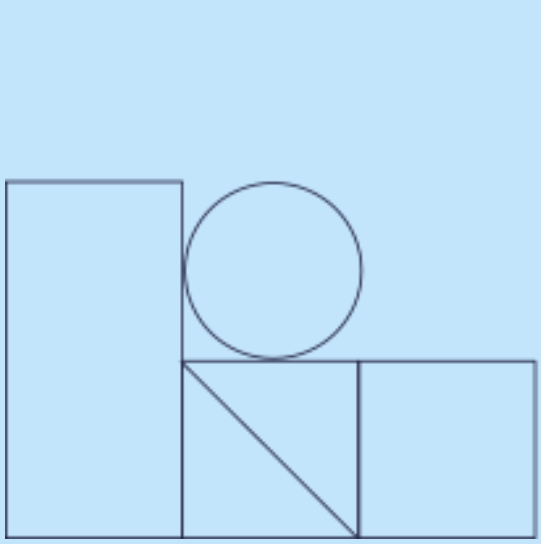
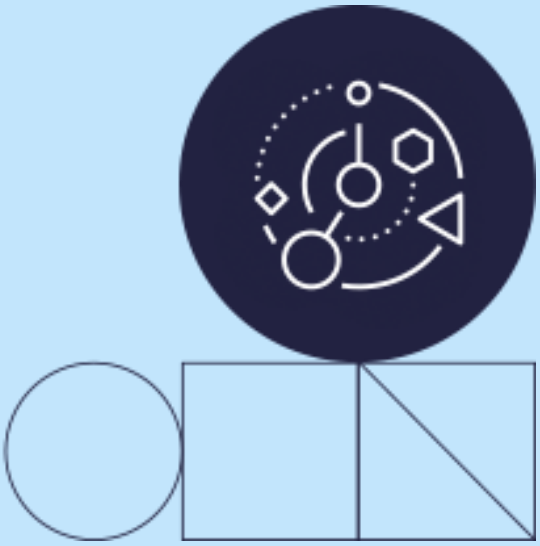
IBM SkillsBuild

IBM

Conceptos básicos y sintaxis de Python

Excepciones





Índice

Introducción

Las excepciones son eventos que permiten controlar el flujo de un programa cuando ocurre un error. Pueden dispararse automáticamente, cuando ocurre un error y bajo demanda en nuestro código. Es posible capturar una excepción para corregir el error que la ha originado o escalar la misma hacia arriba para que la intercepte otro código de más alto nivel.

Los roles de las excepciones en Python

- **Manejo de errores:** Se utilizan para informar de errores y/o de una situación anómala así como de detener el flujo de programa.
- **Notificación de eventos:** P. ej. terminar una búsqueda sin resultados sin tener que usar variables de control.
- **Manejo de casos especiales:** Podemos dejar el manejo de algunas situaciones especiales que ocurren con poca frecuencia a excepciones.
- **Acciones de limpieza/finalización:** Operaciones de limpieza que se ejecutan tanto como si ha habido errores como si no y que nos ayudan a asegurarnos que este tipo de operaciones ocurren siempre, independientemente de que haya habido un error o no. Esto es útil para asegurarnos que cerramos una conexión, un fichero, etc.

En Python disponemos de 4 sentencias que podemos utilizar para manejar excepciones:

- **try/except:** Intercepta y recupera excepciones disparadas por Python o por nuestro código.
- **try/finally:** Realiza tareas de limpieza ocurran las excepciones o no.
- **raise:** Dispara una excepción manualmente en el código.
- **assert:** Dispara una excepción condicionalmente.

En esta unidad veremos cuándo utilizar cada una de estas sentencias, así como ejemplos de las mismas.

Nuestra primera excepción

A continuación, vamos a crear un código que generará una excepción si se le introduce un parámetro adecuado.

