Tratamento de exceções

Durante o nosso curso, vimos algumas situações nas quais, por conta de alguma falta de atenção ou algum problema no nosso código, o Python não soube lidar com o processamento e exibiu uma mensagem de erro no console.

Temos pelo menos dois tipos de erros presentes no Python: erros de sintaxe e exceções.

Erros de sintaxe

Os erros de sintaxe, ou erros de parse, são muito comuns entre aqueles que ainda estão aprendendo a linguagem, e ocasionalmente pessoas mais experientes também terminam se deparando com esse tipo de erro.

Isso acontece quando o interpretador Python identifica um código que não está aderente à sintaxe da linguagem, ou seja, apresenta alguma inconsistência na estrutura.

Alguns tipos comuns de erros de sintaxe:

- Ausência de aspas fechando uma string;
- Ausência de parênteses/colchetes/chaves fechando uma função/lista/dicionário;
- Ausência de dois pontos ao final das instruções de decisão (`if`) e repetição (`while`, `for`);
- Inclusão de parâmetros padrão antes de parâmetros posicionais na declaração de funções;
- Bloco de código (`if`, `while`, etc.) vazio;
- Uso incorreto do operador de atribuição (`=`) ou de igualdade (`==`);
- Erro de digitação em palavras-chave da linguagem (escrever `wihle` ao invés de `while`, por exemplo).

O interessante dos erros de sintaxe é que o próprio interpretador indica a área na qual foi identificado o erro. Observe no caso abaixo que um sinal `^` é indicado logo após o `True`. Essa é a forma que o interpretador indica que ali está faltando alguma coisa.

Dependendo do caso, é comum o interpretador indicar a instrução que vem **logo a seguir** ao erro de sintaxe. Então, caso não consiga identificar um problema de sintaxe na indicação da seta, procure pela instrução imediatamente anterior.

```
while True
  print("olá, mundo!")
```

```
  File "<ipython-input-1-2dce8ce87c57>", line 1
    while True
              ^
```

Exceções

Mesmo que um código esteja sintaticamente correto, ainda é possível surgir um erro. Esse erro normalmente ocorre quando o interpretador não consegue executar alguma instrução. Esse tipo de erro é chamado de **exceção**.

Uma exceção pode ser fatal para o processamento do programa. Quando o programa encerra por conta de uma exceção, uma mensagem de erro é apresentada no console que executou o script, como no exemplo abaixo:

```
print("abc" + 123)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-44703628bfed> in <module>()
----> 1 print("abc" + 123)

TypeError: can only concatenate str (not "int") to str
```

SEARCH STACK OVERFLOW

A última linha da mensagem de erro indica o tipo de exceção que ocorreu. Existem inúmeros tipos, e é interessante analisar as exceções pré-definidas do Python, além das exceções de bibliotecas externas que porventura você esteja usando. Além do tipo de exceção, a última linha também apresenta uma breve descrição do problema, o que facilita muito a correção.

Um outro recurso que facilita a correção é o **traceback**. Isto nada mais é do que um rastreio que identifica o caminho pelo qual o interpretador chegou no ponto que causa o erro.

Ter esse recurso é essencial em projetos grandes, onde uma função pode ser chamada por diversos módulos no código. Com o traceback, você consegue identificar se a passagem de parâmetros está correta, se há um problema com os tipos dos dados, etc.

No exemplo abaixo, veja que o traceback indica o caminho para chegar no erro, a partir da função mais externa, até a função onde de fato há o erro.

```
def func1():
  print(5 / 0)

def func2():
  func1()

def func3():
  func2()
```

```
func3()
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-a3b938f41792> in <module>()
      8   func2()
      9
---> 10 func3()

 2 frames
<ipython-input-3-a3b938f41792> in func3()
      6
      7 def func3():
----> 8   func2()
      9
     10 func3()

<ipython-input-3-a3b938f41792> in func2()
      3
      4 def func2():
----> 5   func1()
      6
      7 def func3():

<ipython-input-3-a3b938f41792> in func1()
      1 def func1():
----> 2   print(5 / 0)
      3
      4 def func2():
      5   func1()

ZeroDivisionError: division by zero
```

SEARCH STACK OVERFLOW

Tratamento de exceções

Idealmente, o nosso código não deve ser interrompido por nenhum fator externo, apenas por iniciativa do usuário ou pelo encerramento natural do seu processamento. Felizmente, existem recursos no Python que podemos utilizar para tratar a ocorrência de exceções antes mesmo de elas ocorrerem.

