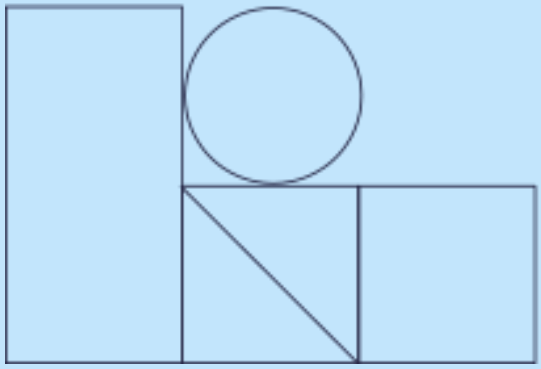
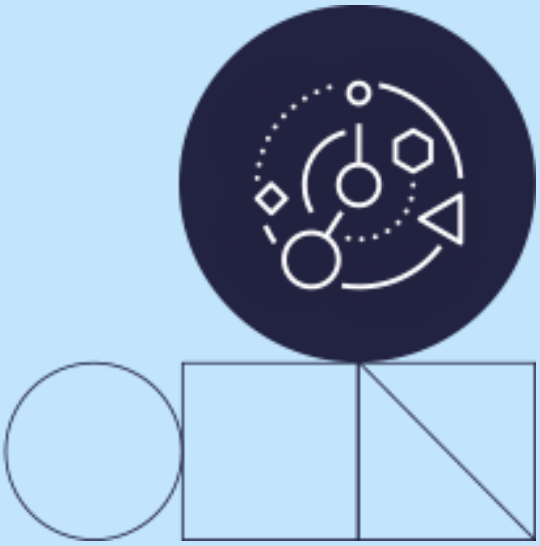
IBM SkillsBuild

IBM

Noções básicas e sintaxe de Python

Exceções





Índice

Introdução

As exceções são eventos que permitem controlar o fluxo de um programa quando ocorre um erro. Eles podem ser acionados automaticamente, quando ocorre um erro e sob demanda em nosso código. É possível pegar uma exceção para corrigir o erro que o causou ou escalá-lo para cima para ser interceptado por um código de nível superior.

Os papéis das exceções em Python

- **Tratamento de erros:** Eles são usados para relatar erros e/ou uma situação anômala, assim como para interromper o fluxo do programa.
- **Notificação de eventos:** P. ex. terminar uma busca sem sucesso sem ter que usar variáveis de controle.
- **Tratamento de casos especiais:** Podemos deixar o tratamento de algumas situações especiais que ocorrem com pouca freqüência a exceções.
- **Ações de limpeza/conclusão:** Operações de limpeza que são executadas quer tenha havido ou não erros, e que nos ajudam a garantir que estes tipos de operações sempre ocorram, independentemente de ter havido ou não um erro. Isto é útil para garantir que fechamos uma conexão, um arquivo, etc.

Em Python temos 4 declarações que podemos usar para lidar com exceções:

- **try/except:** Intercepta e recupera exceções acionadas por Python ou por nosso código.
- **try/finally:** Realiza tarefas de limpeza, quer ocorram ou não exceções.
- **raise:** Aciona uma exceção manualmente no código.
- **assert:** Desencadeia uma exceção condicionalmente.

Nesta unidade, veremos quando usar cada uma destas frases, bem como exemplos delas.

Nossa primeira exceção

Em seguida, vamos criar um código que irá gerar uma exceção se um parâmetro apropriado for inserido.

Primeiro criamos uma função, que retorna o elemento solicitado de uma seqüência de acordo com o índice que solicitamos:

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

indexador('Python', 0)
```

Resultado:

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

'P'
```

Como podemos ver, se pedirmos um índice que existe na seqüência, ele retorna o elemento nesse índice. Por outro lado, se pedirmos um índice muito grande, obtemos um erro 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
```

Note que em outras unidades simplificamos a saída das exceções, nesta não o faremos para que você possa ver todas as informações relacionadas com o erro que causou a exceção.

Por exemplo, na chamada que acabamos de fazer, o intérprete lançou a exceção, parou o programa, e também nos mostrou informações sobre a exceção:

1. Primeiro mostrar o tipo de exceção: ***IndexError***
2. Depois nos mostra o traço da mesma, ou seja, a cadeia de chamadas que a produziu e a linha de código onde foi produzida.
3. Finalmente, o intérprete mostra novamente o tipo de exceção e uma seqüência de informações descrevendo o erro: '*IndexError: string index out of range*'. Neste caso, explica que o índice da corda está fora de alcance.

Observe que intérpretes diferentes podem lhe dar as informações relativas a uma exceção em um formato ligeiramente diferente, mas, em essência, eles quase sempre exibem os três campos que acabamos de mencionar.

Capturando exceções

Para captar exceções, usamos a declaração de bloco ***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
```