Primero creamos una función, que devuelve el elemento que le pidamos de una secuencia según el índice que le pidamos:

```
def indexador(objeto, indice):
    return objeto[indice]

indexador('Python', 0)
```

Resultado:

```
✓ def indexador(objeto, indice): ···

'P'
```

Como vemos, si le pedimos un índice que existe en la secuencia nos devuelve el elemento alojado en ese índice. En cambio, si pedimos un índice demasiado grande, veremos que obtenemos un error de tipo ***IndexError.***

```
def indexador(objeto, indice):
    return objeto[indice]

indexador('Python', 10) #Produce un error
```

Resultado:

```
⊗ def indexador(objeto, indice): ···

---------------------------------------------------
------------------------
IndexError                                Tracebac
k (most recent call last)
j:\WORKSPACE\10 PYTHON\temp.py in line 4
      3 def indexador(objeto, indice):
      4     return objeto[indice]
----> 6 indexador('Python', 10)

j:\WORKSPACE\10 PYTHON\temp.py in line 2, in index
ador(objeto, indice)
      3 def indexador(objeto, indice):
----> 4     return objeto[indice]

IndexError: string index out of range
```

Notad que en otras unidades hemos simplificado las salidas de las excepciones, en este no lo haremos para que podáis ver toda la información relativa al error que ha ocasionado la excepción.

Por ejemplo, en la llamada que acabamos de hacer, el intérprete ha lanzado la excepción, ha detenido el programa, y además nos ha mostrado información a la excepción:

1. Primero muestra el tipo de la excepción: ***IndexError***
2. Luego nos muestra la traza de la misma, es decir la cadena de llamadas que la ha producido y la línea de código donde se ha producido
3. Finalmente, el intérprete nos muestra de nuevo el tipo de excepción y una cadena de información describiendo el error: *'IndexError: string index out of range'*. En este caso nos explica que el índice de la cadena está fuera de rango.

Notad que diferentes intérpretes pueden daros la información relativa a una excepción con un formato ligeramente diferente, pero en esencia, casi siempre muestran los tres campos que acabamos de mencionar.

Capturando excepciones

Para capturar excepciones utilizamos el bloque de sentencias ***try/except:***

```
def indexador(objeto, indice):
    return objeto[indice]

try:
    indexador('Python', 10)
except IndexError:      # Captura
Indexerror
    print('Has puesto un índice muy
grande.')
print('Hemos salido del try-catch')
```

Resultado:

```
✓ def indexador(objeto, indice): ···

Has puesto un índice muy grande.
Hemos salido del try-catch
```

En el bloque ***try*** se incluye el código propenso a causar la Excepción que queremos capturar en la sentencia ***except.*** La sentencia ***except*** está compuesta por la palabra clave que da nombre a la sentencia junto con la clase de la excepción que queremos capturar.

Por ejemplo, ***IndexError*** es una excepción que ocurre cuando intentamos acceder al índice de una secuencia y este índice está fuera de rango.

Dentro del bloque ***except*** incluimos el código necesario para manejar la situación cuando capturamos la excepción. En el ejemplo, simplemente estamos notificando en pantalla que el usuario ha puesto un índice fuera de rango. Notad que sólo entramos en el ***except*** en caso que salte excepción ***IndexError:***

```
def indexador(objeto, indice):
    return objeto[indice]

try:
    print(indexador('Python', 3))
except IndexError:      # Captura
Indexerror
    print('Has puesto un índice muy
grande.')
print('Hemos salido del try-catch')
```

Resultado:

```
✓ def indexador(objeto, indice): ···

h
Hemos salido del try-catch
```

Un aspecto que es importante es que el ***except*** sólo captura las excepciones indicadas en la sentencia. Es decir, si dentro del ***try*** se produce una excepción no contemplada en el ***except*****,** no seremos capaces de capturarla. Eso sí, es posible capturar varios tipos de excepciones a la vez:

```
def indexador(objeto, indice):
    return objeto[indice]

try:
    indexador('Python', 'h')
except (IndexError, TypeError):      #
Captura varios errores
    print('Error.')
print('Hemos salido del try-catch')

try:
    indexador('Python', 'h')
except:      # Captura todos los errores.
No recomendado.
    print('Error.')
print('Hemos salido del try-catch')
```

Resultado:

```
✓ def indexador(objeto, indice): ···

Error.
Hemos salido del try-catch
Error.
Hemos salido del try-catch
```

En este código estamos mostrando dos maneras de interceptar varias excepciones. La primera es la que nosotros recomendamos y consiste en un listado (una tupla) de Excepciones que pueden saltar en nuestro bloque ***try*****.** Cuando salte cualquiera de las Excepciones indicadas en nuestra tupla, entraremos en el bloque ***except*****.**

La segunda opción, es válida sintácticamente en Python, pero está menos recomendada porque captura cualquier excepción. Esto es peligroso porque podríamos estar silenciando excepciones no previstas por nosotros, por lo que estaríamos ocultando errores que luego serán difíciles de encontrar. Utilizad esta segunda opción con mucho cuidado y conociendo el riesgo al que os exponéis.

