Curso Básico de Python

8. Errores y Excepciones

Contenido

Los errores en programación pueden ser de diferentes categorías, desde una orden mal escrita hasta una implemetación de solución inadecuada para el problema en cuestión. El intérprete de Python puede deectar algunas de estas categorías de error, y lo hace mediante un mecanismo denominado *manejo de excepciones* (*Exception handling*). Este capítulo describe brevemente los distintos tipos de errores de programación, y explica el manejo de excepciones en Python.

Lecturas recomendadas:

1. Errores y excepciones en el Tutorial de Python.

Referencias para consulta:

1. Python Docs. La biblioteca estándar de Python, Excepciones incorporadas. Describe las excepciones estándar de Python. Las excepciones ya definidas en Python conforman una Jerarquía de excepciones.

1 Errores de programación

En programación, se reconocen (al menos) tres categorías de errores:

- **Errores de sintaxis.** Hay algo mal escrito en el código, que no cumple las reglas de escritura de Python y en consecuencia el intérprete no puede procesar. El intérprete emite un mensaje de error señalando el punto donde se detectó ese error, pero el punto exacto puede estar antes de donde aparece la indicación. Estos errores son muy comunes; son también los más fáciles de corregir.
- **Errores lógicos.** La sintaxis es correcta, pero hay un error en el orden de las sentencias o en la forma en que se relacionan unas con otras. Un error lógico sería escribir en un archivo, olvidar cerrarlo, e intentar abrirlo para leer.
- **Errores de diseño.** La sintaxis está bien, las sentencias están bien coordinadas, pero las órdenes que se están dando no resuelven correctamente el problema planteado. Un algoritmo inadecuado es un error de diseño. El programa corre perfectamente, pero el resultado no es el esperado, no resuelve bien el problema planteado.

En programación, el proceso de buscar y encontrar errores en los programas se denomina *depuración* (*debugging)*.

2 Excepciones

Cuando el intérprete de comandos detecta un error, construye una estructura de datos (un objeto) denominado una *excepción*, donde informa los detalles del error detectado, y termina la ejecución. No obstante, la excepción puede ser "capturada", permitiendo al programador definir acciones en caso de ocurrir un error, y evitar así la terminación abrupta del programa. Python tiene un conjunto de excepciones ya definidas, correspondientes a distintos tipos de error. La sentencia `try...except` permite capturar una excepción: se coloca el código donde puede ocurrir el error después de la cláusula `try`, y el código para tratar distintos tipos de excepción en una o varias cláusulas `except`. En formato simplificada, la sintaxis de `try ... except` es la siguiente:

```
try:              ### sintaxis de try...except
    bloque        # sentencias donde pueden ocurrir una o varias excepciones
except [expresión [as e_var]]:   # excepciones y su captura en e_var
    bloque        # sentencias a ejecutar cuando se captura la excepción
...               # otras cláusulas except
[else:            # ejecuta si no se produjo ninguna excepción
    bloque ]
[finally:         # ejecuta si no hubo excepción o hubo y se trató
    bloque]
```

La *`expresión`* puede incluir una o varias excepciones; si no se indica ninguna, cualquier tipo de excepción será capturada. Es posible guardar la excepción en una variable *`e_var`*, que puede examinarse con `repr(e_var)`.
Las cláusulas `else` y `finally` son opcionales, tal como indican los paréntesis rectos. El bloque `finally` suele usarse para operaciones de terminación y limpieza (*cleanup*), como cerrar archivos abiertos o liberar estructuras ya no usadas para recuperar memoria.

