Módulo 2 (Intermedio): Excepciones

Errores, fallas y otras plagas

```
Cualquier cosa que pueda salir mal, saldrá mal.
```

Esta es la ley de Murphy, y funciona en todo y siempre. Si la ejecución del código puede salir mal, lo hará.

```
import math

x = float(input("Ingresa x: "))
y = math.sqrt(x)

print("La raíz cuadrada de", x, "es igual a", y)
```

Hay al menos dos formas posibles de que «salga mal» la ejecución. ¿Puedes verlas?

- Como el usuario puede ingresar una cadena de caracteres completamente arbitraria, no hay garantía de que la cadena se pueda convertir en un valor flotante: esta es la primera vulnerabilidad del código.
- La segunda es que la función sqrt() fallará si se le ingresa un valor negativo.

Puedes recibir alguno de los siguientes mensajes de error.

Algo como esto:

```
Ingresa x: Abracadabra

Traceback (most recent call last):

  File "sqrt.py", line 3, in <module>

    x = float(input("Ingresa x: "))

ValueError: could not convert string to float: 'Abracadabra'
```

O algo como esto:

```
Ingresa x: -1

Traceback (most recent call last):

  File "sqrt.py", line 4, in <module>

    y = math.sqrt(x)

ValueError: math domain error
```

¿Puedes protegerte de tales sorpresas? Por supuesto. Además, tienes que hacerlo para ser considerado un buen programador.

Excepciones

Cada vez que tu código intenta hacer algo erróneo, irresponsable o inaplicable, Python hace dos cosas:

- **Detiene tu programa**.
- Crea un tipo especial de dato, llamado **excepción**.

Ambas actividades llevan por nombre **generar una excepción**. Podemos decir que Python siempre genera una excepción (o que una **excepción ha sido generada**) cuando no tiene idea de qué hacer con el código.

¿Qué ocurre después?

La excepción generada espera que alguien o algo lo note y haga algo al respecto. Si la excepción no es resuelta, el programa será **terminado abruptamente**, y verás un **mensaje de error** enviado a la consola por Python. De otra manera, si se atiende la excepción y es **manejada** apropiadamente, el programa puede reanudarse y su ejecución puede continuar. Python proporciona herramientas efectivas que permiten **observar, identificar y manejar las excepciones** eficientemente. Esto es posible debido a que todas las excepciones potenciales tienen un nombre específico, por lo que se pueden clasificar y reaccionar a ellas adecuadamente.

Ya conoces algunos nombres de las excepciones. Observa el siguiente mensaje de diagnóstico:

```
ValueError: math domain error
```

La palabra resaltada es el nombre de la excepción. Vamos a familiarizarnos con algunas otras excepciones.

ZeroDivisionError

```
value = 1
value /= 0
```

Ejecuta el (obviamente incorrecto) programa.

Verás el siguiente mensaje en respuesta:

```
Traceback (most recent call last):
File "div.py", line 2, in
value /= 0
ZeroDivisionError: division by zero
```

Este error es llamado **ZeroDivisionError**.

IndexError

```
my_list = []
x = my_list[0]
```

```
Traceback (most recent call last):
File "lst.py", line 2, in
x = list[0]
IndexError: list index out of range
```

¿Cómo se **manejan** las excepciones? La palabra try es clave para la solución.

Además, también es una palabra clave reservada.

La receta para el éxito es la siguiente:

- Primero, **se debe intentar (try) hacer algo**.
- Después, tienes que **comprobar si todo salió bien**.

Pero, ¿no sería mejor verificar primero todas las circunstancias y luego hacer algo solo si es seguro?

```
first_number = int(input("Ingresa el primer numero: "))
second_number = int(input("Ingresa el segundo numero: "))

if second_number != 0:
    print(first_number / second_number)
else:
    print("Esta operación no puede ser realizada.")

print("FIN.")
```

Es cierto que esta forma puede parecer la mas natural y comprensible, pero en realidad, este método no facilita la programación. Todas estas revisiones pueden hacer al código demasiado grande e ilegible.

Python prefiere un enfoque completamente diferente.

```
first_number = int(input("Ingresa el primer numero: "))
second_number = int(input("Ingresa el segundo numero: "))

try:
    print(first_number / second_number)
except:
    print("Esta operación no puede ser realizada.")

print("FIN.")
```

Este es el enfoque favorito de Python.