Una limitación de las dos aproximaciones anteriores es que, a pesar de que somos capaces de interceptar las excepciones indicadas, las estamos tratando indistintamente. Es decir, que vamos a tratar la excepción capturada de la misma manera independientemente que fuera una u otra. Si queremos hacer un tratamiento especial a un tipo de excepciones, lo podemos hacer encadenando bloques ***except*** uno detrás de otro, de manera similar a bloques ***elif*** en los condicionales:

```
def indexador(objeto, indice):
    return objeto[indice]

try:
    indexador('Python', 'h')
except IndexError:      # Captura
IndexError
    print('Error de índice.')
except TypeError:      # Captura
TypeError
    print('El índice tiene que ser un
número.')

print('Hemos salido del try-catch')
```

Resultado:

```
✓ def indexador(objeto, indice): ···

El índice tiene que ser un número.
Hemos salido del try-catch
```

En este caso, hemos llamado a ***indexador*** produciendo una excepción de tipo ***TypeError.*** Estas excepciones saltan cuando intentamos hacer una operación no admitida por el tipo que estamos utilizando. En este caso hemos intentado acceder al índice de una cadena de texto con otra cadena de texto en lugar de un número, por lo que hemos producido una excepción de tipo.

Lanzando excepciones manualmente

Podemos lanzar excepciones directamente en nuestro código utilizando la sentencia ***raise*** seguida del tipo de excepción que queremos lanzar. Por ejemplo:

```
raise IndexError
```

Resultado:

```
⊗ raise IndexError ···

---------------------------------------------------
------------------------
IndexError                                Tracebac
k (most recent call last)
j:\WORKSPACE\10 PYTHON\temp.py in line 1
----> 5 raise IndexError

IndexError:
```

Aquí acabamos de lanzar nuestra propia excepción de tipo ***IndexError.*** Eso sí, en este caso, al ver la traza del error, no tenemos ninguna información que nos oriente cual ha podido ser la causa del error.

Si queremos añadir esa información, simplemente creamos una instancia de ***IndexError*** (o de la excepción que queramos lanzar) y en su constructor añadimos el mensaje a mostrar:

```
raise IndexError('Excepción lanzada
manualmente')
```

Resultado:

```
⊗ raise IndexError('Excepción lanzada              ···

---------------------------------------------------
------------------------
IndexError                                Tracebac
k (most recent call last)
j:\WORKSPACE\10 PYTHON\temp.py in line 1
----> 4 raise IndexError('Excepción lanzada manual
mente')

IndexError: Excepción lanzada manualmente
```

Ahora sí, vemos que en la última línea del error tenemos el mensaje explicativo del error.

Por supuesto, es posible capturar nuestras propias excepciones lanzadas manualmente:

```
try:
    raise IndexError('Excepción lanzada
manualmente')
except:
    print('He capturado mi pripia
excepción')
```

Resultado:

```
✓ try: ···

He capturado mi pripia excepción
```

La sentencia 'assert'

Además de lanzar excepciones manualmente, es posible hacerlo condicionalmente. Para ello, Python proporciona la sentencia ***assert*** que nos permite lanzar una excepción si se cumple una condición determinada.

Es muy común utilizar la sentencia ***assert*** durante el proceso de depuración, para asegurarnos que se cumplen ciertas condiciones previas.

La sintaxis de ***assert*** es la siguiente:

assert(condición), 'Mensaje de error'

Veamos un ejemplo:

```
a = 10
b = 0
assert(a > b), 'A tiene que ser mayor que B!'   # Si se cumple no salta el error

print('Si se muestra esto es que no ha saltado el assert')
```

Resultado:

```
✓ a = 10 ···

Si se muestra esto es que no ha saltado el assert
```

En este caso, como se cumple la condición, no salta el ***assert,*** y el programa sigue ejecutándose normalmente. Veamos un caso en el que sí salta la excepción producida por el ***assert:***

```
a = 10
b = 0
c = 5
assert(a == c), 'A y C tienen que ser iguales!'
```

Resultado:

```
⊗ a = 10 ···

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
j:\WORKSPACE\10 PYTHON\temp.py in line 4
      5 b = 0
      6 c = 5
----> 7 assert(a == c), 'A y C tienen que ser iguales!'

AssertionError: A y C tienen que ser iguales!
```

Una nota importante sobre esta la sentencia ***assert:*** su uso es muy útil para detectar errores en depuración, pero no se recomienda el uso de ***assert*** en producción.