No bloco ***try*** o código que provavelmente causará a Exceção que queremos capturar está incluído na sentença ***except***. A declaração ***except*** é composta da palavra-chave que dá nome à declaração junto com a classe da exceção que queremos pegar.

Por exemplo, ***IndexError*** é uma exceção que ocorre quando tentamos acessar o índice de uma seqüência e este índice está fora de alcance.

Dentro do bloco ***except*** incluímos o código necessário para lidar com a situação quando pegarmos a exceção. No exemplo, estamos simplesmente notificando a tela que o usuário estabeleceu um índice fora da faixa de variação. Note que nós só entramos no ***except*** no caso de uma exceção ***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
```

Um aspecto que é importante é que o ***except*** só captura as exceções indicadas no julgamento. Isto é, se dentro do ***try*** há uma exceção não prevista no ***except*****,** não seremos capazes de pegá-lo. Entretanto, é possível pegar vários tipos de exceções ao mesmo tempo:

```
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
```

Neste código, estamos mostrando duas maneiras de interceptar várias exceções. A primeira é a que recomendamos e consiste de uma lista (uma tupla) de Exceções que podem ser jogadas em nosso bloco ***try*****.** Quando qualquer uma das Exceções indicadas em nosso tuple é levantada, entramos no bloco ***except*****.**

A segunda opção é sintaticamente válida em Python, mas é menos recomendada porque captura qualquer exceção. Isto é perigoso porque poderíamos estar silenciando exceções não previstas por nós, então estaríamos escondendo erros que mais tarde serão difíceis de encontrar. Use esta segunda opção com muito cuidado e esteja ciente do risco ao qual você está se expondo.

Uma limitação das duas abordagens anteriores é que, embora sejamos capazes de interceptar as exceções indicadas, estamos tratando-as indistintamente. Ou seja, vamos tratar a exceção capturada da mesma forma, independentemente de ter sido uma ou outra. Se quisermos fazer um tratamento especial para um tipo de exceção, podemos fazer isso encadeando blocos ***except*** um após o outro, de forma semelhante aos blocos ***elif*** nos condicionadores:

```
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
```

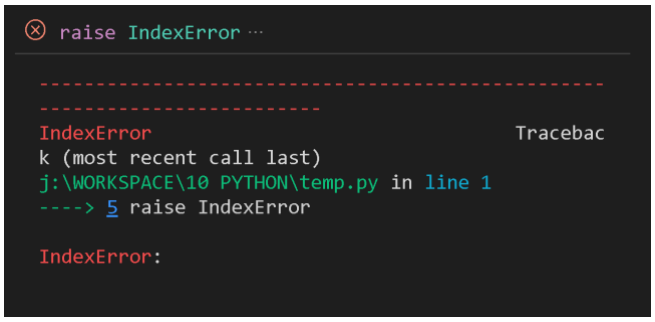
Neste caso, chamamos a ***indexador*** produzindo uma exceção do tipo ***TypeError.*** Estas exceções são lançadas quando tentamos fazer uma operação não suportada pelo tipo que estamos utilizando. Neste caso, tentamos acessar o índice de uma cadeia de texto com outra cadeia de texto em vez de um número, por isso produzimos uma exceção do tipo.

Lançando exceções manualmente

Podemos lançar exceções diretamente em nosso código usando a declaração ***raise*** seguida do tipo de exceção que queremos lançar. Por exemplo:

```
raise IndexError
```

Resultado:

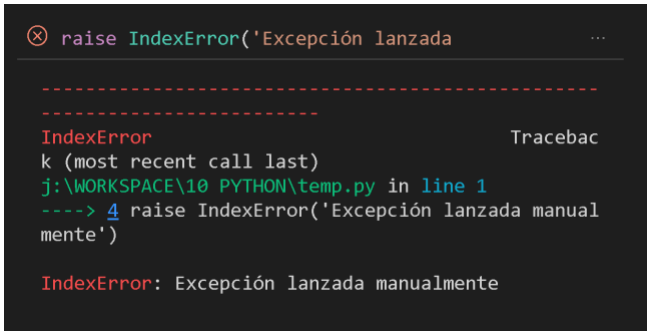


Aqui nós acabamos de lançar nossa própria exceção de tipo ***IndexError.*** Entretanto, neste caso, quando vemos o rastro do erro, não temos informações que nos orientem quanto ao que poderia ter causado o erro.

Se quisermos acrescentar essas informações, simplesmente criamos uma instância de ***IndexError*** (ou da exceção que queremos lançar) e em seu construtor adicionamos a mensagem a ser exibida:

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

Resultado:

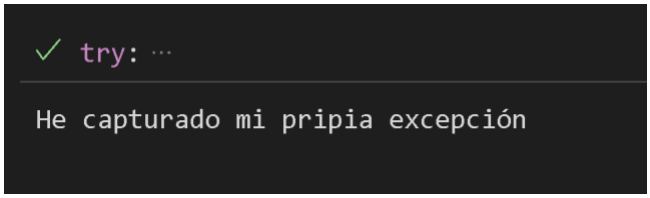


Agora vemos que, na última linha do erro, temos a mensagem que explica o erro.

Naturalmente, é possível pegar suas próprias exceções lançadas manualmente:

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

Resultado:



A declaração de `assert`

Além de atirar exceções manualmente, também é possível atirar exceções condicionalmente. Para fazer isso, Python fornece a declaração ***assert*** que nos permite lançar uma exceção se uma determinada condição for cumprida.

É muito comum usar a declaração ***assert*** durante o processo de depuração, para garantir que certas condições prévias sejam cumpridas.

A sintaxe de ***assert*** é a seguinte:

assert(condición), 'Mensaje de error'

Aqui está um exemplo:

```
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
```

Neste caso, como a condição é cumprida, não é acionada ***assert,*** e o programa continua a funcionar normalmente. Vejamos um caso em que a exceção produzida pela ***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!
```

Uma nota importante sobre esta declaração ***assert:*** seu uso é muito útil para detectar erros na depuração, mas não é recomendado el uso de ***assert*** na produção.

Criando nossas próprias exceções

Até agora, vimos como pegar e lançar exceções incluídas na biblioteca padrão Python. Entretanto, em muitos casos é muito útil criar suas próprias exceções.

Se você ainda não notou até agora, as exceções são as classes. Portanto, se quisermos criar nossa própria exceção, só precisamos criar uma classe que herde da classe base de todas as exceções. ***(Exception)*** ou qualquer outra exceção:

```
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 criar a classe ***MiPropiaExcepcion*** que herda de ***Exception***. Quando uma classe herda de ***Exception***, podemos incluí-la dentro de uma declaração ***raise*** para lançá-la, assim como dentro de um ***except*** para interceptá-la:

```
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
```

Mas nossa exceção é muito básica. Vamos melhorá-lo um pouco para que ele possa representar sua própria mensagem de erro:

```
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
```

Neste exemplo criamos um construtor para nossa exceção que usamos para armazenar um objeto que depois passaremos para o método ***__str__***. Este método é um método especial Python, chamado "método mágico". Especificamente, ***__str__*** define como uma classe deve ser representada se ela deve ser exibida como um string, por exemplo, para ser inserida em um ***print***, etc.

Neste caso, o método ***__str__*** permite que você exiba a mensagem de erro de sua escolha ao lançar sua exceção.

Ações de conclusão e limpeza

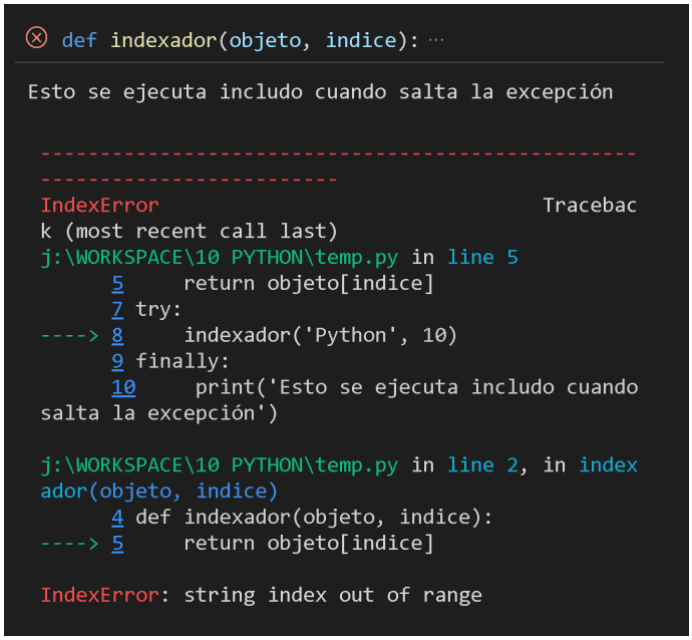
Quando temos exceções, há situações em que queremos realizar operações de limpeza ou encerramento, independentemente de a exceção ter sido levantada ou não. Estes tipos de operações são normalmente, por exemplo, para garantir que fechamos um arquivo, uma conexão, etc.

Para isso, temos a declaração ***finally:***

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

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

Resultado:



Neste código, chamamos a ***indexador*** de forma equivocada e produzimos uma exceção. Normalmente, quando isso acontece, o fluxo do programa é interrompido. Neste caso, ter um bloco ***finally*** o que acontece é que, pouco antes do fluxo do programa parar, nós executamos o código que está incluído em nosso bloco ***finally*.**

Note que o seguinte código, assemelha-se a ele, mas **NÃO** será executado:

```
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')
```

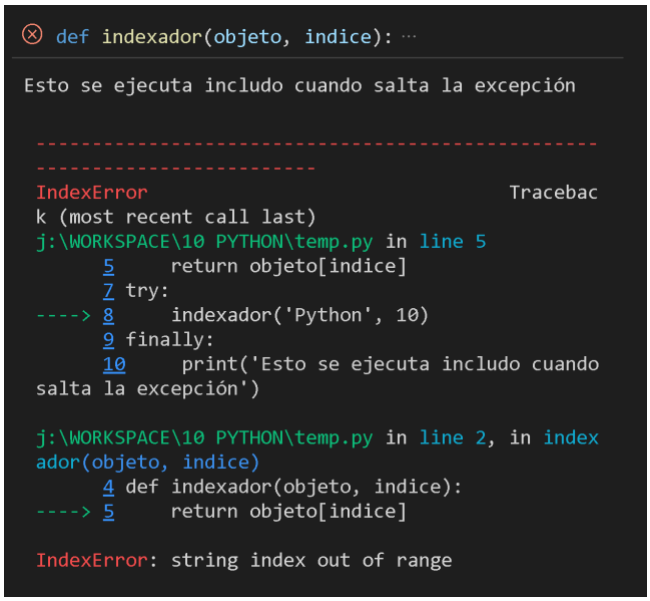
Neste caso, vemos que o ***print*** não foi executado porque a exceção já foi levantada antes e, portanto, o fluxo de execução do programa foi interrompido.

Note também que o ***finally*** é sempre executado, quer a exceção seja levantada ou não, mas se a exceção for levantada, o fluxo do programa pára logo após o ***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:



Em outras palavras, que o ***finally*** apenas garante que o código em seu bloco será executado, mas evita que o fluxo do programa pare. Para isso, lembre-se que temos o bloco ***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?
```

Neste caso, ***indexador*** produz uma exceção, que apanhamos no bloco ***except***, por isso, executamos o código dentro daquele bloco. Então, executamos o código ***finally*** e, depois disso, enquanto executamos o código fora de nosso bloco ***try/except/finally***.

O bloco 'else' em exceções

A última declaração útil no uso de exceções é a declaração ***else***. No caso de exceções, a declaração ***else*** comporta-se da mesma forma que quando colocado após um loop: ele executa o código em seu bloco somente se a exceção *NÃO for levantada* no bloco ***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
```

Neste exemplo, tentamos fazer uma divisão, e controlamos dentro de um bloco ***try/except*** se tivermos tentado fazer uma divisão por 0. Se o usuário tentar dividir por 0, nós pegamos a exceção no ***except***. Se a operação estiver correta, então exibimos o resultado no bloco ***else***.

A vantagem do bloco ***else*** nos poupa de ter que avaliar se temos ou não um resultado (podemos não ter obtido um resultado no caso de divisão por zero).