Nota:

- La palabra reservada **try comienza con un bloque de código** el cual puede o no estar funcionando correctamente.
- Después, Python intenta realizar la acción arriesgada: si falla, se genera una excepción y Python comienza a buscar una solución.
- La palabra reservada **except** comienza con un bloque de código que será **ejecutado si algo dentro del bloque try sale mal**: si se genera una excepción dentro del bloque anterior try, **fallará aquí**, entonces el código ubicado después de la palabra clave reservada except debería proporcionar una reacción adecuada a la excepción planteada.
- Por ultimo, se regresa al nivel de anidación anterior, es decir, se termina la sección **try-except**.

Ejecuta el código y prueba su comportamiento.

Resumamos esto:

```
try:
    :
    :
```

```
except:
    :
    :
```

- En el primer paso, Python intenta realizar todas las instrucciones colocadas entre las instrucciones try: y except:.
- Si no hay ningún problema con la ejecución y todas las instrucciones se realizan con éxito, la ejecución salta al punto después de la última línea del bloque except: , y la ejecución del bloque se considera completa.
- Si algo sale mal dentro del bloque try: o except:, la ejecución salta inmediatamente fuera del bloque y entra en la primera instrucción ubicada después de la palabra clave reservada except:, esto significa que algunas de las instrucciones del bloque pueden ser silenciosamente omitidas.

```
try:
    print("1")
    x = 1 / 0
    print("2")
except:
    print("Oh cielos, algo salió mal...")

print("3")
```

```
1
Oh cielos, algo salió mal...
3
```

Nota: la instrucción print(«2») se perdió en el proceso.

Este enfoque tiene una desventaja importante: si existe la posibilidad de que más de una excepción se salte a un apartado except:, puedes tener **problemas para descubrir lo que realmente sucedió**.

Al igual que en el código en el editor. Ejecútalo y ve lo que pasa.

El mensaje: *Oh cielos, algo salio mal...* que aparece en la consola no dice nada acerca de la razón, mientras que hay dos posibles causas de la excepción:

- Datos no enteros fueron ingresados por el usuario.
- Un valor entero igual a 0 fue asignado a la variable x.

Técnicamente, hay dos formas de resolver el problema:

- Construir dos bloques consecutivos try-except, uno por cada posible motivo de excepción (fácil, pero provocará un crecimiento desfavorable del código).
- Emplear una variante más avanzada de la instrucción.

Se ve de la siguiente manera:

```
try:
    :
except exc1:
    :
except exc2:
    :
except:
    :
```

Así es como funciona:

- Si el try genera la excepción exc1, esta será manejada por el bloque except exc1:.
- De la misma manera, si el try genera la excepción exc2, esta será manejada por el bloque except exc2:.
- Si el try genera cualquier otra excepción, será manejado por el bloque sin nombre except.

```
try:
    x = int(input("Ingresa un numero: "))
    y = 1 / x
    print(y)
except ZeroDivisionError:
    print("No puedes dividir entre cero, lo siento.")
except ValueError:
    print("Debes ingresar un valor entero.")
except:
    print("Oh cielos, algo salió mal...")

print("FIN.")
```

Nuestra solución esta ahí.

El código, cuando se ejecute, producirá una de las siguientes cuatro variantes de salida:

Si se ingresa un valor entero válido distinto de cero (por ejemplo, 5) dirá:

```
0.2
FIN.
```

Si se ingresa 0, dirá:

```
No puedes dividir entre cero, lo siento.
FIN.
```

Si se ingresa cualquier cadena no entera, verás:

```
Debes ingresar un valor entero.
FIN.
```

(Localmente en tu máquina) si presionas Ctrl-C mientras el programa está esperando la entrada del usuario (provocará una excepción denominada KeyboardInterrupt), el programa dirá:

```
Oh cielos, algo salió mal...
FIN.
```

No olvides que:

