplicaciones robustas y seguras, ya que garantiza que, incluso ante problemas inesperados, el sistema pueda responder de manera controlada.

**Manejo de Excepciones en Python**

En Python, el manejo de excepciones se organiza mediante bloques específicos de código: try, except, else, finally, y la declaración raise. Estos bloques permiten gestionar diferentes situaciones de error, manejar los flujos alternativos de control, y garantizar que ciertos bloques de código se ejecuten, incluso en presencia de errores.

1. **Bloque** try

El bloque try es donde se coloca el código que puede generar una excepción. Python "intenta" ejecutar el código en este bloque. Si el código dentro del try ejecuta correctamente, el flujo continúa de forma normal. Sin embargo, si ocurre un error, se genera una excepción, que será "capturada" y manejada en un bloque except.

|  |
| --- |
| try:  resultado = 10 / 0 # Esto genera una excepción de división por cero |

* **¿Por qué se usa** try**?**: Este bloque se usa para envolver operaciones que son propensas a fallar. Por ejemplo, operaciones de entrada/salida (lectura de archivos), cálculos matemáticos que podrían involucrar valores problemáticos (como la división entre cero), o acceso a recursos externos (como bases de datos o redes).

2. **Bloque** except

El bloque except se ejecuta solo si una excepción ocurre en el bloque try. Aquí se especifica qué tipo de excepción se está esperando capturar. Si se omite el tipo de excepción, Python capturará cualquier excepción que se haya producido.

|  |
| --- |
| try:  resultado = 10 / 0 #<------------------------------- Dividimos sobre cero  except Exception as Error:  print(f"Error encontrado: {Error}") |

·

|  |
| --- |
| Error encontrado: division by zero |

·

|  |
| --- |
| try:  resultado = 2 \* x #<------------------------------- o x no existe  except Exception as Error:  print(f"Error encontrado: {Error}") |

·

|  |
| --- |
| Error encontrado: name 'x' is not defined |

·

* **Captura de múltiples excepciones**: Se pueden tener múltiples bloques except para manejar distintos tipos de excepciones:

|  |
| --- |
| try:  valor = int(input("Ingrese un número: "))  resultado = 10 / valor  except ValueError:  print("Error: Ingrese un número válido")  except ZeroDivisionError:  print("Error: División entre cero") |

* **Uso general de** except Exception as: También es común capturar cualquier excepción genérica para manejar casos imprevistos:

|  |
| --- |
| except Exception as error:  print(f"Se produjo un error: {error}") |

Este patrón es muy útil en sistemas críticos donde no se puede permitir que una excepción sin manejar detenga el flujo del programa.

3. **Bloque** else

El bloque else es opcional y se ejecuta solo si **no** ocurre ninguna excepción en el bloque try. Es útil para código que debe ejecutarse únicamente si todo lo anterior se ha completado correctamente.

|  |
| --- |
| try:  resultado = 10 / 2  except ZeroDivisionError:  print("Error: División entre cero")  else:  print(f"Resultado: {resultado}") # Este bloque solo se ejecuta si no hay error |

* **Uso de** else: El bloque else garantiza que el programador tenga un espacio separado para ejecutar el código cuando no ha ocurrido ningún error, manteniendo una clara distinción entre el código que maneja excepciones y el código que sigue el flujo normal.

4. **Bloque** finally

El bloque finally es un bloque que **siempre** se ejecuta, independientemente de si ocurrió o no una excepción. Se utiliza típicamente para limpiar recursos o realizar tareas que deben completarse, como cerrar archivos o conexiones de bases de datos, asegurando que ciertas acciones se realicen sin importar el resultado.

|  |
| --- |
| try:  resultado = 10 / 2  finally:  print("Este mensaje siempre se mostrará, haya o no un error.") |

.

|  |
| --- |
| def mi\_funcion\_dividir(valor\_1,valor\_2):  try:  return valor\_1 / valor\_2  except Exception as Error:  return f"Error encontrado: {Error}"  finally:  return "Estes es el retorno final"  print (f"{mi\_funcion\_dividir(10,2)}")  print (f"{mi\_funcion\_dividir(10,0)}") |

mi\_funcion\_dividir(10,2)

|  |
| --- |
| Estes es el retorno final |

mi\_funcion\_dividir(10,0)

|  |
| --- |
| Estes es el retorno final |

No importa que return se ejecute primero. Siempre el que vale es el de finally·

* **Importancia de** finally: El bloque finally es crucial en programas donde se manejan recursos externos, como conexiones a bases de datos o archivos abiertos, que deben cerrarse sin importar si se produjo una excepción o no.

