

UNIDAD 2.

TRATAMIENTO DE

EXCEPCIONES

Loreto Pelegrín Castillo



MINISTERIO
DE EDUCACIÓN
Y FORMACIÓN PROFESIONAL



Índice:

1. Introducción
2. Excepciones en Python
3. Programación defensiva
4. Depuración de código

1

Introducción

1. Introducción

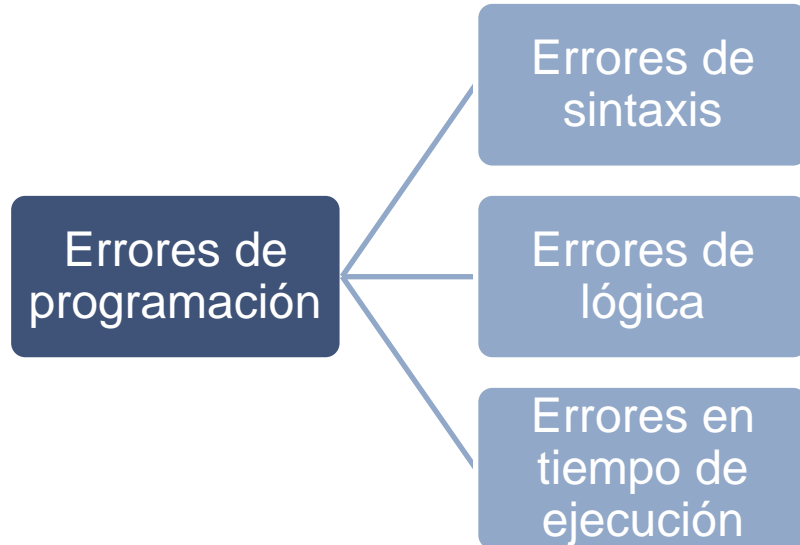
- Para controlar los errores de código que nos surgen a la hora de programar, aparece el concepto de **excepción**.
- Es una herramienta presente en prácticamente todos los lenguajes de programación modernos.

2

Excepciones en Python

2. Excepciones en Python

- Cuando el intérprete de Python encuentra algún error en el programa, ya sea antes o durante la ejecución de este, inmediatamente se detiene.



2.1. Errores de sintaxis

■ Son los más fáciles de solucionar y hay que hacerlo antes de iniciar el programa.

```
#Error de sintaxis

nombre = input("¿Como te llamas?")
print(f"Me llamo {nombres}")
```

■ El intérprete no da error, pero sí en tiempo de ejecución ya que no encuentra la variable nombre.

```
¿Como te llamas?Loreto
Traceback (most recent call last):
  File "c:\Users\narue\Desktop\Academia Avanza\Python\2.DESARROLLO Y VISUALIZACIÓN DE DATOS CON PYTHON - IFCD0011\Unidad2-Excepciones\Ejemplos\ejemploErrores.py", line 4, in <module>
    print(f"Me llamo {nombres}")
                        ^^^^^^^
NameError: name 'nombres' is not defined. Did you mean: 'nombre'?
PS C:\Users\narue\Desktop\Academia Avanza\Python\2.DESARROLLO Y VISUALIZACIÓN DE DATOS CON PYTHON - IFCD0011\Unidad2-Excepciones\Ejemplos> █
```

2.2. Errores de lógica

- Los **errores de lógica** son más sutiles y pueden hacer que tu programa produzca resultados incorrectos sin generar ningún mensaje de error. Estos errores se producen cuando el algoritmo del programa contiene una **falla en su razonamiento o en su implementación**.
- Ocurren en tiempo de ejecución y se controlan mediante las **excepciones**.
- Por ejemplo, cuando las condiciones en un *if*, *elif* o *else* no se expresan correctamente, el flujo del programa puede desviarse hacia una rama equivocada.