- Los bloques except son analizados en el mismo orden en que aparecen en el código.
- No debes usar más de un bloque de excepción con el mismo nombre.
- El número de diferentes bloques except es arbitrario, la única condición es que si se emplea el try, debes poner al menos un except (nombrado o no) después de el.
- La palabra clave reservada except no debe ser empleada sin que le preceda un try.
- Si uno de los bloques except es ejecutado, ningún otro lo será.
- Si ninguno de los bloques except especificados coincide con la excepción planteada, la excepción permanece sin manejar (lo discutiremos pronto).
- Si un except sin nombre existe, tiene que especificarse como el último.

```
try:
    :
except exc1:
    :
except exc2:
    :
except:
    :
```

Continuemos ahora con los experimentos.

```
try:
    x = int(input("Ingresa un numero: "))
    y = 1 / x
    print(y)
except ValueError:
    print("Debes ingresar un valor entero.")
except:
    print("Oh cielos, algo salió mal...")

print("FIN.")
```

Hemos modificado el programa anterior, hemos eliminado el bloque *ZeroDivisionError*.

¿Qué sucede ahora si el usuario ingresa un 0 como entrada?

Como no existe un **bloque declarado** para la división entre cero, la excepción cae dentro del bloque **general (sin nombre)**: esto significa que en este caso, el programa dirá:

```
Oh cielos, algo salió mal...
FIN.
```

Echemos a perder el código una vez más.

```
try:
    x = int(input("Ingresa un número: "))
    y = 1 / x
    print(y)
except ValueError:
    print("Debes ingresar un valor entero.")

print("FIN.")
```

Esta vez, hemos eliminado el bloque sin nombre.

El usuario ingresa nuevamente un 0, y:

- La excepción no será manejada por ValueError: no tiene nada que ver con ello.
- Como no hay otro bloque, deberías ver este mensaje:

```
Traceback (most recent call last):
File "exc.py", line 3, in
y = 1 / x
```

```
ZeroDivisionError: division by zero
```

Puntos Clave

1. Una excepción es un evento durante la ejecución del programa causado por una situación anormal. La excepción debe manejarse para evitar la terminación del programa. La parte del código que se sospecha que es la fuente de la excepción debe colocarse dentro del bloque try.

Cuando ocurre la excepción, la ejecución del código no se termina, sino que salta al bloque except. Este es el lugar donde debe llevarse a cabo el manejo de la excepción. El esquema general para tal construcción es el siguiente:

```
:
# El código que siempre corre suavemente.
:
try:
    :
    # Código arriesgado.
    :
except:
    :
    # La gestión de la crisis se lleva a cabo aquí.
    :
:
# De vuelta a la normalidad.
:
```

2. Si necesitas manejar más de una excepción proveniente del mismo bloque try, puedes agregar más de un bloque except, pero debes etiquetarlos con diferentes nombres, así:

```
:
# El código que siempre corre suavemente.
:
try:
    :
    # Código arriesgado.
    :
except Except_1:
    # La gestión de la crisis se lleva a cabo aquí.
except Except_2:
    # Salvamos el mundo aqui.
:
# De vuelta a la normalidad.
:
```

En el mejor caso, se ejecuta uno de los bloques except; ninguno de los bloques se ejecuta cuando la excepción generada no coincide con ninguna de las excepciones especificadas.

3. No se puede agregar más de un bloque de excepción sin nombre después de los bloques con nombre.

```
:
# El código que siempre corre suavemente.
:
try:
```

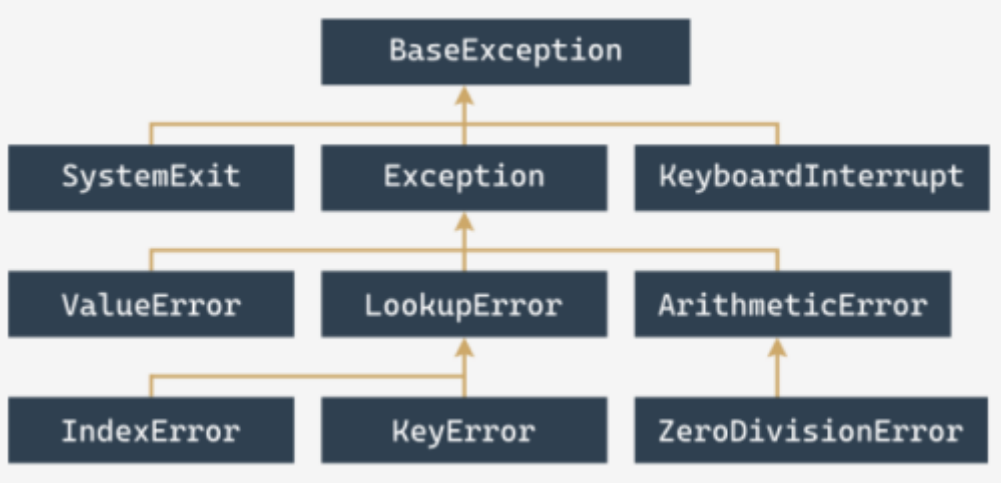
```
    :
    # Código arriesgado.
    :
except Except_1:
    # La gestión de la crisis se lleva a cabo aquí.
except Except_2:
    # Salvamos el mundo aqui.
except:
    # Todos los demás problemas caen aquí.
:
# De vuelta a la normalidad.
:
```

