# UF3.8 Excepciones en Python

Centro Profesional
Universidad Europea de Madrid
LAUREATE INTERNATIONAL UNIVERSITIES

# Contenidos

- Excepciones
- Manejo de Excepciones en Python
- Except
- Finally
- Ejecución de Excepciones en Python
- Propagación de Excepciones
- Acceso a información de contexto
- Excepciones de Python
- Validaciones

# Excepciones

Durante la ejecución de un programa, si dentro de una función surge una excepción y la función no la maneja, la excepción se propaga hacia la función que la invocó, si esta otra tampoco la maneja, la excepción continua propagándose hasta llegar a la función inicial del programa y si esta tampoco la maneja se interrumpe la ejecución del programa. Veamos entonces como manejar excepciones en Python.

Para el manejo de excepciones los lenguajes proveen ciertas palabras reservadas, que nos permiten manejar las excepciones que puedan surgir y tomar acciones de recuperación para evitar la interrupción del programa o, al menos, para realizar algunas acciones adicionales antes de interrumpir el programa.

# Manejo de Excepciones en Python

En el caso de Python, el manejo de excepciones se hace mediante los bloques que utilizan las sentencias **try**, **except** y **finally.**

Dentro del bloque **try** se ubica todo el código que pueda llegar a *levantar* una excepción, utilizamos el término *levantar* para referirnos a la acción de generar una excepción.

A continuación se ubica el bloque **except** que se encarga de capturar la excepción y nos da la oportunidad de procesarla mostrando por ejemplo un mensaje adecuado al usuario.

# Ejemplo

Si ejecutamos:

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

Obtendremos una excepción/error de tipo:

```
ZeroDivisionError: integer division or modulo by zero
```

Para controlar esta excepción deberíamos hacer algo así:

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

# Manejo de Excepciones en Python. except

Dado que dentro de un mismo bloque try pueden producirse excepciones de distinto tipo, es posible utilizar varios bloques **except**, cada uno para capturar un tipo distinto de excepción.

Esto se hace especificando **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, lo cual se hace especificando los nombres de la excepciones separados por comas a continuación de la palabra except.

Es importante destacar que aunque después de un bloque try haya varios bloques except, se ejecutará, a lo sumo, uno de ellos.

# Ejemplo

```
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
```

# Manejo de Excepciones en Python. finally

Después de los bloques try y except, puede ubicarse un bloque **finally** donde se escriben las sentencias de finalización, que son típicamente acciones de limpieza. La particularidad del bloque **finally es que se ejecuta siempre**, haya surgido una excepción o no.

Si hay un bloque except, no es necesario que esté presente el finally.

# Ejemplo

Supongamos que nuestro programa tiene que procesar un archivo. Dado que el acceso a archivos puede levantar excepciones de tipi **IOError**, siempre deberíamos colocar el código de manipulación de archivos dentro de un bloque try.

```
try:
    archivo = open("miarchivo.txt")
    # procesar el archivo
except IOError:
    print "Error de entrada/salida."
    # realizar procesamiento adicional
except:
    # procesar otra posible excepción
finally:
    # si el archivo no está cerrado hay que cerrarlo
    if not(archivo.closed):
        archivo.close()
```

# Ejecución de Excepciones en Python.

Python comienza a ejecutar las instrucciones que se encuentran dentro de un bloque try. Si durante la ejecución de esas instrucciones se levanta una excepción, Python interrumpe la ejecución en el punto exacto en que surgió la excepción y pasa a la ejecución del bloque except correspondiente.

Para ello, Python verifica uno a uno los bloques except y si encuentra alguno cuyo tipo haga referencia al tipo de excepción levantada, comienza a ejecutarlo. Si no encuentra ningún bloque del tipo correspondiente pero hay un bloque except sin tipo, lo ejecuta.

Al terminar de ejecutar el bloque correspondiente, se pasa a la ejecución del bloque finally, si se encuentra definido.

Si, por otra parte, no hay problemas durante la ejecución del bloque try, se completa la ejecución del bloque, y luego se pasa directamente a la ejecución del bloque finally (si es que está definido).

# Propagación de Excepciones

Es posible, además, que después de realizar algún procesamiento particular de una excepción, se quiera que se propague hacia la función que había invocado a la función en la que se ha producido. Para hacer esto Python nos ofrece la instrucción **raise**.

Si se invoca esta instrucción dentro de un bloque except, sin pasarle parámetros, Python propagará la excepción atrapada por ese bloque.

Nos podría interesar que, en lugar de propagar la excepción tal cual fue atrapada, quisiéramos lanzar una excepción más significativa para quien invocó a la función actual y que posiblemente contenga cierta información de contexto. Para levantar una excepción de cualquier tipo, utilizamos también la sentencia **raise**, pero indicándole el tipo de excepción que deseamos lanzar y pasando a la excepción los parámetros con información adicional que queramos facilitar.

# Ejemplo

```
def dividir(dividendo, divisor):
    try:
        resultado = dividendo / divisor
        return resultado
    except ZeroDivisionError:
        raise ZeroDivisionError("El divisor no puede ser cero")

def main():
    try:
        dividendo = 5
        divisor = 0
        dividir(dividendo, divisor)
    except ZeroDivisionError as ex:
        print "Ha ocurrido una excepción: “ , ex

main()
```

# Acceso a información de contexto

Para acceder a la información de contexto estando dentro de un bloque except existen dos alternativas.

