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.

Última edición: 26 de Septiembre de 2018