Excepciones

Python 3 define **63 excepciones integradas**, y todas ellas forman una **jerarquía en forma de árbol**, aunque el árbol es un poco extraño ya que su raíz se encuentra en la parte superior.

Algunas de las excepciones integradas son más generales (incluyen otras excepciones) mientras que otras son completamente concretas (solo se representan a sí mismas). Podemos decir que **cuanto más cerca de la raíz se encuentra una excepción, más general (abstracta) es**. A su vez, las excepciones ubicadas en los extremos del árbol (podemos llamarlas **hojas**) son concretas.

Observa la figura:



Muestra una pequeña sección del árbol completo de excepciones. Comencemos examinando el árbol desde la hoja ZeroDivisionError.

Nota:

- ZeroDivisionError es un caso especial de una clase de excepción más general llamada ArithmeticError.
- ArithmeticError es un caso especial de una clase de excepción más general llamada solo Exception.
- Exception es un caso especial de una clase más general llamada BaseException.

Podemos describirlo de la siguiente manera (observa la dirección de las flechas; siempre apuntan a la entidad más general):

BaseException ↑ Exception ↑ ArithmeticError ↑ ZeroDivisionError

Te mostraremos el funcionamiento esta generalización. Comencemos con un código realmente simple.

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("Uuupsss...")

print("FIN.")
```

Es un ejemplo simple para comenzar. Ejecútalo.

La salida que esperamos ver es la siguiente:

```
Uuuppsss...
FIN.
```

Ahora observa el código a continuación:

```
try:
    y = 1 / 0
except ArithmeticError:
    print("Uuuppsss...")

print("FIN.")
```

Algo ha cambiado: hemos reemplazado **ZeroDivisionError** con **ArithmeticError**.

Ya se sabe que ArithmeticError es una clase general que incluye (entre otras) la excepción ZeroDivisionError.

Por lo tanto, la salida del código permanece sin cambios. Pruébalo.

Esto también significa que reemplazar el nombre de la excepción ya sea con Exception o BaseException no cambiará el comportamiento del programa.

Vamos a resumir:

- Cada excepción generada cae en la primer coincidencia.
- La coincidencia correspondiente no tiene que especificar exactamente la misma excepción, es suficiente que la excepción sea más general (más abstracta) que la generada.

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("¡División entre Cero!")
except ArithmeticError:
    print("¡Problema Aritmético!")

print("FIN.")
```

La primera coincidencia es la que contiene ZeroDivisionError. Significa que la consola mostrará:

```
¡División entre Cero!
FIN.
```

¿Cambiará algo si intercambiamos los dos bloques except? Justo como aquí abajo:

```
try:
    y = 1 / 0
except ArithmeticError:
    print("¡Problema Aritmético!")
except ZeroDivisionError:
    print("¡División entre Cero!")

print("FIN.")
```

El cambio es radical: la salida del código es ahora:

```
¡Problema Aritmético!
FIN.
```

¿Por qué, si la excepción generada es la misma que antes?

La excepción es la misma, pero la excepción más general ahora aparece primero: también capturará todas las divisiones entre cero. También significa que no hay posibilidad de que alguna excepción llegue a ZeroDivisionError. Ahora es completamente inalcanzable.

Recuerda:

- ¡El orden de las excepciones importa!
- No pongas excepciones más generales antes que otras más concretas.
- Esto hará que el último sea inalcanzable e inútil.
- Además, hará que el código sea desordenado e inconsistente.
- Python no generará ningún mensaje de error con respecto a este problema.

