Excepciones en Python

Los errores detectados durante la ejecución de un programa se llaman excepciones. Durante la ejecución de nuestra aplicación, muchas cosas pueden salir mal. Un disco puede llenarse y no podemos salvar a nuestro archivo. Una conexión a internet puede caerse y aplicación no puede conectarse a un sitio. Todo esto podría resultar en un accidente de nuestra aplicación. Para evitar que suceda esto, tenemos que hacer frente a todos los posibles errores que pudieran ocurrir. Para ello, podemos utilizar el control de excepciones.

Python utiliza las excepciones para comunicar errores y anomalías. Una excepción es un objeto que indica un error o una condición anómala. Cuando Python detecta un error provoca una excepción, es decir, Python señala la ocurrencia de una condición anómala transmitiendo un objeto de excepción al mecanismo propagación de excepción. Nuestro código puede de manera explícita lanzar una excepción ejecutando una sentencia raise.

```
In [1]:  4/0
---------------------------------------------------
ZeroDivisionError           Traceback (most recent call last)
<ipython-input-2-6de94738d89d> in <module>()
----> 1 4/0

ZeroDivisionError: integer division or modulo by zero
```

No es posible dividir por cero. Si tratamos de hacer esto, una llamada ZeroDivisionError se eleva y la secuencia de comandos se interrumpe.

```
#!/usr/bin/python

def input_numbers():
    a = float(raw_input("Ingresa un primer numero:"))
    b = float(raw_input("Ingresa un segundo numero:"))
    return a, b

x, y = input_numbers()
print "%d / %d is %f" % (x, y, x/y)
```

En este script, se obtienen dos números de consola. Dividimos estos dos números. Si el segundo número es cero, obtenemos una excepción.

```
%edit exception0.py
Editing... done. Executing edited code...
Ingresa el primer numero:4
Ingresa el segundo numero:0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/usr/lib/python2.7/dist-packages/IPython/utils/py3compat.pyc in execfile(fname, *where)
    176             else:
    177                 filename = fname
--> 178             __builtin__.execfile(filename, *where)

/home/cesarlaraavila/Python-ACECOM-GISCIA/exception0.py in <module>()
      7
```

```
      8 x, y = input_numbers()
----> 9 print "%d / %d is %f"%(x, y, x/y)

ZeroDivisionError: float division by zero
```

La gestión de una excepción implica recibir el objeto de excepción desde el mecanismo de propagación y realizar todas las acciones necesarias para abordar la situación irregular. Si un programa no controla la excepción el programa finaliza con un mensaje traceback (informe de seguimiento o de rastreo) de error. Pero el programa puede gestionar las excepciones y continuar ejecutándose.

Nosotros podemos manejar esto de dos maneras:

```
#!/usr/bin/python

def input_numbers():
    a = float(raw_input("Ingresa el primer numero:"))
    b = float(raw_input("Ingresa el segundo numero:"))
    return a, b


x, y = input_numbers()

while True:
    if y != 0:
        print "%d / %d is %f" % (x, y, x/y)
        break
    else:
       print "No se puede dividir por cero"
       x, y = input_numbers()
```

En primer lugar, simplemente chequeamos que si el valor de **y** no es cero. Si el valor de y es igual a cero, podemos imprimir un mensaje de advertencia y repetimos el ciclo de entrada de nuevo. De esta manera, manejamos el error y el script no se interrumpe.

```
%edit exception0A.py
Editing... done. Executing edited code...
Ingresa el primer numero:5
Ingresa el segundo numero:0
No se puede dividir por cero
Ingresa el primer numero:4
Ingresa el segundo numero:9
4 /9 is 0.444444
```

La otra manera es hacerlo con excepciones...

```
#!/usr/bin/python

def input_numbers():
    a = float(raw_input("Ingresa un numero:"))
    b = float(raw_input("Ingresa un segundo numero:"))
    return a, b
```

```
x, y = input_numbers()

while True:
    try:
        print "%d / %d is %f" % (x, y, x/y)
        break
    except ZeroDivisionError:
       print "No puedes dividir por cero"
       x, y = input_numbers()
```

Después de la palabra clave try, ponemos el código, donde nos espera una excepción. La palabra clave except detecta la excepción, si es invocado. Se especifica, que tipo de excepción estamos buscando.

La sentencia try proporciona el mecanismo de gestión de excepciones a Python. Es una sentencia compuesta que puede tener una o dos formas diferentes.

- Una cláusula try seguida por una o más cláusulas except y opcionalmente una cláusula else.
- Una cláusula try seguida por cláusulas finally.

