ESTRUCTURA DE DATOS Y PROGRAMACIÓN ORIENTADA A OBJETOS

Tema Nº2:Excepciones

Indicador de logro Nº2:Evalúa los errores más comunes y las diferentes excepciones al ejecutar un programa utilizando un lenguaje de programación.

**TEMA 01 Teoría de los**

Imagen que contiene Icono

Descripción generada automáticamente

**TEMA Nº2:**

Excepciones

**Subtema 2.1:**

Errores: el pan de cada día del programador.

Las excepciones son la herramienta que implementa Python para manejar los errores potenciales de un programa. Cuando una porción de código quiere indicar que ocurrió un error, se dice que debe lanzar una excepción, mientras que, cuando otra quiere saber si surgió un error y actuar en consecuencia, se dice que debe capturarla o manejarla. Si una excepción es lanzada y ningún código la captura, el programa finaliza.

Las excepciones están generalmente acompañadas de un mensaje que nos indica qué salió mal y en dónde (nombre del archivo y la línea). En el caso de la consola interactiva, las excepciones no manejadas se imprimen automáticamente en la pantalla, pero, como se dijo anteriormente, para programas reales un hecho tal implica la finalización del mismo.

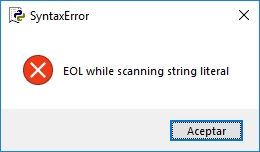
Las excepciones son, en definitiva, clases. Por ello llevan la nomenclatura denominada CamelCase. ValueError es una excepción incorporada (definida por Python) que es lanzada cuando una función espera un valor determinado, pero recibe otro.

* **ERRORES**

Los errores detienen la ejecución del programa y tienen varias causas. Para poder estudiarlos mejor vamos a provocar algunos intencionadamente:

**ERRORES DE SINTAXIS:**

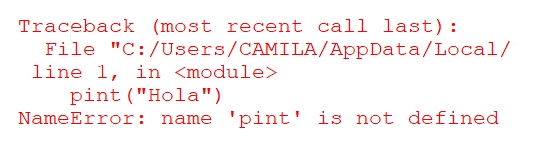
Identificados con el código **SyntaxError**, son los que podemos apreciar repasando el código, por ejemplo, al dejarnos de cerrar un paréntesis:

Código

print("Hola

**ERRORES DE NOMBRE:**

Se producen cuando el sistema interpreta que debe ejecutar alguna función, método, etc. pero no lo encuentra definido. Devuelven el código **NameError**.

Código

pint("Hola")

La mayoría de errores sintácticos y de nombre los identifican los editores de código antes de la ejecución, pero existen otros tipos que pasan más desapercibidos.

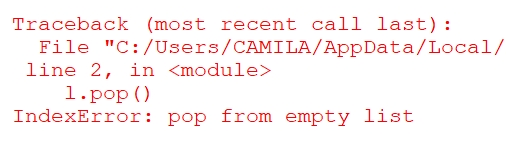
**ERRORES SEMÁNTICOS:**

Estos errores son muy difíciles de identificar porque van ligados al sentido del funcionamiento y dependen de la situación. Algunas veces pueden ocurrir y otras no.

La mejor forma de prevenirlos es programando mucho y aprendiendo de tus propios fallos, la experiencia es la clave. Veamos un par de ejemplos:

Ejemplo **pop()** con lista vacía:

Si intentamos sacar un elemento de una lista vacía, algo que no tiene mucho sentido, el programa dará fallo de tipo **IndexError**. Esta situación ocurre sólo durante la ejecución del programa, por lo que los editores no lo detectarán:



Código

l = []

l.pop()

Para prevenir el error deberíamos comprobar que una lista tenga como mínimo un elemento antes de intentar sacarlo, algo factible utilizando la función **len()**.

l = []

if len(l) > 0:

l.pop()

Ejemplo lectura de cadena y operación sin conversión a número.

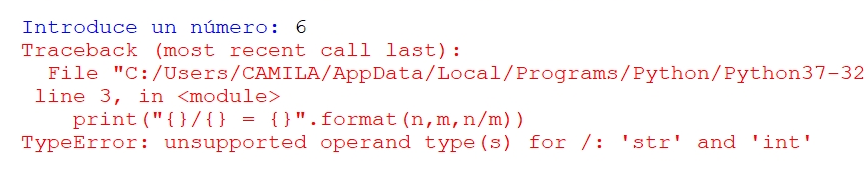
Cuando leemos un valor con la función **input()**, éste siempre se obtendrá como una cadena de caracteres. Si intentamos operarlo directamente con otros números tendremos un fallo **TypeError** que tampoco detectan los editores de código.

Código

n = input("Introduce un número: ")

m = 4

print("{}/{} = {}".format(n,m,n/m))



Como ya sabemos este error se puede prevenir transformando la cadena a entero o flotante:

Código

n = float(input("Introduce un número: "))

m = 4

print("{}/{} = {}".format(n,m,n/m))

Sin embargo, no siempre se puede prevenir, como cuando se introduce una cadena que no es un número:

Código

n = float(input("Introduce un número: "))

m = 4

print("{}/{} = {}".format(n,m,n/m))

Es difícil prevenir fallos que ni siquiera nos habíamos planteado que podían existir. Por suerte para esas situaciones existen las excepciones.