Si deseas **manejar dos o más excepciones** de la misma manera, puedes usar la siguiente sintaxis:

```
try:
    :
except (exc1, exc2):
    :
```

Simplemente tienes que poner todos los nombres de las excepciones empleadas en una lista separada por comas y no olvidar los paréntesis.

Si una **excepción es generada dentro de una función**, puede ser manejada:

- Dentro de la función.
- Fuera de la función.

```
def bad_fun(n):
    try:
        return 1 / n
    except ArithmeticError:
        print("¡Problema Aritmético!")
    return None

bad_fun(0)

print("FIN.")
```

La excepción ZeroDivisionError (la cual es un caso concreto de la clase ArithmeticError) es generada dentro de

la función badfun(), y la función en sí misma se encarga de ella.

La salida del programa es:

```
¡Problema Aritmético!
FIN.
```

También es posible dejar que la excepción se propague **fuera de la función**. Probémoslo ahora.

Observa el código a continuación:

```
def bad_fun(n):
    return 1 / n

try:
    bad_fun(0)
except ArithmeticError:
    print("¿Que pasó? ¡Se generó una excepción!")

print("FIN.")
```

El problema tiene que ser resuelto por el invocador (o por el invocador del invocador, y así sucesivamente).

La salida del programa es:

```
¿Qué pasó? ¡Se generó una excepción!
FIN.
```

Nota: la **excepción generada puede cruzar la función y los límites del módulo**, y viajar a través de la cadena de invocación buscando una cláusula except capaz de manejarla.

Si no existe tal cláusula, la excepción no se controla y Python resuelve el problema de la manera estándar - **terminando el código y emitiendo un mensaje de diagnóstico**.

La instrucción **raise** genera la excepción especificada denominada **exc** como si fuese generada de manera natural:

```
raise exc
```

Nota: raise es una palabra clave reservada.

La instrucción te permite:

- **Simular excepciones reales** (por ejemplo, para probar tu estrategia de manejo de excepciones).
- Parcialmente **manejar una excepción** y hacer que otra parte del código sea responsable de completar el manejo.

```
def bad_fun(n):
    raise ZeroDivisionError


try:
    bad_fun(0)
except ArithmeticError:
    print("¿Que pasó? ¿Un error?")
```

```
print("FIN.")
```

Así es como puedes usarlo en la práctica.

La salida del programa permanece sin cambios.

De esta manera, puedes **probar tu rutina de manejo de excepciones** sin forzar al código a hacer cosas incorrectas.

La instrucción raise también se puede utilizar de la siguiente manera (toma en cuenta la ausencia del nombre de la excepción):

```
raise
```

Existe una seria restricción: esta variante de la instrucción raise puede ser utilizada **solamente dentro del bloque except**; usarla en cualquier otro contexto causa un error.

La instrucción volverá a generar la misma excepción que se maneja actualmente.

Gracias a esto, puedes distribuir el manejo de excepciones entre diferentes partes del código.

```
def bad_fun(n):
    try:
        return n / 0
    except:
        print("¡Lo hice otra vez!")
        raise


try:
    bad_fun(0)
except ArithmeticError:
    print("¡Ya veo!")

print("FIN.")
```

La excepción ZeroDivisionError es generada dos veces:

- Primero, dentro del try debido a que se intentó realizar una división entre cero.
- Segundo, dentro de la parte except por la instrucción raise.

En efecto, la salida del código es:

```
¡Lo hice otra vez!
¡Ya veo!
FIN.
```

Ahora es un buen momento para mostrarte otra instrucción de Python, llamada **assert** (afirmar). Esta es una palabra clave reservada.

```
assert expression
```

¿Cómo funciona?

- Se evalúa la expresión.

- Si la expresión se evalúa como True (Verdadera), o un valor numérico distinto de cero, o una cadena no vacía, o cualquier otro valor diferente de None, no hará nada más.
- De lo contrario, automáticamente e inmediatamente se genera una excepción llamada AssertionError (en este caso, decimos que la afirmación ha fallado).

¿Cómo puede ser utilizada?

