Módulo 3 (Intermedio): Programación Orientada a Objetos - Excepciones

Más acerca de excepciones

El discutir sobre la programación orientada a objetos ofrece una muy buena oportunidad para volver a las excepciones. La naturaleza orientada a objetos de las excepciones de Python las convierte en una herramienta muy flexible, capaz de adaptarse a necesidades específicas, incluso aquellas que aún no conoces.

Antes de adentrarnos en el **lado orientado a objetos de las excepciones**, queremos mostrarte algunos aspectos sintácticos y semánticos de la forma en que Python trata el bloque try-except, ya que ofrece un poco más de lo que hemos presentado hasta ahora.

La primera característica que queremos analizar aquí es un bloque adicional que se puede colocar dentro (o más bien, directamente detrás) del bloque try-except: es la parte del código que comienza con *else*

```
def reciprocal(n):
    try:
        n = 1 / n
    except ZeroDivisionError:
        print("División fallida")
        return None
    else:
        print("Todo salió bien")
        return n


print(reciprocal(2))
print(reciprocal(0))
```

Un código etiquetado de esta manera se ejecuta cuando (y solo cuando) no se ha generado ninguna excepción dentro de la parte del *try:*. Podemos decir que este bloque se ejecuta después del *try:*, ya sea el que comienza con *except* (no olvides que puede haber más de un bloque de este tipo) o la que comienza con *else*.

Nota: el bloque *else:* debe ubicarse después del último bloque *except*.

El código de ejemplo produce el siguiente resultado:

```
Todo salió bien
0.5
División fallida
None
```

El try-except se puede extender de una manera más: agregando una parte encabezada por la palabra clave reservada *finally* (debe ser el último bloque del código diseñado para manejar excepciones).

Nota: estas dos variantes (*else* y *finally*) no son dependientes entre si, y pueden coexistir u ocurrir de manera independiente.

El bloque *finally* siempre se ejecuta (finaliza la ejecución del bloque try-except, de ahí su nombre), sin importar lo que sucedió antes, incluso cuando se genera una excepción, sin importar si esta se ha manejado o no.

```
def reciprocal(n):
```

```
    try:
        n = 1 / n
    except ZeroDivisionError:
        print("División fallida")
        n = None
    else:
        print("Todo salió bien")
    finally:
        print("Es momento de decir adiós")
        return n


print(reciprocal(2))
print(reciprocal(0))
```

Su salida es:

```
Todo salió bien
Es momento de decir adiós
0.5
División fallida
Es momento de decir adiós
None
```

Las excepciones son clases

Los ejemplos anteriores se centraron en detectar un tipo específico de excepción y responder de manera apropiada. Ahora vamos a profundizar más y mirar dentro de la excepción misma.

Probablemente no te sorprenderá saber que **las excepciones son clases**. Además, cuando se genera una excepción, se crea una instancia de un objeto de la clase y pasa por todos los niveles de ejecución del programa, buscando el bloque «except» que está preparado para tratar con la excepción.

Tal objeto lleva información útil que puede ayudarte a identificar con precisión todos los aspectos de la situación pendiente. Para lograr ese objetivo, Python ofrece una variante especial de la cláusula de excepción.

```
try:
    i = int("¡Hola!")
except Exception as e:
    print(e)
    print(e.__str__())
```

Como puedes ver, la sentencia *except* se extendió y contiene una frase adicional que comienza con la palabra clave reservada *as*, seguida por un identificador. El identificador está diseñado para capturar la excepción con el fin de analizar su naturaleza y sacar conclusiones adecuadas.

Nota: el alcance del identificador solo es dentro del *except*, y no va más allá.

El ejemplo presenta una forma muy simple de utilizar el objeto recibido: simplemente imprímelo (como puedes ver, la salida es producida por el método del objeto `__str__()`) y contiene un breve mensaje que describe la razón.

Se imprimirá el mismo mensaje si no hay un bloque *except* en el código, y Python se verá obligado a manejarlo

por sí mismo.

Todas las excepciones integradas de Python forman una jerarquía de clases. Si lo deseas, puedes extenderlo sin problema.

Como **un árbol es un ejemplo perfecto de una estructura de datos recursiva**, la recursión parece ser la mejor manera de recorrerlo. La función `print_exception_tree()` toma dos argumentos:

- Un punto dentro del árbol desde el cual comenzamos a recorrerlo.
- Un nivel de anidación (lo usaremos para construir un dibujo simplificado de las ramas del árbol).