El siguiente código captura y muestra una excepción cualquiera, de la clase `Exception`.

```
>>> for div in [1, 0]:
    try:
        print("   div =", div, ", 1/div =", 1/div)
    except Exception as e:
        print("   div =", div, ", 1/div = ERROR")
        repr(e)

   div = 1 , 1/div = 1.0                  # ejecutó sin error
   div = 0 , 1/div = ERROR                # error capturado en "except"
"ZeroDivisionError('division by zero')"    # salida de "repr(e)"
>>>
```

El siguiente código captura dos excepciones específicas, y la excepción genérica al final.

```
for div in [1, 0, 3, 4]:
    try:
        if div == 3:
            print(no_var)   # no_var no existe
        elif div == 4:
            ls = []
            print(ls[1])    # índice fuera de rango
        else:
            print("   div =", div, ", 1/div =", 1/div)
    except ZeroDivisionError as e:
        print("   div =", div, ", 1/div = ERROR")
        repr(e)
    except NameError as e:
        print("   variable no_var no existe, ERROR")
        repr(e)
    except Exception as e:
        print("   otra excepción")
        repr(e)
```

```
   div = 1 , 1/div = 1.0
   div = 0 , 1/div = ERROR
"ZeroDivisionError('division by zero')"
   variable no_var no existe, ERROR
'NameError("name \'no_var\' is not defined")'
   otra excepción
"IndexError('list index out of range')"
>>>
```

El siguiente programa muestra la captura de varios tipos de excepciones, así como el uso de las cláusula `else` y `finally` (adaptado de PEW's Corner. Python 3 Quick Reference).

```
'''Detección y tratamiento de excepciones.

Define una lista de expresiones, recorre la lista evaluando cada una,
levanta una excepción según el ressultado; incluye cláusulas opcionales
"else" y "finally", emite mensajes para mostrar el flujo de ejecución.
'''

import sys

lst_expr = (
    '1/0',          # división por 0
    '[][1]',        # índice más allá de límite de la lista, aquí vacía
    '{}[2]',        # clave inválida en un diccionario, aquí vacío
    '(3).a',        # atributo 'a' inexistente en el tipo entero
    ')',            # error de sintaxis, paréntesis sin pareja
    'sys.exit(5)',  # salida del sistema, termina la ejecución
    '6'             # número entero, no levanta excepción
    )

for expr in lst_expr:
    try:
        eval(expr)
        # evalua la expresión, eventualmente levantando excepciones
        # continúa en este bloque si no hubo excepciones
        print('      -- no hubo excepciones')
    except ZeroDivisionError:   # no guarda la excepción
        # sigue aquí si hubo excepción ZeroDivisionError en el bloque "try"
        print('      -- división por 0')
    except (IndexError, KeyError, AttributeError) as e:
        # sigue aquí si en el bloque "try" de dio alguna de esas excepciones
        # retiene la excepción en la variable "e"
        print('      -- ', repr(e))
    except Exception as e:
        # sigue aquí con cualquier excepción de clase Exception
        # no tratada antes
        print('      -- captura excepción genérica de tipo "Exception":')
        print('        ', repr(e))
        print('         -- "continue" salta el resto del "for"')
        continue
        # saltea el resto del "for" después de ejecutar "finally",
        #     continúa con el siguiente item en la lista del "for"
    except SystemExit as e:
        # sique aquí con la excepción SystemExit,
        #    que es subclase de BaseException, no de Exception
        print('      -- excepción de salida, subclase de "BaseException"')
        print('      -- ', repr(e))
    else:
        # opcional; sigue aquí si no hubo excepciones en "try",
        #    es lo mismo que agregar código al final del "try",
        #    pero si aparecen excepciones aquí no se tratan
        print('      -- "else", no hubo excepciones en try, todo OK')
```

```
        print('         -- "break" sale del "for"')
        break
        # sale del for después de ejecutar "finally",
        #     no continúa trata los restantes items en la lista del "for"
    finally:
        # opcional; se ejecuta aún si hubo excepciones no tratadas o
        # sentencias return/break/continue en las partes try/except/else
        print('      -- "finally", ejecuta el bloque final')
    print('   -- ejecuta el resto del bloque repetitivo "for"')
```

