Errores y excepciones

Hasta ahora los mensajes de error no habían sido más que mencionados, pero si probaste los ejemplos probablemente hayas visto algunos. Hay (al menos) dos tipos diferentes de errores: *errores de sintaxis* y *excepciones*.

Errores de sintaxis

Los errores de sintaxis, también conocidos como errores de interpretación, son quizás el tipo de queja más común que tenés cuando todavía estás aprendiendo Python:

```
>>> while True print('Hola mundo')
  File "<stdin>", line 1
    while True print('Hola mundo')
                   ^
SyntaxError: invalid syntax
```

El intérprete repite la línea culpable y muestra una pequeña 'flecha' que apunta al primer lugar donde se detectó el error. Este es causado por (o al menos detectado en) el símbolo que *precede* a la flecha: en el ejemplo, el error se detecta en la función **`print()`**, ya que faltan dos puntos (`':'`) antes del mismo. Se muestran el nombre del archivo y el número de línea para que sepas dónde mirar en caso de que la entrada venga de un programa.

Excepciones

Incluso si la declaración o expresión es sintácticamente correcta, puede generar un error cuando se intenta ejecutarla. Los errores detectados durante la ejecución se llaman *excepciones*, y no son incondicionalmente fatales: pronto aprenderás cómo manejarlos en los programas en Python. Sin embargo, la mayoría de las excepciones no son manejadas por los programas, y resultan en mensajes de error como los mostrados aquí:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

La última línea de los mensajes de error indica qué sucedió. Las excepciones vienen de distintos tipos, y el tipo se imprime como parte del mensaje: los tipos en el ejemplo son: **`ZeroDivisionError`**, **`NameError`** y **`TypeError`**. La cadena mostrada como tipo de la excepción es el nombre de la excepción predefinida que ocurrió. Esto es verdad para todas las excepciones predefinidas del intérprete, pero no necesita ser verdad para excepciones definidas por el usuario (aunque es una convención útil). Los nombres de las excepciones estándar son identificadores incorporados al intérprete (no son palabras clave reservadas).

El resto de la línea provee un detalle basado en el tipo de la excepción y qué la causó.

La parte anterior del mensaje de error muestra el contexto donde la excepción sucedió, en la forma de un *trazado del error* listando líneas fuente; sin embargo, no mostrará líneas leídas desde la entrada estándar.

Excepciones integradas lista las excepciones predefinidas y sus significados.

Manejando excepciones

Es posible escribir programas que manejen determinadas excepciones. Mirá el siguiente ejemplo, que le pide al usuario una entrada hasta que ingrese un entero válido, pero permite al usuario interrumpir el programa (usando Control-C o lo que sea que el sistema operativo soporte); notá que una interrupción generada por el usuario se señaliza generando la excepción `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(input("Por favor ingrese un número: "))
...         break
...     except ValueError:
...         print("Oops! No era válido. Intente nuevamente...")
...
```

La declaración `try` funciona de la siguiente manera:

- Primero, se ejecuta el *bloque try* (el código entre las declaración `try` y `except`).
- Si no ocurre ninguna excepción, el *bloque except* se saltea y termina la ejecución de la declaración `try`.
- Si ocurre una excepción durante la ejecución del *bloque try*, el resto del bloque se saltea. Luego, si su tipo coincide con la excepción nombrada luego de la palabra reservada `except`, se ejecuta el *bloque except*, y la ejecución continúa luego de la declaración `try`.
- Si ocurre una excepción que no coincide con la excepción nombrada en el `except`, esta se pasa a declaraciones `try` de más afuera; si no se encuentra nada que la maneje, es una *excepción no manejada*, y la ejecución se frena con un mensaje como los mostrados arriba.

Una declaración `try` puede tener más de un `except`, para especificar manejadores para distintas excepciones. A lo sumo un manejador será ejecutado. Sólo se manejan excepciones que ocurren en el correspondiente `try`, no en otros manejadores del mismo `try`. Un `except` puede nombrar múltiples excepciones usando paréntesis, por ejemplo:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Una clase en una clausula `except` es compatible con una excepción si la misma esta en la misma clase o una clase base de la misma (pero no de la otra manera --- una clausula except listando una clase derivada no es compatible con una clase base). Por ejemplo, el siguiente código imprimirá B, C, D, en ese orden:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
```

```
        print("C")
    except B:
        print("B")
```

Notese que si las clausulas de except estuvieran invertidas (con `except B` primero), habría impreso B, B, B --- la primera clausula de except coincidente es usada.

