

Python: excepciones

SISTEMAS DE GESTIÓN EMPRESARIAL – 148FA (DAM)

Tipos de errores en Python

En un programa nos podemos encontrar con varios tipos de errores. Estos pueden ser los principales o más comunes:

- **Errores de sintaxis** (*SyntaxError*): generalmente son consecuencia de equivocaciones al escribir el programa (por ejemplo si escribimos `wile` en vez de `while`).
- **Errores semánticos**: ocurren cuando el programa, a pesar de no generar ningún mensaje de error, no devuelve el resultado esperado. Puede ocurrir si tenemos un algoritmo incorrecto o falta alguna sentencia dentro del programa.
- **Errores de ejecución**: ocurren durante la ejecución del programa. Pueden ocurrir por el uso incorrecto del programa por parte del usuario (por ejemplo si el usuario ingresa una cadena de caracteres y no un número). Puede ocurrir también si el programa no está bien planteado o si los archivos que se vayan a utilizar en el programa (un documento `.txt`, por ejemplo) están dañados.

Excepciones

Los errores de ejecución son llamados comúnmente ***excepciones***

Durante la ejecución de un programa, dentro de una función puede surgir una excepción. Dependiendo de cuál sea el error y si tenemos definida una excepción, la ejecución del programa accederá a dicha parte del código. Si no es controlada, la ejecución del programa terminará.

Manejo de excepciones

Cada lenguaje tiene palabras reservadas que permiten manejar las excepciones para evitar la interrupción del programa o realizar acciones adicionales antes de su interrupción.

Sentencias para el manejo de excepciones en Python:

- **try**
- **except**
- **finally**

Excepciones

try

Dentro del bloque try se encuentra todo el código que puede levantar una excepción

except

Este bloque se encarga de capturar la excepción para poder procesarla.

```
>>> dividendo = 5
>>> divisor = 0
>>> dividendo / divisor
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: division by zero
```

En este ejemplo se levanta la excepción **ZeroDivisionError** al hacer la división entre 0.

Para evitar que se levante la excepción y se detenga la ejecución del programa, se deberá de usar el bloque **try-except**

Excepciones

Bloque try-except

```
try:
    cociente = dividendo / divisor
except:
    print ("No se permite la división por cero")
```

- Dentro del mismo bloque try pueden ocurrir excepciones de diferente tipo.
- Es posible utilizar varios bloques except, cada uno para capturar un tipo distinto de excepción.

Excepciones

- Dentro del mismo bloque try **pueden ocurrir excepciones de diferente tipo.**
- Es **posible utilizar varios bloques except**, cada uno para capturar un tipo distinto de excepción.
- Para ello debemos especificar a continuación de la sentencia except el **nombre de la excepción** que se pretende capturar.
- **Un mismo bloque except puede atrapar varios tipos de excepciones.** Para ello se deben especificar los nombres de cada una, separados por comas a continuación de la palabra except.
- Después de un bloque try, si existen varios bloques except, **se ejecutará sólo uno de ellos.**

Excepciones

try:

aquí ponemos el código que puede lanzar excepciones

except IOError:

entrará aquí en caso que se haya producido # una excepción IOError

except ZeroDivisionError:

entrará aquí en caso que se haya producido # una excepción ZeroDivisionError

except:

entrará aquí en caso que se haya producido

una excepción que no corresponda a ninguno

de los tipos especificados en los except previos

Excepciones

- Podemos utilizar una sentencia **except sin especificar el tipo de excepción** a capturar. En este caso se captura cualquiera, sin tener en cuenta su tipo.
- Nota: para ello debemos indicarlo **como último bloque except**. Es decir, primero habrá que indicar las que están asociadas a un tipo y finalmente la que no tiene en cuenta su tipo.

Excepciones

Bloque finally

- Después de try-except, se puede incluir el bloque finally, donde se escriben las **sentencias de finalización**.
- Se consideran acciones de limpieza.
- Este bloque **se ejecuta siempre**, sin importar que haya surgido una excepción o no.
- Si hay un bloque except, no es necesario incluir un bloque finally (es opcional)
- Es posible tener un bloque try sólo con finally, sin el bloque except.