2.3. Errores en tiempo de ejecución

- Los **errores en tiempo de ejecución** son aquellos que aparecen solo después de **compilar y ejecutar el código**. Esto implica código que puede parecer correcto en que no tiene errores de sintaxis, pero que no se ejecutarán. Se controla mediante **excepciones**.
- Por ejemplo, podría escribir correctamente una línea de código para abrir un archivo. Pero si el archivo no existe, la aplicación no puede abrir el archivo y produce una excepción.

2.4. Tipos de excepciones más comunes

- Python posee múltiples de excepciones ya definidas. En el ejemplo anterior, aparecía *NameError*, uno de los errores mas comunes.
- Aunque Python tenga definidas una serie de excepciones puede ocurrir que el interprete nos proporciones información poco exacta si el error es complejo.
- <https://docs.python.org/es/3/tutorial/errors.html>
- Excepciones definidas:
<https://docs.python.org/es/3/library/exceptions.html#builtin-exceptions>

2.4. Tipos de excepciones más comunes

Función	Descripción
NameError	No se encuentra la variable con ese nombre
TypeError	Cuando a una operación se le proporciona un argumento de tipo no adecuado.
ValueError	Valor del argumento de la operación no apropiado
ImportError	Intentamos importar un módulo que no se encuentra
KeyError	Se busca una clave de un diccionario y no se puede encontrar
IndentationError	Cuando tenemos una indentación incorrecta en el código
IndexError	Cuando buscamos un índice en una secuencia y este se sale de rango

2.5. Manejo de excepciones

- Para manejar las excepciones Python nos proporciona tres palabras reservadas:
- **try:** Permite envolver el bloque de código donde pensamos que puede haber un error para controlarlo en caso de que ocurra.
- **except:** Nos permite indicar un mensaje con las acciones que llevar a cabo para solucionar el problema. Esta permitido usar varios *except* de manera consecutiva para capturar varios errores. Solamente el ultimo puede ser por defecto, los demás tienen que indicar el tipo de error que es. Se puede añadir un *else* al final. Se ejecutará si el *try* ha funcionado correctamente.
- **finally:** Es opcional, se coloca al final del bloque *try-except*. Siempre se ejecutará independientemente de lo que ocurra con la excepción.

2.5. Manejo de excepciones

```
#Version 1
#Error al dividir por cero

num1,num2=8,0

try:
    print(num1/num2)
except:
    print("No esta permitido dividir por 0")
```



```
No esta permitido dividir por 0
Se ha producido un error de otro tipo
Hay algun tipo de error en el programa
```



```
#Version 2
#Algun tipo de error en el programa

num1,num2="8",0

try:
    print(num1/num2)


#Debemos introducir un tipo de error
except ZeroDivisionError:
    print("No esta permitido dividir por 0")

except:
    print("Se ha producido un error de otro tipo")

finally:
    print("Hay algun tipo de error en el programa")
```

2.5. Manejo de excepciones

```
#Ejemplo con la sentencia else
try:
    nombre = input("Introduce tu nombre: ")
    if type(nombre) == str:
        print(f"Mi nombre es {nombre}")
except:
    print("Introduce bien el nombre")
else:
    #Al funcionar bien el try se ejecuta el bloque else
    print("Todo ha ido correctamente")
```



```
Introduce tu nombre: Loreto
Mi nombre es Loreto
Todo ha ido correctamente
```

2.5. Manejo de excepciones

```
#Calcular el cubo de un numero
try:
    numero = int(input("Introduce un numero: "))
    print("El cubo es:"+str(numero**3))
#Guardamos la excepcion en la variable error
except Exception as error:
    #__name__ es un método especial para obtener información.
    print(f"El error cometido es de tipo {type(error).__name__}")
```

Introduce un numero: 3
El cubo es:27

Introduce un numero: ddf
El error cometido es de tipo ValueError

Ejercicios de clase.

