GOBIERNO DE ESPAÑA
MINISTERIO DE HACIENDA Y FUNCIÓN PÚBLICA
INAP
INSTITUTO NACIONAL DE ADMINISTRACIÓN PÚBLICA

Introducción a la programación en Python

Tema 9. Excepciones

Autor: Antonio Villena Martín

Contenido

1. Introducción

Hay que distinguir entre errores de sintaxis y excepciones. Los **errores de sintaxis** son los errores que se cometen al escribir el código y que el intérprete de Python detecta e interrumpe el programa cuando se ejecuta indicando el tipo de error. Estos errores suelen deberse a la falta de algún signo de puntuación como dos puntos (':'), comas (','), indentaciones, etc.

En cambio, las **excepciones**, aunque sintácticamente correctas, puede ser que, al ser ejecutado un determinado código, lance un error en la ejecución, por eso también se les conoce como errores de ejecución. Ejemplos de excepciones pueden ser dividir un numero por cero, conversión de tipos incorrecto, etc.

2. Excepciones y su manejo

Las excepciones permiten continuar con la ejecución de un programa a pesar de que pueda ocurrir un error en la ejecución de un script. Gracias al manejo de excepciones en Python se pueden tomar acciones de recuperación para evitar la interrupción del programa. El lenguaje proporciona una serie de palabras reservadas que se gestionan mediante bloques a través de las sentencias `try`, `except` y `finally`.

Dentro del bloque `try` se coloca el código que pueda llegar a lanzar una excepción. Se usa el término *lanzar* para indicar la acción de generar una excepción.

Posteriormente, se ubica el bloque `except`, que se encarga de *capturar* la excepción y permite procesarla mostrando un mensaje informando, si procede, al usuario.

El siguiente ejemplo es el típico de intentar realizar una división por cero.

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

Se puede aprecia que se ha lanzado la excepción con el identificador `ZeroDivisionError` al intentar realizar la división por cero. Si se pretende que no se detenga la ejecución del programa al hacer este tipo de operaciones, se ha de usar el bloque `try-except`.

```
dividendo = 5
divisor = 0
```

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

El contenido del bloque **`finally`** se ejecuta siempre, como se puede apreciar en el siguiente apartado.

Python dispone de una amplia gama de excepciones predefinidas que extienden de la clase `BaseException`, que se pueden consultar en:

https://docs.python.org/es/3/library/exceptions.html#bltin-exceptions.

Cada excepción está definida por un identificador (como por ejemplo el anteriormente visto `ZeroDivisionError`). Con estas clases predefinidas, se pueden usar para definir otras nuevas como veremos más adelante en el apartado «Excepciones definidas por el usuario».

3. Excepciones múltiples

En un mismo bloque `try`, se pueden producir diferentes tipos de excepciones. Un mismo bloque `except` puede atrapar varios tipos de excepciones especificando los nombres de las excepciones (identificador) que se quieren capturar.

Un bloque `try` puede albergar varios bloques `except`.

```
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
```

Puede ubicarse un bloque `finally` al final donde se escriben las sentencias de finalización, que suelen usarse para limpieza. Este bloque se ejecuta siempre, haya o no surgido una excepción. Si hay un bloque `except` no es necesario que esté presenta `finally`. También puede darse el caso de tener un bloque `try` solo con `finally`, sin `except`.

Se va a ver ahora como actúa Python al encontrarse con estos bloques. Python empieza a ejecutar las instrucciones dentro del bloque `try`. Si durante la ejecución de esas instrucciones se lanza una excepción, Python interrumpe la ejecución en el punto exacto donde surgió la excepción y pasa a ejecutar el contenido del bloque `except` correspondiente.

En este punto, Python verifica uno a uno los bloques `except` y si encuentra alguno que coincida con la referencia al tipo de excepción lanzada, lo ejecuta. Si no encuentra ningún bloque que se le corresponda pero hay un bloque `except` sin tipo definido, lo ejecuta. Al terminar de ejecutar el bloque correspondiente, se pasa a la ejecución del bloque `finally`, en caso de que se haya definido previamente.

Si no surge ningún problema durante la ejecución del bloque `try`, se completa la ejecución del bloque y se pasa directamente al bloque `finally`.