El último **`except`** puede omitir nombrar qué excepción captura, para servir como comodín. Usá esto con extremo cuidado, ya que de esta manera es fácil ocultar un error real de programación. También puede usarse para mostrar un mensaje de error y luego re-generar la excepción (permitiéndole al que llama, manejar también la excepción):

```
import sys

try:
    f = open('miarchivo.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("Error OS: {0}".format(err))
except ValueError:
    print("No pude convertir el dato a un entero.")
except:
    print("Error inesperado:", sys.exc_info()[0])
    raise
```

Las declaraciones **`try`** ... **`except`** tienen un *bloque else* opcional, el cual, cuando está presente, debe seguir a los except. Es útil para aquel código que debe ejecutarse si el *bloque try* no genera una excepción. Por ejemplo:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('no pude abrir', arg)
    else:
        print(arg, 'tiene', len(f.readlines()), 'lineas')
        f.close()
```

El uso de **`else`** es mejor que agregar código adicional en el **`try`** porque evita capturar accidentalmente una excepción que no fue generada por el código que está protegido por la declaración **`try`** ... **`except`**.

Cuando ocurre una excepción, puede tener un valor asociado, también conocido como el *argumento* de la excepción. La presencia y el tipo de argumento depende del tipo de excepción.

El **`except`** puede especificar una variable luego del nombre de excepción. La variable se vincula a una instancia de excepción con los argumentos almacenados en `instance.args`. Por conveniencia, la instancia de excepción define **`__str__()`** para que se pueda mostrar los argumentos directamente, sin necesidad de hacer referencia a `.args`. También se puede instanciar la excepción primero, antes de generarla, y agregarle los atributos que se desee:

```
>>> try:
...     raise Exception('carne', 'huevos')
... except Exception as inst:
...     print(type(inst))     # la instancia de excepción
...     print(inst.args)      # argumentos guardados en .args
...     print(inst)           # __str__ permite imprimir args directamente,
...                           # pero puede ser cambiado en subclases de la exc
...     x, y = inst           # desempacar argumentos
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
```

```
('carne', 'huevos')
('carne', 'huevos')
x = carne
y = huevos
```

Si una excepción tiene argumentos, estos se imprimen como la última parte (el 'detalle') del mensaje para las excepciones que no están manejadas.

Los manejadores de excepciones no manejan solamente las excepciones que ocurren en el *bloque try*, también manejan las excepciones que ocurren dentro de las funciones que se llaman (inclusive indirectamente) dentro del *bloque try*. Por ejemplo:

```
>>> def esto_falla():
...     x = 1/0
...
>>> try:
...     esto_falla()
... except ZeroDivisionError as err:
...     print('Manejando error en tiempo de ejecución:', err)
...
Manejando error en tiempo de ejecución: division by zero
```

Levantando excepciones

La declaración `raise` permite al programador forzar a que ocurra una excepción específica. Por ejemplo:

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

El único argumento a `raise` indica la excepción a generarse. Tiene que ser o una instancia de excepción, o una clase de excepción (una clase que hereda de `Exception`). Si se pasa una clase de excepción, la misma sera instanciada implicitamente llamandoa su constructor sin argumentos:

```
raise ValueError  # atajo para 'raise ValueError()'
```

Si necesitás determinar cuando una excepción fue lanzada pero no querés manejarla, una forma simplificada de la instrucción `raise` te permite relanzarla:

```
>>> try:
...     raise NameError('Hola')
... except NameError:
...     print('Voló una excepción!')
...     raise
...
Voló una excepción!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: Hola
```

Excepciones definidas por el usuario

Los programas pueden nombrar sus propias excepciones creando una nueva clase excepción (mirá *Clases* para más información sobre las clases de Python). Las excepciones, típicamente, deberán derivar de la clase `Exception`, directa o indirectamente.

Las clases de Excepciones pueden ser definidas de la misma forma que cualquier otra clase, pero usualmente se mantienen simples, a menudo solo ofreciendo un número de atributos con información sobre el error que leerán los manejadores de la

excepción. Al crear un módulo que puede lanzar varios errores distintos, una práctica común es crear una clase base para excepciones definidas en ese módulo y extenderla para crear clases excepciones específicas para distintas condiciones de error:

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

class EntradaError(Error):
    """Excepción 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 operacion intenta una transicion de estado no
    permitida.

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

La mayoría de las excepciones son definidas con nombres que terminan en "Error", similares a los nombres de las excepciones estándar.

Muchos módulos estándar definen sus propias excepciones para reportar errores que pueden ocurrir en funciones propias. Se puede encontrar más información sobre clases en el capítulo *Clases*.

Definiendo acciones de limpieza

La declaración `try` tiene otra cláusula opcional que intenta definir acciones de limpieza que deben ser ejecutadas bajo ciertas circunstancias. Por ejemplo:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Chau, mundo!')
...
Chau, mundo!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
```

Una *cláusula finally* siempre es ejecutada antes de salir de la declaración `try`, ya sea que una excepción haya ocurrido o no. Cuando ocurre una excepción en la cláusula `try` y no fue manejada por una cláusula `except` (o ocurrió en una cláusula `except` o `else`), es relanzada luego de que se ejecuta la cláusula `finally`. El `finally` es también ejecutado "a la salida"

cuando cualquier otra cláusula de la declaración `try` es dejada via `break`, `continue` or `return`. Un ejemplo más complicado:

```
>>> def dividir(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("¡división por cero!")
...     else:
...         print("el resultado es", result)
...     finally:
...         print("ejecutando la clausula finally")
...
>>> dividir(2, 1)
el resultado es 2.0
ejecutando la clausula finally
>>> dividir(2, 0)
¡división por cero!
ejecutando la clausula finally
>>> divide("2", "1")
ejecutando la clausula finally
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Como podés ver, la cláusula `finally` es ejecutada siempre. La excepción `TypeError` lanzada al dividir dos cadenas de texto no es manejado por la cláusula `except` y por lo tanto es relanzada luego de que se ejecuta la cláusula `finally` .

En aplicaciones reales, la cláusula `finally` es útil para liberar recursos externos (como archivos o conexiones de red), sin importar si el uso del recurso fue exitoso.

Acciones predefinidas de limpieza

Algunos objetos definen acciones de limpieza estándar que llevar a cabo cuando el objeto no es más necesitado, independientemente de que las operaciones sobre el objeto hayan sido exitosas o no. Mirá el siguiente ejemplo, que intenta abrir un archivo e imprimir su contenido en la pantalla.:

```
for linea in open("miarchivo.txt"):
    print(linea, end="")
```

El problema con este código es que deja el archivo abierto por un periodo de tiempo indeterminado luego de que esta parte termine de ejecutarse. Esto no es un problema en scripts simples, pero puede ser un problema en aplicaciones más grandes. La declaración `with` permite que objetos como archivos sean usados de una forma que asegure que siempre se los libera rápido y en forma correcta.:

```
with open("miarchivo.txt") as f:
    for linea in f:
        print(linea, end="")
```

Luego de que la declaración sea ejecutada, el archivo *f* siempre es cerrado, incluso si se encuentra un problema al procesar las líneas. Objetos que, como los archivos, provean acciones de limpieza predefinidas lo indicarán en su documentación.