- Puedes ponerlo en la parte del código donde quieras estar **absolutamente a salvo de datos incorrectos**, y donde no estés absolutamente seguro de que los datos hayan sido examinados cuidadosamente antes (por ejemplo, dentro de una función utilizada por otra persona).
- El generar una excepción AssertionError asegura que tu código no produzca resultados no válidos y muestra claramente la naturaleza de la falla.
- Las aserciones no reemplazan las excepciones ni validan los datos, son suplementos.

Si las excepciones y la validación de datos son como conducir con cuidado, la aserción puede desempeñar el papel de una bolsa de aire.

Veamos a la instrucción assert en acción.

```
import math

x = float(input("Ingresa un número: "))
assert x >= 0.0

x = math.sqrt(x)

print(x)
```

El programa se ejecuta sin problemas si se ingresa un valor numérico válido mayor o igual a cero; de lo contrario, se detiene y emite el siguiente mensaje:

```
Traceback (most recent call last):
  File ".main.py", line 4, in
    assert x >= 0.0
AssertionError
```

Puntos Clave

1. No se puede agregar más de un bloque except sin nombre después de los bloques con nombre.

```
:
# El código que siempre corre suavemente.
:
try:
    :
    # Código arriesgado.
    :
except Except_1:
    # La gestión de la crisis se lleva a cabo aquí.
except Except_2:
    # Salvamos el mundo aqui.
except:
    # Todos los demás problemas caen aquí.
:
# De vuelta a la normalidad.
```

```
:
```

2. Todas las excepciones de Python predefinidas forman una jerarquía, es decir, algunas de ellas son más generales (la llamada BaseException es la más general) mientras que otras son más o menos concretas (por ejemplo, IndexError es más concreta que LookupError).

No debes poner excepciones más concretas antes de las más generales dentro de la misma secuencia de bloques except. Por ejemplo, puedes hacer esto:

```
try:
    # Código arriesgado.
except IndexError:
    # Solucionando problemas con listas.
except LookupError:
    # Lidiando con búsquedas erróneas.
```

Pero no hagas esto (a menos de que estés absolutamente seguro de que quieres que alguna parte de tu código sea inaccesible).

```
try:
    # Código arriesgado.
except LookupError:
    # Lidiando con búsquedas erróneas.
except IndexError:
    # Nunca llegarás aquí.
```

3. La sentencia de Python **raise** ExceptionName puede generar una excepción bajo demanda. La misma sentencia pero sin ExceptionName, se puede usar **solamente** dentro del bloque try, y genera la misma excepción que se está manejando actualmente.

4. La sentencia de Python **assert expression** evalúa la expresión y genera la excepción AssertError cuando la expresión es igual a cero, una cadena vacía o None. Puedes usarla para proteger algunas partes críticas de tu código de datos devastadores.

Excepciones integradas

Te mostraremos una breve lista de las excepciones más útiles. Si bien puede sonar extraño llamar «útil» a una cosa o un fenómeno que es un signo visible de una falla o retroceso, como sabes, errar es humano y si algo puede salir mal, saldrá mal.

Las excepciones son tan rutinarias y normales como cualquier otro aspecto de la vida de un programador.

Para cada excepción, te mostraremos:

- Su nombre.
- Su ubicación en el árbol de excepciones.
- Una breve descripción.
- Un fragmento de código conciso que muestre las circunstancias en las que se puede generar la excepción.

Hay muchas otras excepciones para explorar: simplemente no tenemos el espacio para revisarlas todas aquí.

ArithmeticError

Ubicación: BaseException ← Exception ← ArithmeticError

Descripción: una excepción abstracta que incluye todas las excepciones causadas por operaciones aritméticas como división cero o dominio inválido de un argumento.

AssertionError

Ubicación: BaseException ← Exception ← AssertionError

Descripción: una excepción concreta generada por la instrucción assert cuando su argumento se evalúa a False (falso), None (ninguno), 0, o una cadena vacía.

```
from math import tan, radians
angle = int(input('Ingresa el angulo entero en grados: '))

# Debemos estar seguros de que angle != 90 + k * 180
assert angle % 180 != 90
print(tan(radians(angle)))
```

BaseException

Ubicación: BaseException

Descripción: la excepción más general (abstracta) de todas las excepciones de Python: todas las demás excepciones se incluyen en esta; se puede decir que las siguientes dos excepciones son equivalentes: except: y except BaseException:.

