# **Errores y excepciones**

En algunas ocasiones nuestros programas pueden fallar ocasionando su detención. Ya sea por errores de sintaxis o de lógica, tenemos que que ser capaces de detectar esos momentos y tratarlos debidamente para prevenirlos.

# **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:

print("Hola"

File "<ipython-input-1-8bc9f5174855>", line 1

print("Hola"

^

SyntaxError: unexpected EOF while parsing

## **Errores de nombre**

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

pint("Hola")

<ipython-input-2-155163d628c2> in <module>()

----> 1 pint("Hola")

NameError: name 'pint' is not defined

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:

l = []

l.pop()

<ipython-input-6-9e6f3717293a> in <module>()

----> 1 l.pop()

IndexError: pop from empty list

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:

n = input("Introduce un número: ")

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

Introduce un número: 4

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-12-85bb893ab3e3> in <module>()

----> 1 print("{}/{} = {}".format(n,m,n/m))

TypeError: unsupported operand type(s) for /: 'str' and 'int'

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

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

m = 4

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

Introduce un número: 10

10.0/4 = 2.5

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

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

m = 4

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

Introduce un número: aaa

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-14-c0e7fd4a26a9> in <module>()

----> 1 n = float(input("Introduce un número: "))

2 m = 4

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

ValueError: could not convert string to float: 'aaa'

Como podéis suponer, 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:

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

m = 4

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

Introduce un número: aaa

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-14-c0e7fd4a26a9> in <module>()

----> 1 n = float(input("Introduce un número: "))

2 m = 4

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

ValueError: could not convert string to float: 'aaa'

## **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:

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")

Introduce un número: aaa

Ha ocurrido un error, introduce bien el número

Como vemos 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:

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")

Introduce un número: aaa

Ha ocurrido un error, introduce bien el número

Introduce un número: sdsdsd

Ha ocurrido un error, introduce bien el número

Introduce un número: sdsdsd

Ha ocurrido un error, introduce bien el número

Introduce un número: sdsd

Ha ocurrido un error, introduce bien el número

Introduce un número: 10

10.0/4 = 2.5

## **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:

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

Introduce un número: 10

10.0/4 = 2.5

Todo ha funcionado correctamente

## **Bloque finally**

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

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

Introduce un número: aaa

Ha ocurrido un error, introduce bien el número

Fin de la iteración

Introduce un número: 10

10.0/4 = 2.5

Todo ha funcionado correctamente

Fin de la iteración

# **Excepciones múltiples**

En una misma pieza de código pueden ocurrir muchos errores distintos y quizá nos interese actuar de forma diferente en cada caso.

Para esas situaciones algo que podemos hacer es asignar una excepción a una variable.

De esta forma es posible analizar el tipo de error que sucede gracias a su identificador:

try:

n = input("Introduce un número: ") # no transformamos a número

5/n

except Exception as e: # guardamos la excepción como una variable e

print("Ha ocurrido un error =>", type(e).\_\_name\_\_)

Introduce un número: 10

Ha ocurrido un error => TypeError

Cada error tiene un identificador único que curiosamente se corresponde con su tipo de dato. Aprovechándonos de eso podemos mostrar la clase del error utilizando la sintaxis:

print( type(e) )

<class 'TypeError'>

Es similar a conseguir el tipo (o clase) de cualquier otra variable o valor literal:

print(type(1))

print(type(3.14))

print(type([]))

print(type(()))

print(type({}))

<class 'int'>

<class 'float'>

<class 'list'>

<class 'tuple'>

<class 'dict'>

Como vemos siempre nos indica eso de "class" delante. Eso es porque en Python todo son clases, pero hablaremos de este concepto más adelante. Lo importante ahora es que podemos mostrar solo el nombre del tipo de dato (la clase) consultando su propiedas especial name de la siguiente forma:

print( type(e).\_\_name\_\_)

print(type(1).\_\_name\_\_)

print(type(3.14).\_\_name\_\_)

print(type([]).\_\_name\_\_)

print(type(()).\_\_name\_\_)

print(type({}).\_\_name\_\_)

TypeError

int

float

list

tuple

dict

Gracias a los identificadores de errores podemos crear múltiples comprobaciones, siempre que dejemos en último lugar la excepción por defecto Excepcion que engloba cualquier tipo de error (si la pusiéramos al principio las demas excepciones nunca se ejecutarían):

try:

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

5/n

except TypeError:

print("No se puede dividir el número entre una cadena")

except ValueError:

print("Debes introducir una cadena que sea un número")

except ZeroDivisionError:

print("No se puede dividir por cero, prueba otro número")

except Exception as e:

print("Ha ocurrido un error no previsto", type(e).\_\_name\_\_ )

Introduce un número divisor: 0

No se puede dividir por cero, prueba otro número

# **Invocación de excepciones**

En algunas ocasiones quizá nos interesa llamar un error manualmente, ya que un print común no es muy elegante:

def mi\_funcion(algo=None):

if algo is None:

print("Error! No se permite un valor nulo (con un print)")

mi\_funcion()

Error! No se permite un valor nulo (con un print)

## **Instrucción raise**

Gracias a raise podemos lanzar un error manual pasándole el identificador. Luego simplemente podemos añadir un except para tratar esta excepción que hemos lanzado:

def mi\_funcion(algo=None):

try:

if algo is None:

raise ValueError("Error! No se permite un valor nulo")

except ValueError:

print("Error! No se permite un valor nulo (desde la excepción)")

mi\_funcion()

Error! No se permite un valor nulo (desde la excepción)