■ Ejercicio 1:

- Escribe un programa que pida al usuario un número entero y lo convierta a un tipo entero. Maneja las excepciones si el usuario ingresa un valor no numérico. (No hacer casting en el input)

■ Ejercicio 2:

- Crea un programa que tenga una lista de nombres y permita al usuario acceder a un índice específico. Maneja la excepción si el índice está fuera del rango.

Ejercicios de clase.

■ Ejercicio 3:

- Crea un programa que pida al usuario ingresar números y calcule su suma. Termina el bucle cuando el usuario ingresa "fin". Maneja la excepción para entradas no numéricas.

■ Ejercicio 4:

- Crea un programa que pida al usuario su edad y verifique que sea un número válido y mayor que 0.

Ejercicios de clase.

■ Ejercicio 5:

- Escribe un programa que calcule la raíz cuadrada de un número ingresado por el usuario. Maneja la excepción si el número es negativo.

■ Ejercicio 6:

- Escribe un programa que intente convertir todos los elementos de una lista a enteros. Maneja las excepciones para cada conversión individual.

Ejercicios de clase.

■ Ejercicio 7:

- Crea un programa que simule un registro de usuarios. Pide al usuario un nombre de usuario y asegúrate de que no esté vacío.

■ Ejercicio 8:

- Crea un programa que permita al usuario buscar un valor en un diccionario utilizando una clave. Maneja la excepción si la clave no existe.

3

Programación defensiva

3. Programación defensiva

- En Python existe un tipo de programación que consiste en la anticipación de nuestros errores.
- Para ello se utilizan las palabras reservadas **assert** y **raise**.
- Estas palabras nos permiten implementar algunas características de lo que se conoce como **programación defensiva**.

3.1. Assert

- La palabra reservada **assert** nos permite introducir una condición en nuestro código, si la condición devuelve True, el código se ejecuta, en caso contrario devuelve un **AssertionError**.
- Si se quiere introducir algún tipo de mensaje por si no se cumple la condición, podemos introducirlo a continuación de la sentencia assert.
- Es una forma de verificar que tu código se está comportando como esperas y puede ayudarte a detectar errores.

3.1. Assert

- Cuándo usar assert:
 - Pruebas unitarias: Para verificar que tus funciones se comporten como se espera en condiciones normales.
 - Precondiciones: Para asegurar que los argumentos de una función sean válidos antes de ejecutar el código.
 - Postcondiciones: Para verificar que el resultado de una función cumple con ciertas propiedades.
- Assert no se utiliza en modo de producción.

3.1. Assert

```
#Ejemplo dividir entre 0

def divide(x, y):
    assert y != 0, "División por cero"
    return x / y

resultado = divide(10, 2) # Funciona correctamente
resultado = divide(10, 0) # Lanza AssertionError: División por cero
```


3.2. Raise

- La palabra clave **raise** se utiliza para lanzar una excepción de forma explícita.
- Puedes lanzar cualquier tipo de excepción definida en Python o crear tus propias clases de excepción.
- *raise TipoDeExcepcion("Mensaje de error")*
 - *Donde TipoDeExcepcion es el tipo de excepción que deseas lanzar. Puede ser una excepción predefinida en Python (como ValueError, TypeError, etc.) o una clase de excepción que hayas definido tú mismo.*

3.2. Raise

```
def sumar(a, b):  
    #Utilizamos isinstance para controlar si son instancias de ese tipo los parametros a y b  
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):  
        #Si la validación anterior falla, se lanza una excepción de tipo TypeError con el mensaje "Los argumentos deben ser números."  
        # Esto indica que se ha producido un error de tipo de dato.  
        raise TypeError("Los argumentos deben ser números.")  
    return a + b  
  
try:  
    resultado = sumar("hola", 5)  
except TypeError as e:  
    # Si se produce una excepción de tipo TypeError durante la ejecución del bloque try, se captura la excepción y  
    # se imprime un mensaje de error junto con el mensaje de la excepción declarada en el raise.  
    print("Error:", e) #Error: Los argumentos deben ser números.
```