Para isso, a forma mais comum é usarmos o recurso `try/except`. Esse recurso possui a seguinte sintaxe:

```
try:
  <codigo>
except <excecao>:
  <codigo_em_caso_de_erro>
except <outra_excecao>:
  <codigo_em_caso_de_erro>
...
```

Veja o exemplo abaixo, no qual tratamos uma exceção de valor, quando o usuário passa um dado que não é inteiro:

```
while True:
  try:
    num = int(input("Insira um número: "))
    break
  except ValueError:
    print("Dado inválido! Tente novamente...")
```

```
Insira um número: texto
Dado inválido! Tente novamente...
Insira um número: exemplo
Dado inválido! Tente novamente...
Insira um número: abc
Dado inválido! Tente novamente...
Insira um número:
Dado inválido! Tente novamente...
Insira um número: 2
```

A instrução `try` funciona da seguinte maneira:

- Primeiramente, o código dentro do bloco `try` é executado linha a linha;
- Caso nenhuma exceção ocorra, os blocos `except` são ignorados e a instrução é finalizada;
- Caso uma exceção ocorra durante a execução do bloco `try`, o código deste bloco é interrompido e, caso aquela exceção tenha sido prevista em um bloco `except`, o interpretador vai para ele, onde executa todas as linhas e a instrução é, então, finalizada;
- Caso uma exceção ocorra no bloco `try` e não foi prevista em nenhum bloco `except`, o interpretador sobe a mensagem de erro no terminal.

Um bloco `except` pode conter vários tipos de exceções, que devem ser incluídos num formato de tupla:

```
while·True:
··try:
····num·=·int(input("Informe·um·número:·"))
····break
··except·(ValueError,·KeyboardInterrupt):
····print("Dado·inválido!·Tente·novamente!")
```

```
Informe um número: abc
Dado inválido! Tente novamente!
Informe um número: 12
```

Aqui vale uma observação: é possível iniciar um bloco `except` sem passar, necessariamente, o tipo de exceção que o programa deve capturar. Deve-se usar isso com muita cautela, já que, nessas situações, o interpretador Python vai receber toda e qualquer exceção e inserir no bloco `except`!

Então, mesmo que você só queira capturar exceções do tipo `ValueError`, o interpretador também vai capturar uma exceção do tipo `RuntimeError`, por exemplo. O PEP-8 não recomenda iniciar blocos `except` sem detalhar os erros associados ao bloco.

Os blocos `else` e `finally`

A instrução `else` pode ser usada para executar um código caso os blocos `except` não sejam executados. Apesar de parecer desnecessário, é interessante, do ponto de vista de organização do código, incluir dentro do bloco `try` apenas aquelas linhas que podem provocar a exceção. Assim fica fácil de saber onde a exceção surgiu.

Nosso exemplo poderia ficar, portanto:

```
while·True:
··try:
····num·=·int(input("Informe·um·número:·"))
··except·(ValueError,·KeyboardInterrupt):
····print("Dado·inválido!·Tente·novamente!")
··else:
····break
```

```
Informe um número: abc
Dado inválido! Tente novamente!
Informe um número: de
Dado inválido! Tente novamente!
Informe um número: 123
```

Já um bloco `finally` é incluído quando queremos executar uma ação independente de ter surgido a exceção ou não. Utilizamos este bloco para, por exemplo, fechar um arquivo aberto, imprimir na tela informações, etc.

Veja que, no exemplo, abaixo, a linha tracejada é inserida no resultado independente tanto quando a exceção é observada, quanto no caso em que ela não aparece.

```
while True:
  num = None
  try:
    num = int(input("Informe um número: "))
  except (ValueError, KeyboardInterrupt):
    print("Dado inválido! Tente novamente!")
  else:
    break
  finally:
```

```
  finally:
    print("-" * 15)
```

```
Informe um número: abc
Dado inválido! Tente novamente!
---------------
Informe um número: 123
---------------
```

A instrução `raise`

A instrução raise é interessante para os casos em que precisamos criar nossas próprias exceções.