IndexError

Ubicación: BaseException ← Exception ← LookupError ← IndexError

Descripción: una excepción concreta que surge cuando se intenta acceder al elemento de una secuencia inexistente (por ejemplo, el elemento de una lista).

```
# El codigo muestra una forma extravagante
# de dejar el bucle.

the_list = [1, 2, 3, 4, 5]
ix = 0
do_it = True

while do_it:
    try:
        print(the_list[ix])
        ix += 1
    except IndexError:
        do_it = False

print('Listo')
```

KeyboardInterrupt

Ubicación: BaseException ← KeyboardInterrupt

Descripción: una excepción concreta que surge cuando el usuario usa un atajo de teclado diseñado para terminar la ejecución de un programa (Ctrl-C en la mayoría de los Sistemas Operativos); si manejar esta excepción no conduce a la terminación del programa, el programa continúa su ejecución.

Nota: esta excepción no se deriva de la clase Exception. Ejecuta el programa en IDLE.

```
# Este código no puede ser terminado
# presionando Ctrl-C.

from time import sleep

seconds = 0

while True:
    try:
        print(seconds)
        seconds += 1
        sleep(1)
    except KeyboardInterrupt:
        print("¡No hagas eso!")
```

LookupError

Ubicación: BaseException ← Exception ← LookupError

Descripción: una excepción abstracta que incluye todas las excepciones causadas por errores resultantes de referencias no válidas a diferentes colecciones (listas, diccionarios, tuplas, etc.).

MemoryError

Ubicación: BaseException ← Exception ← MemoryError

Descripción: se genera una excepción concreta cuando no se puede completar una operación debido a la falta de memoria libre.

```
# Este código causa la excepción MemoryError.
# Advertencia: el ejecutar este código puede afectar tu Sistema Operativo.
# ¡No lo ejecutes en entornos de producción!

string = 'x'
try:
    while True:
        string = string + string
        print(len(string))
except MemoryError:
    print('¡Esto no es gracioso!')
```

OverflowError

Ubicación: BaseException ← Exception ← ArithmeticError ← OverflowError

Descripción: una excepción concreta que surge cuando una operación produce un número demasiado grande para ser almacenado con éxito.

```
# El código imprime los valores subsequentes
# de exp(k), k = 1, 2, 4, 8, 16, ...

from math import exp

ex = 1

try:
    while True:
        print(exp(ex))
        ex *= 2
except OverflowError:
    print('El número es demasiado grande.')
```

ImportError

Ubicación: BaseException ← Exception ← StandardError ← ImportError

Descripción: se genera una excepción concreta cuando falla una operación de importación.

```
# Una de estas importaciones fallará, ¿cuál será?

try:
    import math
    import time
    import abracadabra

except:
    print('Una de tus importaciones ha fallado.')
```

KeyError

Ubicación: BaseException ← Exception ← LookupError ← KeyError

Descripción: una excepción concreta que se genera cuando intentas acceder al elemento inexistente de una colección (por ejemplo, el elemento de un diccionario).

```
# ¿Cómo abusar del diccionario
# y cómo lidiar con ello?

dictionary = {'a': 'b', 'b': 'c', 'c': 'd'}
ch = 'a'

try:
    while True:
        ch = dictionary[ch]
```

```
        print(ch)
except KeyError:
    print('No existe tal clave:', ch)
```

Hemos terminado con excepciones por ahora, pero volverán cuando discutamos la programación orientada a objetos en Python. Puedes usarlas para proteger tu código de accidentes graves, pero también tienes que aprender a sumergirte en ellas, explorando la información que llevan.

De hecho, las excepciones son objetos; sin embargo, no podemos decirle nada sobre este aspecto hasta que te presentemos clases, objetos y similares.

Por el momento, si deseas obtener más información sobre las excepciones por tu cuenta, consulta la Biblioteca Estándar de Python en https://docs.python.org/3.6/library/exceptions.html.

Puntos Clave

1. Algunas excepciones integradas abstractas de Python son:

- ArithmeticError.
- BaseException.
- LookupError.

2. Algunas excepciones integradas concretas de Python son:

- AssertionError.
- ImportError.
- IndexError.
- KeyboardInterrupt.
- KeyError.
- MemoryError.
- OverflowError.