Comencemos desde la raíz del árbol: la raíz de las clases de excepciones de Python es la clase *BaseException* (es una superclase de todas las demás excepciones).

```
def print_exception_tree(thisclass, nest = 0):
    if nest > 1:
        print("   |" * (nest - 1), end="")
    if nest > 0:
        print("   +---", end="")

    print(thisclass.__name__)

    for subclass in thisclass.__subclasses__():
        print_exception_tree(subclass, nest + 1)


print_exception_tree(BaseException)
```

Para cada una de las clases encontradas, se realiza el mismo conjunto de operaciones:

- Imprimir su nombre, tomado de la propiedad __name__.
- Iterar a través de la lista de subclases provistas por el método `__subclasses__()`, e invocar recursivamente la función `printExcTree()`, incrementando el nivel de anidación respectivamente. Toma en cuenta como hemos dibujado las ramas. La impresión no está ordenada de alguna manera: si deseas un desafío, puedes intentar ordenarla tú mismo. Además, hay algunas imprecisiones sutiles en la forma en que se presentan algunas ramas. Eso también se puede arreglar, si lo deseas.

Así es como se ve:

```
BaseException
   +---Exception
   |   +---TypeError
   |   +---StopAsyncIteration
   |   +---StopIteration
   |   +---ImportError
   |   |   +---ModuleNotFoundError
   |   |   +---ZipImportError
   |   +---OSError
   |   |   +---ConnectionError
   |   |   |   +---BrokenPipeError
   |   |   |   +---ConnectionAbortedError
   |   |   |   +---ConnectionRefusedError
   |   |   |   +---ConnectionResetError
   |   |   +---BlockingIOError
   |   |   +---ChildProcessError
   |   |   +---FileExistsError
   |   |   +---FileNotFoundError
```

```
    |   |   +---IsADirectoryError
    |   |   +---NotADirectoryError
    |   |   +---InterruptedError
    |   |   +---PermissionError
    |   |   +---ProcessLookupError
    |   |   +---TimeoutError
    |   |   +---UnsupportedOperation
    |   |   +---herror
    |   |   +---gaierror
    |   |   +---timeout
    |   |   +---Error
    |   |   |   +---SameFileError
    |   |   +---SpecialFileError
    |   |   +---ExecError
    |   |   +---ReadError
    |   +---EOFError
    |   +---RuntimeError
    |   |   +---RecursionError
    |   |   +---NotImplementedError
    |   |   +---_DeadlockError
    |   |   +---BrokenBarrierError
    |   +---NameError
    |   |   +---UnboundLocalError
    |   +---AttributeError
    |   +---SyntaxError
    |   |   +---IndentationError
    |   |   |   +---TabError
    |   +---LookupError
    |   |   +---IndexError
    |   |   +---KeyError
    |   |   +---CodecRegistryError
    |   +---ValueError
    |   |   +---UnicodeError
    |   |   |   +---UnicodeEncodeError
    |   |   |   +---UnicodeDecodeError
    |   |   |   +---UnicodeTranslateError
    |   |   +---UnsupportedOperation
    |   +---AssertionError
    |   +---ArithmeticError
    |   |   +---FloatingPointError
    |   |   +---OverflowError
    |   |   +---ZeroDivisionError
    |   +---SystemError
    |   |   +---CodecRegistryError
    |   +---ReferenceError
    |   +---BufferError
    |   +---MemoryError
    |   +---Warning
    |   |   +---UserWarning
    |   |   +---DeprecationWarning
    |   |   +---PendingDeprecationWarning
    |   |   +---SyntaxWarning
    |   |   +---RuntimeWarning
    |   |   +---FutureWarning
```

```
   |   |   +---ImportWarning
   |   |   +---UnicodeWarning
   |   |   +---BytesWarning
   |   |   +---ResourceWarning
   |   +---error
   |   +---Verbose
   |   +---Error
   |   +---TokenError
   |   +---StopTokenizing
   |   +---Empty
   |   +---Full
   |   +---_OptionError
   |   +---TclError
   |   +---SubprocessError
   |   |   +---CalledProcessError
   |   |   +---TimeoutExpired
   |   +---Error
   |   |   +---NoSectionError
   |   |   +---DuplicateSectionError
   |   |   +---DuplicateOptionError
   |   |   +---NoOptionError
   |   |   +---InterpolationError
   |   |   |   +---InterpolationMissingOptionError
   |   |   |   +---InterpolationSyntaxError
   |   |   |   +---InterpolationDepthError
   |   |   +---ParsingError
   |   |   |   +---MissingSectionHeaderError
   |   +---InvalidConfigType
   |   +---InvalidConfigSet
   |   +---InvalidFgBg
   |   +---InvalidTheme
   |   +---EndOfBlock
   |   +---BdbQuit
   |   +---error
   |   +---_Stop
   |   +---PickleError
   |   |   +---PicklingError
   |   |   +---UnpicklingError
   |   +---_GiveupOnSendfile
   |   +---error
   |   +---LZMAError
   |   +---RegistryError
   |   +---ErrorDuringImport
   +---GeneratorExit
   +---SystemExit
   +---KeyboardInterrupt
```