Creando nuestras propias excepciones

Hasta ahora hemos visto como capturar y lanzar excepciones incluidas en la librería estándar de Python. Sin embargo, en muchos casos es de gran utilidad crear nuestras propias excepciones.

Si no os habíais fijado hasta ahora, las excepciones son clases. Por eso, si queremos crear nuestra propia excepción, sólo tenemos que crear una clase que herede de la clase base de todas las excepciones ***(Exception)*** o de cualquier otra excepción:

```
class miPropiaExcepcion(Exception): #Las excepciones heredan de Exception
    pass

raise miPropiaExcepcion
```

Resultado:

```
⊗ class miPropiaExcepcion(Exception): #Las
  ------------------------------------------------
  ------------------------
  miPropiaExcepcion                          Tracebac
  k (most recent call last)
  j:\WORKSPACE\10 PYTHON\temp.py in line 4
        4 class miPropiaExcepcion(Exception): #Las e
  xcepciones heredan de Exception
        5     pass
  ----> 7 raise miPropiaExcepcion

  miPropiaExcepcion:
```

Acabamos de crear la clase ***MiPropiaExcepcion*** que hereda de ***Exception***. Cuando una clase hereda de ***Exception***, podemos incluirla dentro de una sentencia ***raise*** para lanzarla, así como dentro de un ***except*** para interceptarla:

```
class miPropiaExcepcion(Exception): #Las
excepciones heredan de Exception
    pass

try:
    raise miPropiaExcepcion

except miPropiaExcepcion:
    print('He capturado mi propia
excepción')
```

Resultado:

```
✓ class miPropiaExcepcion(Exception): #Las
He capturado mi propia excepción
```

Pero nuestra excepción es muy básica. Vamos a mejorarla un poco para que pueda representar su propio mensaje de error:

```
class miError(Exception):

    def __init__(self, valor):
        self.valor = valor

    def __str__(self):
        return str(self.valor)

raise(miError('Mensaje de error'))
```

Resultado:

```
⊗ class miError(Exception): ···
  ------------------------------------------------
  ------------------------
  miError                                    Tracebac
  k (most recent call last)
  j:\WORKSPACE\10 PYTHON\temp.py in line 7
        7     def __str__(self):
        8         return str(self.valor)
  ----> 10 raise(miError('Mensaje de error'))

  miError: Mensaje de error
```

En este ejemplo hemos creado un constructor de nuestra excepción que utilizamos para almacenar un objeto que luego pasaremos al método ***__str__***. Este método es un método especial de Python, llamado "método mágico". En concreto, ***__str__*** define como hay que representar una clase si se la quiere mostrar como un string (una cadena de texto), por ejemplo, para introducirla en un ***print***, etc.

En este caso, el método ***__str__*** permite mostrar el mensaje de error que queramos al lanzar nuestra excepción.

Acciones de finalización y limpieza

Cuando tenemos excepciones, hay situaciones en las que queremos hacer operaciones de limpieza o finalización sin importar si la excepción ha saltado o no. Este tipo de operaciones suelen ser, por ejemplo, asegurarnos que cerramos un fichero, una conexión, etc.

Para esto tenemos la sentencia ***finally:***

```
def indexador(objeto, indice):
    return objeto[indice]

try:
    indexador('Python', 10)
finally:
    print('Esto se ejecuta incluso cuando
salta la excepción')
```

Resultado:

```
⊗ def indexador(objeto, indice): ···

Esto se ejecuta incluso cuando salta la excepción

  ---------------------------------------------------
  -------------------------
  IndexError                                Tracebac
  k (most recent call last)
  j:\WORKSPACE\10 PYTHON\temp.py in line 5
        5     return objeto[indice]
        7 try:
  ----> 8     indexador('Python', 10)
        9 finally:
       10     print('Esto se ejecuta incluso cuando
  salta la excepción')

  j:\WORKSPACE\10 PYTHON\temp.py in line 2, in index
  ador(objeto, indice)
        4 def indexador(objeto, indice):
  ----> 5     return objeto[indice]

  IndexError: string index out of range
```