Excepciones

Ejecución de bloques try-except-finally

- Python comienza a ejecutar las instrucciones que se encuentran dentro de un bloque try.
- Si durante la ejecución de las instrucciones se levanta una excepción, se interrumpe la ejecución en el punto donde surge la excepción y pasa a la ejecución del bloque except correspondiente.
- Si no se encuentra ningún bloque except del tipo correspondiente, pero hay un bloque except sin tipo, lo ejecuta.
- Al terminar de ejecutar el bloque except correspondiente, se ejecuta el bloque finally, si ha sido definido.
- Si no se ha levantado ninguna excepción dentro del bloque, se completa su ejecución y pasa directamente a la ejecución del bloque finally, en caso de ser definido.

Excepciones - Ejemplo

Ejemplo

- Nuestro programa tiene que procesar información que ingresa el usuario, y guardarla en un archivo.
- El acceso a archivos puede levantar alguna excepción, por lo que dentro del **bloque try** incluiremos el código de manipulación de archivos.
- **Bloque except:** incluiremos una excepción de **IOError**: tipo de excepción que lanzan las funciones de manipulación de archivos.
- Podemos añadir un bloque except sin un tipo asignado.
- Añadimos un **bloque finally** para cerrar el archivo, haya surgido o no una excepción.

Excepciones - Ejemplo

```
try:
    archivo = open("miarchivo.txt")
    # procesar el archivo

except IOError:
    print ("Error de entrada/salida.")
    # realizar procesamiento adicional

except:
    # procesar la excepción

finally:
    # si el archivo no está cerrado hay que cerrarlo
    if not(archivo.closed):
        archivo.close()
```

Procesamiento y propagación de errores

Qué debemos hacer al levantar o atrapar una excepción

Se puede escribir un archivo de log o mostrar un mensaje por pantalla, con el fin de dejar constancia de la excepción.

Información que mostrar:

- Tipo de excepción ocurrida
- Momento en el que ocurrió
- Llamadas previas a la excepción

El objetivo de esta información es facilitar el diagnóstico en caso de que alguien deba corregir el programa para evitar que la excepción siga apareciendo.

Procesamiento y propagación de errores

raise:

- Sirve para lanzar una excepción, a pesar de no haberse activado automáticamente.
- Si se invoca esta instrucción dentro de un bloque except, sin pasarle parámetros, Python levantará la excepción atrapada por ese bloque.
- Puede ocurrir también que queramos lanzar una excepción distinta, más significativa para el usuario que ha ejecutado la función actual. Para ello utilizaremos la sentencia **raise** indicándole la excepción que se quiere lanzar y los parámetros que queramos.

Excepciones – Ejemplo (raise)

```
>>> raise NameError('Hola Mundo')
```

```
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
NameError: Hola Mundo
```

Excepciones – Información de contexto

Acceder a la información de contexto desde un bloque except:

1. Utilizar la función `exc_info`, del módulo `sys` (`import sys`): devuelve una tupla con información sobre la última excepción atrapada, que contiene tres elementos: tipo de excepción, valor de la excepción y llamadas realizadas.
2. Utilizar la misma sentencia `except` pasándole un identificador para que almacene una referencia a la excepción.

`try:`

código que puede lanzar una excepción

`except Exception, ex:`

procesamiento de la excepción cuya información

es accesible a través del identificador ex

Excepciones – Validaciones

- **Validaciones:** técnicas que permiten asegurar que los valores con los que se vaya a operar estén dentro de un dominio específico.
- Las validaciones muestran al usuario información que le pueda resultar útil para procesar el error.
- Las validaciones son importantes cuando se utilizan entradas del usuario o de un archivo en el programa.
- **Comprobaciones:** comprobar el tipo de dato, comprobar que una variable sea de un tipo en particular...
- **Opciones:** devolver algo que reconozca el usuario. A veces es más cómodo lanzar una excepción.
- **Programación defensiva:** uso intensivo de validaciones.

Comprobaciones por usuario

assert

Con la función **assert** Python asegura una condición que se deba cumplir, y si no se cumple, lanzará la excepción **AssertionError** que se podrá capturar en un **try/except**

```
assert n >= 0 # Asegura que n debe ser mayor o igual a 0
```

```
# Dentro de un bloque try
```

```
try:  
    assert(n>=0)  
except AssertionError:  
    raise ValueError
```



Aseguramos que `n` (variable procedente de una llamada a la función o de sentencias anteriores) sea mayor que 0, o de lo contrario lanzará la excepción **AssertionError**

Comprobaciones por contenido

Comprobar variables

- Es preferible comprobar los datos provistos que hayan sido ingresados por un usuario o provenientes de un archivo para que el funcionamiento del programa sea el adecuado.
- No siempre es posible realizar comprobaciones
 - Podemos realizar acciones automáticas que realicen conversiones de tipos de datos de alguna variable, por ejemplo.
 - Podemos crear un bucle que asegure que el usuario mete bien los datos