Anatomía detallada de las excepciones

Echemos un vistazo más de cerca al objeto de la excepción, ya que hay algunos elementos realmente interesantes aquí (volveremos al tema pronto cuando consideremos las técnicas base de entrada y salida de Python, ya que su subsistema de excepción extiende un poco estos objetos).

La clase *BaseException* introduce una propiedad llamada *args*. Es una tupla **diseñada para reunir todos los**

argumentos pasados al constructor de la clase. Está vacío si la construcción se ha invocado sin ningún argumento, o solo contiene un elemento cuando el constructor recibe un argumento (no se considera el argumento *self* aquí), y así sucesivamente.

Hemos preparado una función simple para imprimir la propiedad *args* de una manera elegante, puedes ver la función en el editor.

```
def print_args(args):
    lng = len(args)
    if lng == 0:
        print("")
    elif lng == 1:
        print(args[0])
    else:
        print(str(args))


try:
    raise Exception
except Exception as e:
    print(e, e.__str__(), sep=' : ' ,end=' : ')
    print_args(e.args)

try:
    raise Exception("mi excepción")
except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    print_args(e.args)

try:
    raise Exception("mi", "excepción")
except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    print_args(e.args)
```

Hemos utilizado la función para imprimir el contenido de la propiedad *args* en tres casos diferentes, donde la excepción de la clase Exception es generada de tres maneras distintas. Para hacerlo más espectacular, también hemos impreso el objeto en sí, junto con el resultado de la invocación `__str__()`.

El primer caso parece de rutina, solo hay el nombre Exception después de la palabra clave reservada raise. Esto significa que el objeto de esta clase se ha creado de la manera más rutinaria.

El segundo y el tercer caso pueden parecer un poco extraños a primera vista, pero no hay nada extraño, son solo las invocaciones del constructor. En la segunda sentencia *raise*, el constructor se invoca con un argumento, y en el tercero, con dos.

Como puedes ver, la salida del programa refleja esto, mostrando los contenidos apropiados de la propiedad *args*:

```
 :  :
mi excepción : mi excepción : mi excepción
('mi', 'excepción') : ('mi', 'excepción') : ('mi', 'excepción')
```

Cómo crear tu propia excepción

La jerarquía de excepciones no está cerrada ni terminada, y siempre puedes ampliarla si deseas o necesitas crear tu propio mundo poblado con tus propias excepciones.

Puede ser útil cuando se crea un módulo complejo que detecta errores y genera excepciones, y deseas que las excepciones se distingan fácilmente de cualquier otra de Python.

Esto se puede hacer al **definir tus propias excepciones como subclases derivadas de las predefinidas**.

Nota: si deseas crear una excepción que se utilizará como un caso especializado de cualquier excepción incorporada, derivala solo de esta. Si deseas construir tu propia jerarquía, y no quieres que esté estrechamente conectada al árbol de excepciones de Python, derivala de cualquiera de las clases de excepción principales, tal como: Exception.

Imagina que has creado una aritmética completamente nueva, regida por sus propias leyes y teoremas. Está claro que la división también se ha redefinido, y tiene que comportarse de una manera diferente a la división de rutina. También está claro que esta nueva división debería plantear su propia excepción, diferente de la incorporada ZeroDivisionError, pero es razonable suponer que, en algunas circunstancias, tu (o el usuario de tu aritmética) pueden tratar todas las divisiones entre cero de la misma manera.