* **EXCEPCIONES**

Las excepciones son bloques de código que nos permiten continuar con la ejecución de un programa pese a que ocurra un error. Siguiendo con el ejemplo de la lección anterior, teníamos el caso en que leíamos un número por teclado, pero el usuario no introducía un número:

Código

n = float(input("Introduce un número: "))

m = 4

print("{}/{} = {}".format(n,m,n/m))



**BLOQUES TRY – EXCEPT**:

Para prevenir el fallo debemos poner el código propenso a errores en un bloque **try** y luego encadenar un bloque **except** para tratar la situación excepcional mostrando que ha ocurrido un fallo:

Código

try:

n = float(input("Introduce un número: "))

m = 4

print("{}/{} = {}".format(n,m,n/m))

except:

print("Ha ocurrido un error, introduce bien el número")

Esta forma nos permite controlar situaciones excepcionales que generalmente darían error y en su lugar mostrar un mensaje o ejecutar una pieza de código alternativo.

Podemos aprovechar las excepciones para forzar al usuario a introducir un número haciendo uso de un bucle **while**, repitiendo la lectura por teclado hasta que lo haga bien y entonces romper el bucle con un **break**.

Código

while(True):

try:

n = float(input("Introduce un número: "))

m = 4

print("{}/{} = {}".format(n,m,n/m))

break # Importante romper la iteración si todo ha salido bien

except:

print("Ha ocurrido un error, introduce bien el número")

**BLOQUE ELSE:**

Es posible encadenar un bloque **else** después del **except** para comprobar el caso en que todo funcione correctamente (no se ejecuta la excepción). El bloque **else** es un buen momento para romper la iteración con break si todo funciona correctamente:

Código

while(True):

try:

n = float(input("Introduce un número: "))

m = 4

print("{}/{} = {}".format(n,m,n/m))

except:

print("Ha ocurrido un error, introduce bien el número")

else:

print("Todo ha funcionado correctamente")

break # Importante romper la iteración si todo ha salido bien

**BLOQUE FINALLY:**

Por último, es posible utilizar un bloque **finally** que se ejecute al final del código, ocurra o no ocurra un error:

Código

while(True):

try:

n = float(input("Introduce un número: "))

m = 4

print("{}/{} = {}".format(n,m,n/m))

except:

print("Ha ocurrido un error, introduce bien el número")

else:

print("Todo ha funcionado correctamente")

break # Importante romper la iteración si todo ha salido bien

finally:

print("Fin de la iteración") # Siempre se ejecuta

**Actividad:**

**CASO PRÁCTICO 1:**

Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

**Ejercicio:**

resultado = 10/0

**Solución:**

try:

resultado = 10/0

except ZeroDivisionError:

print("Error:\tNo es posible dividir entre cero,\n"

"\tdebes introducir un número distinto.")

**Resultado:**

Error: No es posible dividir entre cero,

debes introducir un número distinto.

**CASO PRÁCTICO 2:**

Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

**Ejercicio:**

lista = [1, 2, 3, 4, 5]

lista[10]

**Ejecución:**

lista = [1, 2, 3, 4, 5]

try:

lista[10]

except IndexError:

print("Error:\tEl índice se encuentra fuera del rango,\n"

"\tdebes utilizar un número mayor o igual que cero\n"

"\ty menor que la longitud de la lista.")

**Resultado:**

Error: El índice se encuentra fuera del rango,

debes utilizar un número mayor o igual que cero

y menor que la longitud de la lista.

**CASO PRÁCTICO 3:**

Crea el siguiente módulo:

El módulo se denominará **operaciones.py** y contendrá 4 funciones para realizar una suma, una resta, un producto y una división entre dos números. Todas ellas devolverán el resultado. En las funciones del módulo deberá de haber tratamiento e invocación manual de errores para evitar que se quede bloqueada una funcionalidad, eso incluye:

* **TypeError:** En caso de que se envíen valores a las funciones que no sean números. Además, deberá aparecer un mensaje que informe Error: Tipo de dato no válido.
* **ZeroDivisionError**: En caso de realizar una división por cero. Además, deberá aparecer un mensaje que informe Error: No es posible dividir entre cero.

Una vez creado el módulo, crea un script **calculos.py** en el mismo directorio en el que deberás importar el módulo y realizar las siguientes instrucciones. Observa si el comportamiento es el esperado:

**calculos.py**

from operaciones import \*

a, b, c, d = (10, 5, 0, "Hola")

print( "{} + {} = {}".format(a, b, suma(a, b) ) )

print( "{} - {} = {}".format(b, d, resta(b, d) ) )

print( "{} \* {} = {}".format(b, b, producto(b, b) ) )

print( "{} / {} = {}".format(a, c, division(a, c) ) )

**SOLUCIÓN:**

**operaciones.py**

def suma(a, b):

try:

r = a + b

except TypeError:

print("Error: Tipo de dato no válido")

else:

return r

def resta(a, b):

try:

r = a - b

except TypeError:

print("Error: Tipo de dato no válido")

else:

return r

def producto(a ,b):

try:

r = a \* b

except TypeError:

print("Error: Tipo de dato no válido")

else:

return r

def division(a, b):

try:

r = a / b

except TypeError:

print("Error: Tipo de dato no válido")

except ZeroDivisionError:

print("Error: No es posible dividir entre cero")

else:

return r

**Ejecución:**

python calculos.py

**Resultado**

10 + 5 = 15

Error: Tipo de dato no válido

5 - Hola = None

5 \* 5 = 25

Error: No es posible dividir entre cero

10 / 0 = None