Comprobaciones por contenido

```
def lee_entero():  
    ''' Solicita un valor entero y lo devuelve.  
        Mientras el valor ingresado no sea entero, vuelve a solicitarlo. '''  
    while True:  
        valor = input("Ingrese un número entero: ")  
        try:  
            valor = int(valor)  
            return valor  
        except ValueError:  
            print ("ATENCIÓN: Debe ingresar un número entero.")
```

Comprobaciones por contenido

```
def lee_entero():  
    """ Solicita un valor entero y lo devuelve.  
        Si el valor ingresado no es entero, da 5 intentos para ingresarlo correctamente,  
        y de no ser así, lanza una excepción. """  
    intentos = 0  
    while intentos < 5:  
        valor = input("Ingresa un número entero: ")  
        try:  
            valor = int(valor)  
            return valor  
        except ValueError: intentos += 1  
    raise (ValueError, "Valor incorrecto ingresado en 5 intentos")
```

Comprobaciones por contenido

Cuando la entrada ingresada sea una cadena, no es esperable que el usuario la vaya a ingresar en mayúsculas o minúsculas, ambos casos deben ser considerados.

```
def lee_opcion():  
    """ Solicita una opción de menú y la devuelve. """  
    while True:  
        print ("Ingrese A (Altas) - B (Bajas) - M (Modificaciones): “)  
        opcion = input().upper()  
        if opcion in ["A", "B", "M"]:  
            return opcion
```

Comprobaciones por tipo

La función **type**(variable) de Python nos indica el tipo de una variable.

Si queremos comprobar que una variable sea de tipo entero:

```
if type(i) != int:  
    raise (TypeError, "i debe ser del tipo int")
```

Es posible comprobar el tipo de la variable contra una secuencia de tipos posibles:

```
if type(i) not in (int, float, long, complex):  
    raise (TypeError, "i debe ser numérico")
```

Comprobaciones por tipo


Función **isinstance(variable, tipos)**

```
if not isinstance(i, (int, float, long, complex) ):
    raise (TypeError, "i debe ser numérico")
```

Con esto comprobamos si una variable es de determinado tipo o subtipo de éste

Comprobaciones por tipo

```
def division_entera(x,y):  
    """ Calcula la división entera después de  
        convertir los parámetros a enteros. """  
    try:  
        dividendo = int(x)  
        divisor = int(y)  
        return dividendo/divisor  
    except ValueError:  
        raise (ValueError, "x e y deben poder convertirse a enteros")  
    except ZeroDivisionError:  
        raise (ZeroDivisionError, "y no puede ser cero")
```



casting

Notas

- Los errores que se pueden presentar en un programa son: de **sintaxis** (detectados por el intérprete), de **semántica** (el programa no funciona correctamente), o de **ejecución** (excepciones).
- Cuando el código a ejecutar pueda producir una excepción es deseable encerrarlo en los bloques correspondientes para actuar en consecuencia.
- Si no hay una excepción definida para el tipo de error, la ejecución del programa se interrumpe.
- Cuando una porción de código puede levantar diversos tipos de excepciones, **es deseable tratarlas por separado**.
- Cuando se genera una excepción es importante actuar en consecuencia, ya sea mostrando un mensaje de error, guardándolo en un archivo (log), o modificando el resultado final de la función.
- Antes de actuar sobre un dato en una porción de código, es deseable corroborar que se lo pueda utilizar, se puede validar su contenido, su tipo o sus atributos.

Ejercicios

Crea una función llamada **secuencia_fibonacci**, que devuelva una lista con la secuencia de Fibonacci. Como parámetro de entrada debe incluir el número que se pida por pantalla fuera de esta función para definir la longitud de la secuencia, como programa inicial. Este número debe estar entre 1 y 20. Si el usuario mete otro número fuera del rango, lanzará un error.

Crea un programa inicial que incluya los bloques de excepciones **try** y **except**, y si es posible **finally**.

El programa debe solicitar al usuario que inserte un número entero, que debe estar entre el 1 y 20.

- En caso de no ser correcto debe lanzar una excepción con un mensaje
- En caso de introducir un número correcto, debe ejecutar la función **secuencia_Fibonacci**, que sacará por pantalla la secuencia de **n** de longitud (número que meta el usuario por pantalla).