En este código hemos llamado a ***indexador*** de manera errónea y hemos producido una excepción. Normalmente, cuando esto ocurre, se detiene el flujo de programa. En este caso, al tener un bloque ***finally*** lo que ocurre es que, justo antes de detenerse el flujo de programa, ejecutamos el código que esté incluido en nuestro bloque ***finally*.**

Notad que el código siguiente, se le parece, pero **NO** se ejecutará:

```
def indexador(objeto, indice):
    return objeto[indice]

try:
    indexador('Python', 10)
    print('Este print no se ejecuta')
finally:
    print('Esto se ejecuta incluso cuando
salta la excepción')
```

En este caso vemos que el ***print*** no se ha ejecutado ya que antes ha saltado la excepción y, por lo tanto, se ha detenido el flujo de ejecución del programa.

Notad también que el ***finally*** se ejecuta siempre, salte o no salte la excepción, pero si la excepción ha saltado, el flujo de programa se detiene justo después del ***finally*.**

```
def indexador(objeto, indice):
    return objeto[indice]

try:
    indexador('Python', 10)
finally:
    print('Esto se ejecuta incluso cuando
salta la excepción')
print('Este print tampoco se ejecuta')
```

Resultado:

```
⊗ def indexador(objeto, indice): ···

Esto se ejecuta incluso cuando salta la excepción

  ---------------------------------------------------
  -------------------------
  IndexError                                Tracebac
  k (most recent call last)
  j:\WORKSPACE\10 PYTHON\temp.py in line 5
        5     return objeto[indice]
        7 try:
  ----> 8     indexador('Python', 10)
        9 finally:
       10     print('Esto se ejecuta incluso cuando
  salta la excepción')

  j:\WORKSPACE\10 PYTHON\temp.py in line 2, in index
  ador(objeto, indice)
        4 def indexador(objeto, indice):
  ----> 5     return objeto[indice]

  IndexError: string index out of range
```

Es decir, que el ***finally*** sólo asegura que el código de su bloque se va a ejecutar, pero impide que el flujo de programa se detenga. Para eso, recordad que tenemos el bloque ***except*:**

```
def indexador(objeto, indice):
    return objeto[indice]
try:
    indexador('Python', 10)
except IndexError:
    print('Capturamos la excepción')
finally:
    print('Esto se ejecuta incluso cuando
salta la excepción')
print('Se ejecutará este print?')
```

Resultado:

```
✓ def indexador(objeto, indice): ···

Capturamos la excepción
Esto se ejecuta incluso cuando salta la excepción
Se ejecutará este print?
```

En este caso, ***indexador*** produce una excepción, que capturamos en el bloque ***except,*** por lo que ejecutamos el código dentro de ese bloque. Luego, ejecutamos el código ***finally*** y, tras ello, como ejecutamos el código fuera de nuestro bloque ***try/except/finally.***

El bloque 'else' en las excepciones

La última sentencia útil en el uso de excepciones es la sentencia ***else.*** En el caso de excepciones, la sentencia ***else*** se comporta de manera muy parecida a cómo lo hacía al ponerla tras un bucle: ejecuta el código de su bloque sólo si *NO salta la excepción* en el bloque ***try/except:***

```
def divide(x, y):
    try:
        resultado = x/y
    except ZeroDivisionError:
        print('No se puede dividir por
cero')
    else:
        print('El resultado es: ')
    finally:
        print('Ejecutamos el finally')


divide(4, 12)

divide(4, 0)
```

Resultado:

```
✓ def divide(x, y): ···

El resultado es:
Ejecutamos el finally
No se puede dividir por cero
Ejecutamos el finally
```

En este ejemplo intentamos hacer una división, y controlamos dentro de un bloque ***try/except*** si hemos intentado hacer una división por 0. Si el usuario intenta dividir por 0, capturamos la excepción en el ***except.*** Si la operación es correcta, entonces mostramos el resultado en el bloque ***else.***

La ventaja del bloque ***else*** nos ahorra tener que evaluar si tenemos resultado o no (podríamos no haber obtenido un resultado en el caso de división por cero).