Se puede utilizar la función **exc_info del módulo sys**. Esta función devuelve una tupla con información sobre la última excepción atrapada en un bloque except. Dicha tupla contiene tres elementos: el tipo de excepción, el valor de la excepción y las llamadas realizadas.

Otra forma de obtener información sobre la excepción es utilizando la misma sentencia except, pasándole un identificador para que almacene una referencia a la excepción atrapada.

# Ejemplo

```
def main():
    try:
        dividendo = 5
        divisor = 0
        dividir(dividendo, divisor)
    except ZeroDivisionError as ex:
        print "Ha ocurrido una excepción: “ , ex
        print sys.exc_info()
        print type(ex)
        print ex.args
```

# Acceso a información de contexto

Para acceder a la información de contexto estando dentro de un bloque except existen dos alternativas.

Se puede utilizar la función **exc_info del módulo sys**. Esta función devuelve una tupla con información sobre la última excepción atrapada en un bloque except. Dicha tupla contiene tres elementos: el tipo de excepción, el valor de la excepción y las llamadas realizadas.

Otra forma de obtener información sobre la excepción es utilizando la misma sentencia except, pasándole un identificador para que almacene una referencia a la excepción atrapada.

# Excepciones de Python

A continuación se listan las excepciones disponibles por defecto, así como la clase de la que deriva cada una de ellas entre paréntesis.

- **BaseException**: Clase de la que heredan todas las excepciones.
- **Exception**(BaseException): Super clase de todas las excepciones que no sean de salida.
- GeneratorExit(Exception): Se pide que se salga de un generador.
- StandarError(Exception): Clase base para todas las excepciones que no tengan que ver con salir del intérprete.
- **ArithmeticError**(StandardError): Clase base para los errores aritméticos.
- FloatingPointError(ArithmeticError): Error en una operación de coma flotante.
- OverflowError(ArithmeticError): Resultado demasiado grande para poder representarse.

# Excepciones de Python

- **ZeroDivisionError**(ArithmeticError): Lanzada cuando el segundo argumento de una operación de división o módulo era 0
- **AssertionError**(StandardError): Falló la condición de un estamento assert.
- **AttributeError**(StandardError): No se encontró el atributo.
- **EOFError**(StandardError): Se intentó leer más allá del final de fichero.
- EnvironmentError(StandardError): Clase padre de los errores relacionados con la entrada/salida.
- **IOError**(EnvironmentError): Error en una operación de entrada/salida.
- OSError(EnvironmentError): Error en una llamada a sistema.
- WindowsError(OSError): Error en una llamada a sistema en Windows.
- **ImportError**(StandardError): No se encuentra el módulo o el elemento del módulo que se quería importar.

# Excepciones de Python

- LookupError(StandardError): Clase padre de los errores de acceso.
- **IndexError**(LookupError): El índice de la secuencia está fuera del rango posible.
- KeyError(LookupError): La clave no existe.
- MemoryError(StandardError): No queda memoria suficiente.
- **NameError**(StandardError): No se encontró ningún elemento con ese nombre.
- UnboundLocalError(NameError): El nombre no está asociado a ninguna variable.
- ReferenceError(StandardError): El objeto no tiene ninguna referencia fuerte apuntando hacia él.
- **RuntimeError**(StandardError): Error en tiempo de ejecución no especificado.

# Excepciones de Python

- NotImplementedError(RuntimeError): Ese método o función no está implementado.
- SyntaxError(StandardError): Clase padre para los errores sintácticos.
- IndentationError(SyntaxError): Error en la indentación del archivo.
- TabError(IndentationError): Error debido a la mezcla de espacios y tabuladores.
- SystemError(StandardError): Error interno del intérprete.
- **TypeError**(StandardError): Tipo de argumento no apropiado.
- **ValueError**(StandardError): Valor del argumento no apropiado.

# Excepciones de Python

- UnicodeError(ValueError): Clase padre para los errores relacionados con unicode.
- UnicodeDecodeError(UnicodeError): Error de decodificación unicode.
- UnicodeEncodeError(UnicodeError): Error de codificación unicode.
- UnicodeTranslateError(UnicodeError): Error de traducción unicode.
- StopIteration(Exception): Se utiliza para indicar el final del iterador.
- Warning(Exception): Clase padre para los avisos.
- DeprecationWarning(Warning): Clase padre para avisos sobre características obsoletas.
- FutureWarning(Warning): Aviso. La semántica de la construcción cambiará en un futuro.

# Excepciones de Python

- ImportWarning(Warning): Aviso sobre posibles errores a la hora de importar.
- PendingDeprecationWarning(Warning): Aviso sobre características que se marcarán como obsoletas en un futuro próximo.
- RuntimeWarning(Warning): Aviso sobre comportmaientos dudosos en tiempo de ejecución.
- SyntaxWarning(Warning): Aviso sobre sintaxis dudosa.
- UnicodeWarning(Warning): Aviso sobre problemas relacionados con Unicode, sobre todo con problemas de conversión.
- UserWarning(Warning): Clase padre para avisos creados por el programador.
- KeyboardInterrupt(BaseException): El programa fué interrumpido por el usuario.
- SystemExit(BaseException): Petición del intérprete para terminar la ejecución.

# Validaciones

Existe una manera en Python de levantar una excepción en función de la validación de un dato. Es utilizando la palabra clave **assert** de la siguiente manera:

```
try:
    assert condicion
except AssertionError:
```

Si no se cumple la condición especificada se produce un AssertionError.

# Bibliografía y Webgrafía


http://librosweb.es/libro/algoritmos_python/

http://docs.python.org.ar/tutorial/3/errors.html