La palabra clave finally siempre se ejecuta. No importa si la excepción se eleva o no. A menudo se utiliza para hacer un poco de limpieza de recursos en un programa. Por ejemplo

```
#!/usr/bin/python

f = None

try:
   f = file('file1.txt', 'r')
   contents = f.readlines()
   for i in contents:
      print i,
except IOError:
   print 'Error al abrir el archivo'
finally:
   if f:
      f.close()
```

En nuestro ejemplo, se intenta abrir un archivo. Si no podemos abrir el archivo, un IOError se eleva. En caso de que abrimos el archivo, queremos cerrar el controlador de archivo. Para ello, utilizamos la palabra clave finally. En el bloque finally, comprobamos si el archivo se abre o no. Si se abre, lo cerramos.

Se trata de una construcción de programación común cuando se trabaja con bases de datos. Es similar la limpieza de las conexiones de base de datos abiertos.

1. Objetos de excepción

Las excepciones son instancias de la clase de herencia integrada Exception clase BaseException salvo para algunas clases como keyboardInterrupt y SystemExit en Python 2.5 que no heredan de Exception sino de la subclase de esta llamada BaseException. La estructura de herencia de las clases de excepción es importante, ya que determina que cláusulas de except gestionan que excepciones. Los errores de runtime más habituales producen excepciones de las siguientes clases:

- AssertionError: Una sentencia assert fallida.
- AttributeError: Una referencia de atributo o assignación fallida.
- FloatingPointError: Una operación en punto flotante fallida. Derivado desde ArithmeticError que es la clase base para excepciones debido a errores aritméticos (es decir OverflowError, ZeroDivisionError, FloatingPointError)
- IOError: Una operación I/O (por ejemplo, no se encontró un archivo o el permiso requerido estaba perdido). Derivado de EnvironmentError la clase base para excepciones producidas por causas externas (es decir IOError, OSError y WindowsError).
- ImportError: Una sentencia **import** no puede encontrar el módulo para importar o no puede encontrar un nombre solicitado desde el módulo.
- IndentationError: El analizador gramatical encontró un error de sintaxis debido a la indentación de un sangrado incorrecto.
- IndexError: Un número entero utilizado para indexar una secuencia está fuera del rango. Derivado de LookupError, la clase base para excepciones que provoca un contenedor cuando recibe una clave o índice no válido.
- KeyError: Una clave utilizada para indexar una correspondencia que no está en la correspondencia.
- KeyboardInterrupt: El usuario pulsa la tecla de interrupción.
- MemoryError: Una operación se quedó sin memoria.
- NameError: Se hizo referencia a una variable, pero su nombre no estaba vinculado.
- NotImplementedError: Provocado por las clases de bases abstractas para indicar que una determinada subclase debe invalidar un método.
- OSError: Producido por funciones en el módulo `os` para indicar errores dependientes de la plataforma. Derivado de EnvironmentError.
- OverflowError: El resultado de una operación en un número entero es demasiado grande para ajustarse a un número entero (el operador ¡¡ no provoca esta excepción, sino suelta el exceso de bits). Los resultados de números enteros demasiados grandes se convierten en números largos evitando esta excepción.
- SyntaxError: El analizador gramatical encontró un error de sintaxis.
- SystemError: Un error interno en el mismo Python o en algún módulo de extensión.
- TypeError: Se aplicó una operación o función a un objeto de un tipo adecuado.
- UnboundLocalError: Se hizo referencia a una variable local, pero ningún valor está vinculado en ese momento a la variable local. Derivado de NameError.
- UnicodeError: Tuvo lugar un error cuando se convirtió Unicode en cadena o viceversa.
- ValueError: Se aplicó una operación o función a un objeto que tiene el tipo correcto pero un valor inadecuado y no se utiliza nada más específico.
- WindowsError: Provocado por funciones en el módulo `os` para indicar errores específicos de Windows. Derivado de OsError.
- ZeroDivisionError: Un divisor (el operador derecho de un operador /, //) o el segundo de una función integrada `divmod` es 0.

En resumen las excepciones se organizan en jerarquias , siendo la clase Exception

```
#!/usr/bin/python

try:
    while True:
       pass
except KeyboardInterrupt:
   print "Programa interrumpido"
```

El script se inicia y el ciclo sigue sin fin. Si pulsamos Ctrl + C, interrumpimos el ciclo. Aquí, atrapamos la excepción KeyboardInterrupt. Esta es la jerarquía de KeyboardInterrupt

```
Exception
  BaseException
    KeyboardInterrupt
```