Vamos considerar, por exemplo, um cenário em que temos uma função que itera sobre um tamanho de uma pilha de dados. Essa pilha de dados, no entanto, possui um limite de 100 entradas. Podemos criar uma exceção para indicar quando a pilha está cheia:

```
MAX_PILHA = 100

class TamanhoPilhaExcedido(Exception):
  pass

def incrementa_pilha(pilha, acrescimo):
  if len(pilha) + acrescimo > MAX_PILHA:
    raise TamanhoPilhaExcedido

  pilha.extend([0] * acrescimo)

def main():
  pilha = []

  while True:
    try:
      incrementa_pilha(pilha, 7)
    except TamanhoPilhaExcedido:
      break
    else:
      print(len(pilha))

main()
```

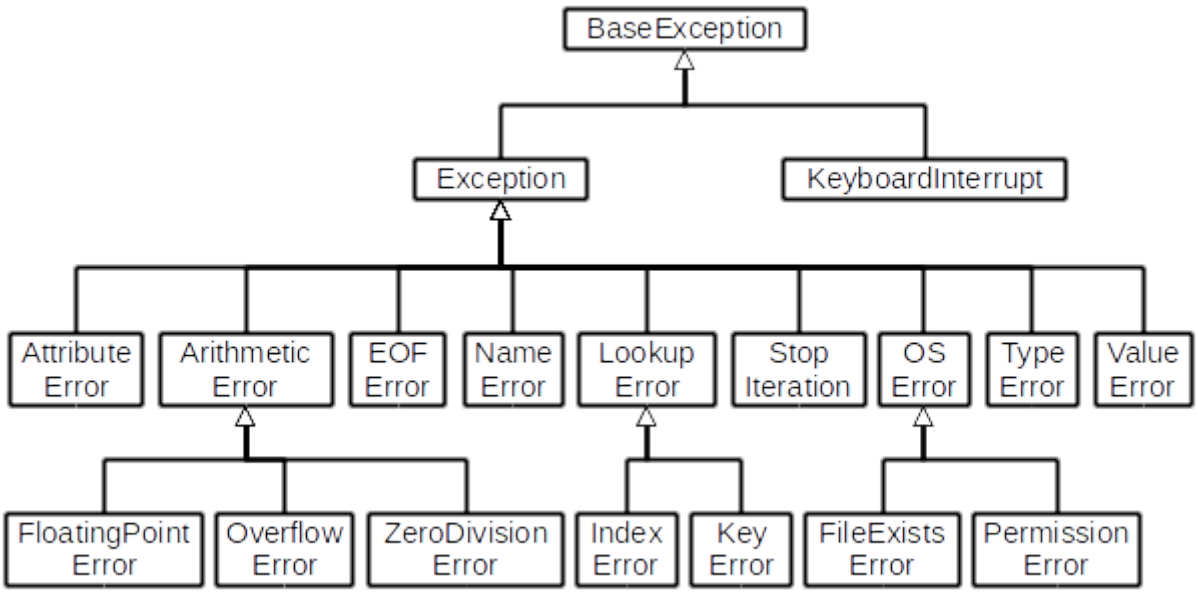
```
7
14
21
28
35
42
```

```
49
56
63
70
77
84
91
98
```

Hierarquia de exceções

Veja que, para criarmos uma nova exceção, precisamos definir uma classe `TamanhoPilhaExcedido`, que herda de uma outra classe `Exception`.

As exceções em Python são organizadas em uma hierarquia, na qual a classe `Exception` é superclasse de todas as outras (veja a figura abaixo para mais detalhes). Como não vamos incluir detalhes adicionais sobre essa exceção, não precisamos busca uma classe mais específica.



Observe que, no nosso exemplo, nós utilizamos o `raise` para subir uma exceção criada por nós, mas isso não é obrigatório. Poderíamos ter uma função que suba uma exceção do tipo `ValueError`, por exemplo.

✓ 0s conclusão: 14:28