3.2. Raise

- Crear una clase personalizada de error sirve para definir el comportamiento exacto que deseas cuando se produce un error específico.
- Por ejemplo:

```
#Crear clase de error personalizada
#Hereda de la clase Exception
class MiErrorPersonalizado(Exception):
    """Esta es una excepción personalizada para representar un error específico."""
    def __init__(self, mensaje, codigo):
        #Super para llamar al constructor de la clase padre (Exception)
        # Primero se inicializan los atributos mensaje y codigo de nuestra clase personalizada.
        # Luego, se llama a super().__init__(mensaje) para pasar el mensaje de error al constructor de la clase base Exception.
        # Esto asegura que el mensaje de error se almacene correctamente en la instancia de la excepción.
        self.mensaje = mensaje
        self.codigo = codigo
        super().__init__(mensaje)
```

3.2. Raise

```
#Funcion de divisor
def dividir(a, b):
    #Control de que el divisor no sea 0
    if b == 0:
        #Utilizamos la clase personalizada, y le pasamos como parametros el mensaje
        #y la función donde esta saltando la excepcion
        raise MiErrorPersonalizado("No puedes dividir por cero", "def dividir")
    return a / b

try:
    resultado = dividir(10, 0)
except MiErrorPersonalizado as e:
    print("Se produjo un error:", e.mensaje, "Código:", e.codigo)
```



Se produjo un error: No puedes dividir por cero Código: def dividir

3.2. Raise

Otro ejemplo:

```
class ErrorConversionAFloat(Exception):
    """Excepción personalizada para errores al convertir a float."""
    def __init__(self, mensaje):
        self.mensaje = mensaje
        super().__init__(self.mensaje)

def convertir_a_float(cadena):
    """Convierte una cadena a un número de punto flotante.

    Args:
        cadena: La cadena a convertir.

    Returns:
        El número de punto flotante.

    Raises:
        ErrorConversionAFloat: Si la cadena no puede ser convertida a float.
    """

    try:
        return float(cadena)
    except ValueError:
        raise ErrorConversionAFloat(f"No se pudo convertir '{cadena}' a un número flotante.")
```

```
cadena = input("Ingrese un número: ")
try:
    resultado = convertir_a_float(cadena)
    print("El número es:", resultado)
except ErrorConversionAFloat as e:
    print("Error:", e.mensaje)
```



```
Ingrese un número: f
Error: No se pudo convertir 'f' a un número flotante.
```

Ejercicios de clase.

Ejercicio 1: Crear una calculadora básica que pueda realizar sumas, restas, multiplicaciones y divisiones.

- ▷ Añadir al menos una excepción personalizada:
- ▷ Realizar validación de datos:
 - ▷ Imprimir un mensaje claro cuando se produzca una excepción.

4

Depuración de código

4. Depuración de código

■ A la hora de depurar nuestros programas no existe una metodología establecida.

■ **Consejos:**

- Los editores de códigos modernos, están diseñados para mostrar todos los fallos en la **sintaxis del programa**. Y a veces nos muestras igualmente algunos errores lógicos como por ejemplo declarar variables que luego no usamos.

4. Depuración de código

- Introducir varias **sentencias print()** en nuestro código para mostrar los valores que esta manejando el programa en un momento determinado.
- Todos los editores hoy en día incluyen **puntos de ruptura**. Son marcas que hacemos en el código durante el proceso de depuración y que nos permiten detener la ejecución del programa en ese punto. Esto posibilita que, si tenemos algún fallo, podamos establecer un punto de ruptura y a partir de ahí ver línea a línea toda la información de lo que esta pasando en el programa

Ejercicios de clase.

■ Ejercicio 1:

- ▶ Depurar el código del archivo Ejercicio1-Clase.py para encontrar los fallos existentes y conseguir que funcione de manera lógica.



¡Gracias!