Este ejemplo trabaja también, la clase BaseException también atrapa la interrupción del teclado, entre otras excepciones.

```
#!/usr/bin/python

try:
    while True:
       pass
except BaseException:
   print "Programa interrumpido"
```

except tiene un segundo argumento obtenemos una referencia al objeto de la excepción.

```
#!/usr/bin/python

try:
    6/0
except ZeroDivisionError, e:
    print "No se puede dividir por cero"
    print "Mensaje:", e.message
    print "Class:", e.__class__
```

Desde el objeto de excepción, se puede obtener el mensaje de error y/o el nombre de la clase.

```
%edit exception2.py
Editing... done. Executing edited code...
No se puede dividir por cero
Mensaje: integer division or modulo by zero
Class: <type 'exceptions.ZeroDivisionError'>
```

2.Nuestras propias excepciones

Podemos crear nuestras propias excepciones, si queremos. Lo hacemos mediante la definición de una nueva clase de excepción.

```
#!/usr/bin/python

class B_Error(Exception):
   def __init__(self, value):
      print "B_Error: B caracter encontrado en la posicion %d" % value

cadena  = "Aprender a programar en Python Basico"

pos = 0
for i in cadena:
   if i == 'B':
      raise BFoundError, pos
   pos = pos + 1
```

En nuestro ejemplo , hemos creado una nueva excepción. La excepción se deriva de la clase Exception, es decir si encontramos la letra B en la cadena, hacemos un raise es decir elevamos nuestra excepción.

```
%edit exception5.py
Editing... done. Executing edited code...
B_Error:B caracter encontrado en la posicion 32
---------------------------------------------------------------------------
B_Error                                   Traceback (most recent call last)
/usr/lib/python2.7/dist-packages/IPython/utils/py3compat.pyc in execfile(fname, *where)
    176             else:
    177                 filename = fname
--> 178             __builtin__.execfile(filename, *where)

/home/cesarlaraavila/Python-ACECOM-GISCIA/exception5.py in <module>()
     10 for i in cadena:
     11     if i == 'B':
---> 12         raise B_Error, pos
     13     pos = pos + 1

B_Error:
```

3. Otras excepciones utilizadas de la biblioteca estándar

Muchos módulos en la biblioteca estándar definen sus propias clases de excepción, que son equivalentes a las clases de excepción personalizadas que nuestros propios módulos pueden definir. Habitualmente, todas las funciones en los módulos de la biblioteca estándar mencionados pueden provocar excepciones de dichas clases, además de excepciones de la jerarquía estándar.

El módulo `socket` proporciona la clase socket.error, que se obtiene de la clase integrada Exception y de varias subclases de error llamadas timeout, gaierror, herror; todas las funciones y métodos en el módulo `socket`, aparte de la excepciones estándar, pueden provocar excepciones de clase socket.error de lo mismo. Por ejemplo

```
#!/usr/bin/python

import socket
name = "wwaxww.python.org"
try:
    host = socket.gethostbyname(name)
    print host
```

```
except socket.gaierror, err:
    print "No se encuentra el servidor : ", name, err
```

Lo cúal produce el siguiente resultado en Ipython

```
%edit exception7.py
Editing... done. Executing edited code...
No se encuentra el servidor: wwwax2.python.org [Errno -2] Name or service not known
```

Python utiliza también las excepciones para indicar algunas situaciones especiales que no son errores. Por ejemplo el método next de un iterador provoca la excepción StopIteration cuando el iterador no tiene más elementos. Esto no es un error y no es una anomalía aún , puesto que la mayoría de iteradores con el tiempo ejecutan sin elementos.

```
def una_funcion():
    for i in xrange(4):
        yield i

for i in una_funcion():
    print i
```

```
%edit generatorA.py
Editing... done. Executing edited code...
0
1
2
3
```

Esto es básicamente lo que hace el intérprete de python con el código anterior:

```
#!/usr/bin/python

class it:
    def __init__(self):
        # Empezamos en -1 tal que conseguimos 0 cuando agregamos 1
        self.count = -1
    # El metodo __iter__ es llamado para el bucle for
    # todo ocurre sobre el objeto que devuelve este metodo
    # en este caso es el objeto en si mismo

    def __iter__(self):
        return self

    # El metodo next es llamado repetidamente por el bucle for
    # hasta que el aparezca StopIteration

    def next(self):
        self.count += 1
        if self.count < 4:
            return self.count
        else:
            # StopIteration es lanzada (raise)
```

```
            raise StopIteration

def una_funcion():
    return it()

for i in una_funcion():
    print i
```

Ejecutando esto tenemos el mismo resultado

```
%edit exception-generatorA.py
Editing... done. Executing edited code...
0
1
2
3
```

Las estrategias fundamentales para comprobar y gestionar errores y otras situaciones especiales en Python son por lo tanto diferentes de lo que podría ser mejor en otros lenguajes.