Se va a ver con un ejemplo como se va ejecutando el código en Python. Si se tiene el siguiente fragmento de código de un script que tiene que procesar cierta información introducida por el usuario y guardarla en un fichero externo. El acceso a archivos puede lanzar excepciones, con lo que siempre se debería de colocar el código de manipulación de archivos dentro de un bloque `try`. Posteriormente se debería de colocar un bloque `except` que atrape una excepción del tipo `IOerror`, que es del tipo de excepciones que lanzan las funciones de manipulación de archivos. Adicionalmente a esto, se podría agregar un bloque `except` sin tipo por si surge alguna otra excepción. Finalmente se debería agregar un bloque `finally` para cerrar el archivo, haya surgido o no una excepción.

```
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()
```

4. Invocación de excepciones

La declaración `raise` permite forzar que ocurra una excepción especifica indicando el identificador. Por ejemplo:

```
>>>raise NameError('Hola')
#Imprime:

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

El único argumento de `raise` indica la excepción a generarse. Puede ser o una instancia de excepción o una clase de excepción (heredada de `Exception`).

Se suele usar si se quiere determinar cuándo una excepción fue lanzada pero no se quiere manejarla. Una forma simplificada de la sentencia `raise` permite lanzarla.

```
try:
    raise NameError('Hola')
except NameError:
    print('Ha ocurrido una excepción!')
    raise

#Imprime:
Ha ocurrido una excepción!
Traceback (most recent call last):
  File "<input>", line 2, in <module>
NameError: Hola
```

5. Excepciones definidas por el usuario

El usuario puede crear sus propias excepciones creando una nueva clase heredada de la clase `Exception`. Se recomienda que deriven las nuevas excepciones a partir de las excepciones predefinidas o de la clase `Exception`, pero nunca de `BaseException`, ya que ésta no está pensaba para ser heredada directamente por las clases definidas por el usuario.

Por ejemplo:

```
class MiError(Exception):
    def __init__(self, valor):
        self.valor = valor
    def __str__(self):
        return repr(self.valor)

try:
    raise MiError(2*2)
except MiError as e:
    print('Ha ocurrido mi excepción, valor:', e.valor)
```

```
#Imprime:
Ha ocurrido mi excepción, valor: 4
```

```
raise MiError('oops!')
#Imprime:
Traceback (most recent call last):
  File "<input>", line 1, in <module>
MiError: 'oops!'
```

Se puede ver que el método `__init__()` de la clase `Exception` se ha sobrescrito. El nuevo comportamiento simplemente crea el atributo `valor`. Esto reemplaza el comportamiento de crear el atributo `args` que estaba por defecto.

Las clases que heredan de `Exception` se pueden definir de manera similar que cualquier otra clase, manteniéndolas simples y a menudo ofreciendo un numero de atributos con información del error que leerán los manejadores de la excepción. Se suele crear una clase base para excepciones definidas en ese módulo y extenderla para crear clases de excepciones específicas para las distintas condiciones de error. Por ejemplo:

```
class Error(Exception):
    """Clase base para excepciones en el módulo."""
    pass

class EntradaError(Error):
    """Exception lanzada por errores en las entradas.

    Atributos:
        expresion -- expresión de entrada en la que ocurre el error
        mensaje -- explicación del error
    """
```

```
    def __init__(self, expresion, mensaje):
        self.expresion = expresion
        self.mensaje = mensaje

class TransicionError(Error):
    """Lanzada cuando una operación intenta una
       transición de estado no permitida.

    Atributos:
        previo -- estado al principio de la transición
        siguiente -- nuevo estado intentado
        mensaje -- explicación de porque la transición no esta permitida
    """
    def __init__(self, previo, siguiente, mensaje):
        self.previo = previo
        self.siguiente = siguiente
        self.mensaje = mensaje
```

Por convención las excepciones se definen con nombres que terminan en «`Error`», similares a los nombres de las excepciones estándar.

6. Definir acciones de limpieza

La sentencia `try` tiene una sentencia opcional que intenta definir acciones de limpieza que deben ser ejecutadas bajo ciertas circunstancias. Esta sentencia opcional es **`finally`**, ya vista en apartados anteriores. El bloque dentro de `finally` siempre se ejecuta antes de salir de la sentencia `try`, tanto si ha ocurrido una excepción como si no. Cuando se produce una excepción en la sentencia `try` y no fue manejada por una sentencia `except`, se relanza después de que se ejecute el bloque `finally`. La sentencia `finally` se ejecuta también al final, cuando cualquier otra sentencia de try es detenida mediante `break`, `continue` o `return`.

Por ejemplo:

```
def dividir(x, y):
    try:
        resultado = x / y
    except ZeroDivisionError:
        print("¡división por cero!")
    else:
        print("el resultado es", resultado)
    finally:
        print("ejecutando la clausula finally")
>>> dividir(2,1)
```

```
# Imprime:
el resultado es 2.0
ejecutando la clausula finally
>>> dividir(2,0)
¡división por cero!
ejecutando la clausula finally
>>> dividir("2","0")
ejecutando la clausula finally
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 3, in dividir
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

La sentencia `finally` se suele usar para liberar recursos externos, como archivos o conexiones de red que pueden quedarse abiertas, sin importar si el uso del recurso fue exitoso.

7. Recursos

https://docs.python.org/es/3/library/exceptions.html#bltin-exceptions