Manejo de excepciones en Python

Mario González
Facultad de Ingeniería y Ciencias Aplicadas
Universidad de las Américas, Quito, Ecuador

Agosto, 2021

Manejo de excepciones en Python

- Python proporciona dos características muy importantes para manejar cualquier error inesperado en un programa y añadir capacidades de depuración en ellos:
- Manejo de excepciones (Exception Handling).
- Aserciones (Assertions).

Aserciones en Python

- Una aserción es una comprobación de validez que se puede activar o desactivar cuando haya terminado de probar el programa.
- La manera más fácil de pensar en una aserción es compararlo con un `raise-if` (o para ser más exactos, un `raise-if not`).
- Cuando una expresión es probada, si da como resultado `False`, se produce una excepción.
- Los programadores a menudo usan las aserciones al inicio de una función para comprobar si hay una entrada válida, y después de la llamada a una función para comprobar la salida devuelta.

La sentencia `assert`

- Cuando Python encuentra una sentencia `assert`, Python evalúa la expresión que acompaña, esperando que sea cierta. Si la expresión es falsa, Python devuelve una excepción de tipo `AssertionError`.
- La sintaxis es: **`assert Expression[, Arguments]`**
- Ejemplo:

```
def KelvinToFahrenheit(Temperatura):
    assert (Temperatura >= 0),"Mas frio que el cero absoluto!"
    return ((Temperatura-273)*1.8)+32

print(KelvinToFahrenheit(273))

print(KelvinToFahrenheit(-5))
```

Qué es una excepción

- Una excepción es un evento que ocurre durante la ejecución de un programa normal y que interrumpe el flujo de las instrucciones del programa.
- En general, cuando un script Python se encuentra con una condición a la que no puede hacer frente, se lanza una excepción (raise an exception).
- Una excepción es un objeto de Python que representa un error.
- Cuando un script Python lanza una excepción, esta debe ser manejada inmediatamente en caso contrario el el programa termina y se cierra.

  Manejo de una excepción Si usted tiene algún código sospechoso de que pueda plantear una excepción, puede defender su programa colocando el código sospechoso en un **`try: bloque`**. Después `try: bloque`, incluya un **`except: declaracion`**, seguido de un bloque de código que maneje el problema lo más elegante posible.

Manejo de excepciones I

- Sintaxis `try....except...else blocks`:

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Manejo de excepciones II

- Una sola sentencia `try` puede tener múltiples declaraciones de excepción (`except`).
- Esto es útil cuando el bloque `try` contiene declaraciones que pueden lanzar diferentes tipos de excepciones.
- También es posible proporcionar una cláusula `except` genérica, que maneje cualquier excepción.
- Después de las cláusulas `except`, se puede incluir una cláusula `else`.
- Si el código en el `try` no devuelve ninguna excepción, se ejecuta el bloque de la cláusula `else`.
- El bloque `else` es un buen lugar para el código que no necesita la protección `try: bloque`

Manejo de excepciones III

Ejemplos:

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print("Error: can\'t find file or read data")
else:
   print("Written content in the file successfully")
   fh.close()
```

Manejo de excepciones IV

```
try:
    fh = open("testfile", "r")
    fh.write("This is my test file for exception handling!!")
except IOError:
    print("Error: can\'t find file or read data")
else:
    print("Written content in the file successfully")
```

Cláusula `except` con ninguna excepción

- Podemos utilizar la sentencia `except` sin definir ninguna excepción:

```
try:
    You do your operations here;
    ......................
except:
    If there is any exception, then execute this block.
    ......................
else:
    If there is no exception then execute this block.
```

- Este tipo de sentencias `try-except` captura todas las excepciones que ocurren.
- Esto no es considerado una buena práctica de programación, dado que no permite al programador identificar la raíz del problema que pueda ocurrir.

Cláusula `except` con múltiples excepciones

- También podemos usar la sentencia `except` para manejar múltiples excepciones:

```
try:
    You do your operations here;
    ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
    If there is any exception from the given exception list,
    then execute this block.
    ......................
else:
    If there is no exception then execute this block.
```

Cláusula `try-finally` I

- Es posible utilizar `finally: bloque` junto con un `try: bloque`.
- El código en el bloque `finally` será ejecutado si el bloque `try` devuelve una excepción o no.

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

Cláusula `try-finally` II

Ejemplo:

```
try:
    fh = open("testfile", "r")
    try:
        fh.write("This is my test file for exception handling!!"
    finally:
        print("Going to close the file")
        fh.close()
except IOError:
    print("Error: can\'t find file or read/write data")
```

- Cuando se produce una excepción en el bloque `try`, la ejecución inmediatamente pasa al bloque `finally`. Después de todas las declaraciones en el bloque `finally` se ejecutan, la excepción se levanta de nuevo y se maneja en las sentencias `except` si está presente en la siguiente capa más alta de la sentencia `try-except`.

Argumentos de una excepción

- Una excepción puede tener un argumento, que es un valor que proporciona información adicional sobre el problema. El contenido del argumento varía según la excepción.
- El argumento de una excepción se captura mediante el suministro de una variable en la cláusula `except` de la siguiente manera:

```
# Define a function here.
def temp_convert(var):
   try:
      return int(var)
   except ValueError as Argumento:
      print("Error:", Argumento)

# Call above function here.
temp_convert("xyz");
```

Levantando excepciones (Raising exceptions) I

- Es posible levantar excepciones en varias formas mediante la instrucción `raise`. La sintaxis general para la sentencia `raise` es la siguiente:

```
raise [Exception [, args [, traceback]]]
```

- En este caso, la excepción es el tipo de excepción (por ejemplo, **`ValueError`**) y el argumento es un valor para el argumento de la excepción.

  El argumento es opcional, si no se proporciona, el argumento de excepción es `None`.

- El argumento final **traceback**, rastreo, es opcional (y se utiliza raramente en la práctica), y si está presente, indica el objeto del rastreo que se utiliza para la excepción.

Levantando excepciones (Raising exceptions) II

- Ejemplo:

```
def checkLevel(level):
    if level < 1:
        raise ValueError('Level must be 1 or larger')
        # The code below to this would not be executed
        # if we raise the exception
    else:
        print("Level is OK")

try:
    checkLevel(0)
except Exception as arg:
    # Catch exception
    print('Error:', arg)
```

Lista de excepciones estándar I

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.

Lista de excepciones estándar II

LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
IOError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.

Lista de excepciones estándar III

ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Recursos

- Real Python exceptions.
- Lectura recomendada: Exceptions.
- The Python Wiki.