La sentencia `try ... finally` sin cláusula `except` se usa para detectar una excepción a tratar por otra sentencia `try ... except` exterior.

```
try:              ### sintaxis de try...finally
    bloque        # bloque de sentencias
finally:          # ejecuta si hubo excepción y no se trató
    bloque        # bloque de sentencias
```

El siguiente programa levanta una excepción en un bloque `try...finally` interno y la trata en un bloque `try...except` externo (adaptado de PEW's Corner. Python 3 Quick Reference).

```
'''Levanta excepciones en "try ... finally".

El bloque interno "try...finally" levanta una excepción pero no la trata, y
es capturada por un bloque "try...except" externo.
'''

import sys

# este "try" exterior captura todas las excepciones no tratadas
#    en el "try" interior
try:
    # el "try" interior, solo con la parte "finally"
    try:
        print('--- en "try" interno, levanta una excepción con sys.exit()')
        sys.exit()
        # levanta la excepción SystemExit, pero no la trata
        print('--- esto no llega a ejecutarse nunca por la excepción')
    finally:
    # este bloque se ejecuta aunque haya habido excepción y no se trate
        print('--- en "finally" del "try" interno')
except:
    # captura la excepción SystemExit ocurrida en el "try" interno.
    # el "except" no indica tipo de excepción, captura todos los tipos;
    # "except Exception" captura solo las de tipo "Exception",
    # "SystemExit" es de tipo "BaseException", no "Exception".
    print('--- en "except" del "try" eexterior')
```

3 Levantar excepciones

Es posible levantar una excepción mediante el comando `raise`. Los siguientes ejemplos valen para cualquiera de las excepciones estándar definidas en Python.

Levanta una excepción sin indicar argumentos:

```
>>> raise SyntaxError      # equivale a raise SyntaxError()
Traceback (most recent call last):
  File "/usr/lib/python3.8/idlelib/run.py", line 559, in runcode
    exec(code, self.locals)
  File "<pyshell#24>", line 1, in <module>
SyntaxError: None
```

Levanta una excepción con un argumento propio del usuario:

```
>>> raise SyntaxError("excepción levantada")
Traceback (most recent call last):
  File "/usr/lib/python3.8/idlelib/run.py", line 559, in runcode
    exec(code, self.locals)
  File "<pyshell#23>", line 1, in <module>
SyntaxError: excepción levantada
```

Levanta y captura una excepción:

```
>>> try:
    raise SyntaxError("excepción levantada")
except Exception as e:
    print(repr(e))
SyntaxError('excepción levantada')
```

Es posible crear nuevas excepciones, definiendo una subclase que herede de la clase estándar de Python `Exception`. La herencia se explica en el capítulo sobre Clases y objetos.

4 Ejercicios

1. Estudiar y ejecutar el programa `try_except.py` en los ejemplos correspondientes a este capítulo. Comparar cuidadosamente el código con la salida impresa del programa, para comprender bien el flujo de ejecución según van ocurriendo las excepciones.

2. Estudiar y ejecutar el programa `try_finally.py` en los ejemplos correspondientes a este capítulo. Estudiar los comentarios y comparar con la salida del programa.

3. Revisar la Jerarquía de excepciones definidas en Python.
   - ¿De qué tipo son las excepciones `ImportError`, `KeyboardInterrupt`, `ZeroDivisionError`? Ejemplo: `Warning` es de tipo `Exception`.
   - Identificar de qué tipo son las excepciones tratadas en el programa `try_except.py`.
   - Verificar que la excepción `SystemExit` que levanta la sentencia `sys.exit()` no es de tipo `Exception` sino `BaseException`.

4. Estudiar y ejecutar el programa `raise.py` en los ejemplos, donde se muestran diversas formas de lanzar excepciones. Una excepción puede volver a levantarse; la cláusula `from` en la sentencia `raise` permite indicar a partir de qué otra excepción se levanta la actual.