En el siguiente ejemplo, incluso si se intenta dividir por cero (generando una excepción), el bloque finally se ejecuta, cerrando el archivo.

|  |
| --- |
| try:  file = open('archivo.txt', 'r')  resultado = 10 / 0  except ZeroDivisionError:  print("Error: División entre cero")  finally:  file.close() # Este archivo se cierra aunque haya ocurrido un error  print("Archivo cerrado.") |

|  |
| --- |
| try:  ingreso = int(input("Ingrese un número: "))  resultado = 10 / ingreso  print (f"resultado = {resultado}")  except ValueError:  print("Error: Ingrese un número válido")#< ---------------------- el usuario no ingreso un entero  except ZeroDivisionError:  print("Error: División entre cero")#< --------------------------- el usuario ingreso 0 cero  else:  print("¡No se produjo ninguna excepción!")  finally:  print ("the end..... :)") |

·

|  |
| --- |
| Ingrese un número: 5#<-------------------------------ingreso de la altura del usuario  resultado = 2.0  ¡No se produjo ninguna excepción!  the end..... :) |

·

Sin errores y al final el bloque finally

·

|  |
| --- |
| Ingrese un número: dos#<--------------------------ingreso de la altura del usuario  Error: Ingrese un número válido  the end..... :) |

·

En este ejemplo, se intenta realizar una casting de str a int no valido, lo cual generará una excepción ValueError. El bloque except captura esta excepción y ejecuta el código especificado dentro de él, en este caso, imprime un mensaje de error.

·

|  |
| --- |
| Ingrese un número: 0#<-------------------------------ingreso de la altura del usuario  Error: División entre cero  the end..... :) |

·

En este ejemplo, se intenta realizar una división entre cero, lo cual generará una excepción ZeroDivisionError. El bloque except captura esta excepción y ejecuta el código especificado dentro de él, en este caso, imprime un mensaje de error.

5. **Declaración** raise

El bloque raise se utiliza para **lanzar** una excepción de manera manual, lo que permite que el programador controle cuándo y qué tipo de excepción debe ser generada. Esto es útil cuando se detectan condiciones excepcionales dentro de una función que, aunque no son errores propiamente dichos, deben ser manejadas de forma especial.

"raiser" es una declaración para generar manualmente no un error, sino una “excepción” programada, lanzar explícitamente en un lugar o tiempo específico del código.

|  |
| --- |
| def dividir(a, b):  if b == 0:  raise ValueError("Nunca dividirás por cero")  return a / b  try:  resultado = dividir(10, 0)  except ValueError as Error:  print(f"Error encontrado: {Error}") |

·

|  |
| --- |
| Error encontrado: Nunca dividirás por cero |

|  |
| --- |
| La sintaxis básica de la declaración raise es la siguiente:  raise TipoDeExcepcion("Mensaje de error opcional") |

Donde TipoDeExcepcion es el tipo de excepción que se desea lanzar, como ValueError, TypeError, FileNotFoundError, o un error opcional que se mostrará cuando se capture la excepción.

La declaración raise se lanza cuando el programador encuentra un evento por el cual genera una excepción propia en este caso un ValueError con el mensaje "Nunca dividirás por cero".

Luego, se captura la excepción en el bloque except y se imprime el mensaje .

**No hay error, no hay excepción por bloqueo de script. Raise permite controlar el flujo del programa y señalar condiciones excepcionales de acuerdo con tus necesidades.**

* **Uso de** raise: Con raise, se puede crear un flujo de trabajo más controlado. Por ejemplo, en lugar de permitir que el programa continúe operando con datos erróneos, se puede lanzar una excepción para detener el flujo y señalar al usuario o al programador que algo no está bien.

Esto es especialmente útil en la validación de entradas, cuando es necesario asegurarse de que los datos cumplen con ciertos criterios antes de proceder.

**Importancia del manejo de excepciones**

El manejo adecuado de excepciones en un programa no solo mejora la robustez y fiabilidad del código, sino que también permite crear programas más legibles y mantenibles. Al anticipar errores y manejarlos de manera controlada, evitamos comportamientos indeseados como la detención abrupta del programa o resultados inesperados.

Además, Python proporciona la capacidad de crear excepciones personalizadas para manejar errores específicos de una aplicación. Esta flexibilidad es fundamental para construir aplicaciones complejas donde diferentes módulos pueden necesitar formas especializadas de manejo de errores.

**Resumen práctico**

El manejo de excepciones en Python proporciona una herramienta poderosa para asegurar que los programas sean más robustos, seguros y fáciles de depurar. Permite gestionar los errores de manera controlada, mejorar la experiencia del usuario y garantizar que el sistema responda adecuadamente incluso cuando algo sale mal.

Además, la correcta implementación de try, except, else, finally, y raise permite que el flujo del programa siga su curso sin interrupciones críticas, asegurando un manejo eficaz y eficiente de situaciones inesperadas.

Python no se rompe. El manejo de excepciones es una técnica para detectar y responder adecuadamente a situaciones excepcionales sin que el script se detenga abruptamente.

|  |
| --- |
| Es muy importante aclarar que el flujo del programa se altera con try except y finally |