Demandas como estas pueden cumplirse en la forma presentada en el editor.

```
class MyZeroDivisionError(ZeroDivisionError):
    pass


def do_the_division(mine):
    if mine:
        raise MyZeroDivisionError("peores noticias")
    else:
        raise ZeroDivisionError("malas noticias")


for mode in [False, True]:
    try:
        do_the_division(mode)
    except ZeroDivisionError:
        print('División entre cero')

for mode in [False, True]:
    try:
        do_the_division(mode)
    except MyZeroDivisionError:
        print('Mi división entre cero')
    except ZeroDivisionError:
        print('División entre cero original')
```

- Hemos definido nuestra propia excepción, llamada *MyZeroDivisionError*, derivada de la incorporada *ZeroDivisionError*. Como puedes ver, hemos decidido no agregar ningún componente nuevo a la clase. En efecto, una excepción de esta clase puede ser, dependiendo del punto de vista deseado, tratada como una simple excepción ZeroDivisionError, o puede ser considerada por separado.
- La función `do_the_division()` genera una excepción *MyZeroDivisionError* o *ZeroDivisionError* dependiendo del valor del argumento. La función se invoca cuatro veces en total, mientras que las dos primeras invocaciones se manejan utilizando solo un bloque except (la más general), las dos últimas invocan dos bloques diferentes, capaces de distinguir las excepciones (no lo olvides: el orden de los

bloques hace una diferencia fundamental).

Cuando vas a construir un universo completamente nuevo lleno de criaturas completamente nuevas que no tienen nada en común con todas las cosas familiares, es posible que desees **construir tu propia estructura de excepciones**.

Por ejemplo, si trabajas en un gran sistema de simulación destinado a modelar las actividades de un restaurante de pizza, puede ser conveniente formar una jerarquía de excepciones por separado.

Puedes comenzar a construirla **definiendo una excepción general como una nueva clase base** para cualquier otra excepción especializada. Lo hemos hecho de la siguiente manera:

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza = pizza
```

Nota: vamos a recopilar más información específica aquí de lo que recopila una Excepción regular, entonces nuestro constructor tomará dos argumentos:

- Uno que especifica una pizza como tema del proceso.
- Otro que contiene una descripción más o menos precisa del problema.

Como puedes ver, pasamos el segundo parámetro al constructor de la superclase y guardamos el primero dentro de nuestra propiedad.

Un problema más específico (como un exceso de queso) puede requerir una excepción más específica. Es posible derivar la nueva clase de la ya definida *PizzaError*, como hemos hecho aquí:

```
class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError._init__(self, pizza, message)
        self.cheese = cheese
```

La excepción *TooMuchCheeseError* necesita más información que la excepción regular *PizzaError*, así que lo agregamos al constructor, el nombre cheese es entonces almacenado para su posterior procesamiento.

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese


def make_pizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError(pizza, "no hay tal pizza en el menú")
    if cheese > 100:
        raise TooMuchCheeseError(pizza, cheese, "demasiado queso")
    print("¡Pizza lista!")
```

```
for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
    try:
        make_pizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

Combinamos las dos excepciones previamente definidas y las aprovechamos para que funcionen en un pequeño ejemplo.

Una de ellas es generada dentro de la función `make_pizza()` cuando ocurra cualquiera de estas dos situaciones erróneas: una solicitud de pizza incorrecta o una solicitud de una pizza con demasiado queso.

Nota:

- El remover el bloque que comienza con *except TooMuchCheeseError* hará que todas las excepciones que aparecen se clasifiquen como *PizzaError*.
- El remover el bloque que comienza con *except PizzaError* provocará que la excepción *TooMuchCheeseError* no pueda ser manejada, y hará que el programa finalice.

La solución anterior, aunque elegante y eficiente, tiene una debilidad importante. Debido a la manera algo fácil de declarar los constructores, las nuevas excepciones no se pueden usar tal cual, sin una lista completa de los argumentos requeridos.

Eliminaremos esta debilidad **estableciendo valores predeterminados para todos los parámetros del constructor**. Observa:

```
class PizzaError(Exception):
    def __init__(self, pizza='desconocida', message=''):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza='desconocida', cheese='>100', message=''):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese


def make_pizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError
    if cheese > 100:
        raise TooMuchCheeseError
    print("¡Pizza lista!")


for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
    try:
        make_pizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

Ahora, si las circunstancias lo permiten, es posible usar únicamente los nombres de clase.

Puntos Clave

1. El bloque else: de la sentencia try se ejecuta cuando no ha habido ninguna excepción durante la ejecución del try:.

2. El bloque finally: de la sentencia try es **siempre** ejecutado.

3. La sintaxis except *Exception_Name as exception_object:* te permite interceptar un objeto que contiene información sobre una excepción pendiente. La propiedad del objeto llamada args (una tupla) almacena todos los argumentos pasados al constructor del objeto.

4. Las clases de excepciones pueden extenderse para enriquecerlas con nuevas capacidades o para adoptar sus características a excepciones recién definidas.

Por ejemplo:

```
try:
    assert __name__ == "__main__"
except:
    print("fallido", end=' ')
else:
    print("éxito", end=' ')
finally:
    print("terminado")
```

El código da como salida: éxito terminado.