Exercícios sobre tratamento de exceções em Python

Prof.: Carlos Camarão

14 de Outubro de 2018

Em computação, uma exceção é um evento que ocorre durante a execução de programas que pode ser tratado mas, se não tratado, ocasiona a terminação do programa.

O tratamento pode consistir simplesmente em emitir alguma mensagem contendo informação sobre o erro, ou pode envolver alguma recuperação da situação de erro, como tentar abordagem alternativa para o evento ocorrido.

A vantagem de uma linguagem prover suporte ao tratamento de exceções consiste não só em evitar a ocorrência de erros que provocam terminação anormal da execução de programas, mas permitir programas melhor estruturados, pela separação do código que define o comportamento usual do programa do código que realiza o tratamento de condições de erro (ou condições excepcionais).

Em Python, e em muitas linguagens de programação, o tratamento de exceções é feito usando o comando **try**, e uma exceção pode ser causada em um programa usando o comando **raise**.

Nota: em Haskell, devido à ordem não especificada na avaliação de expressões, exceções podem ser tratadas apenas na mônada *IO*. Exceções são definidas como tipos que são instâncias da classe *Exception*. O tratamento de exceções em Haskell não necessita de sintaxe especial na lingugem, sendo *try* simplesmente uma função (que recebe ação monádica), definida em *Control.Exception*. Uma abordagem semelhante é usada em implementações de Prolog (como por exemplo SWI-Prolog), com uso de predicados em vez de funções.

Um comando **try** tem a seguinte forma:

```
try:
    c1
except Nome:
    c2
```

Por exemplo:

```
while True:
    try:
        x = int(input("Digite um numero inteiro: "))
            break
    except ValueError:
        print("Numero inteiro esperado. Tente novamente.")
```

O tratamento de exceções é baseado na busca por um tratador, quando uma exeção ocorre. O tratamento é feito no comando `c2` especificado após a cláusula **except** do comando **try**. A busca por um tratador começa pela procura por um comando **try** no nível léxico que circunda mais diretamente o comando no qual a exceção ocorreu; se nenhum tratador for encontrado, a exceção é

propagada, o que significa que a exceção é considerada como tendo ocorrido no ponto de chamada da função ou método no qual a exceção ocorreu. O tipo da exceção especificada após **except** tem que casar com o tipo da exceção ocorrida. Exceções são objetos da classe `Exception` ou de uma subclasse de `Exception` em Python, e uma exceção `E` ocorrida casa com uma classe `E'` especificada em uma cláusula **except** `E'` se `E` é igual a `E'` ou é uma subclasse de `E'`.

Por exemplo, o código seguinte imprime `B`, `C`, nessa ordem (o comando **raise** indica a exceção a ser causada):

```
class B(Exception):
    pass
class C(B):
    pass
for E in [B, C]:
    try:
        raise E()
    except C:
        print("C")
    except B:
        print("B")
```

Com as cláusulas invertidas (i.e. com **except** `B` primeiro), it o código imprimiria `B, B` (ou seja, é selecionada a primeira cláusula que casa com a exceção ocorrida).

A última cláusula **except** pode omitir o nome da exceção, servindo como um coringa. Deve ser usada com cuidado, pois especifica casamento com qualquer exceção.

Um exemplo de um programa simples no qual ocorre uma propagação e tratamento da exceção propagada é mostrado a seguir:

```
def lerInt():
    return int(input("Digite um numero inteiro: "))
def main():
    try:
        x = lerInt()
        y = lerInt()
        print('Soma dos inteiros lidos: {0}'.format(str(x+y)))
    except ValueError:
        print("Numero inteiro esperado. Tente novamente.")
        main()
```

Um comando **try** pode incluir uma cláusula **else**, que deve seguir outras cláusulas **except**, e especifica comando que é executado se o comando especificado após **try** não causa uma exceção. Por exemplo:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'tem', len(f.readlines()), 'linhas')
        f.close()
```

Uma cláusula **else** permite especificar código que deve ser executado apenas se nenhuma exceção ocorrer (i.e. evita código que é executado não intencionalmente quando ocorrem exceções).

Uma cláusula **finally** pode ser especificada, para incluir código que é executado quando exceções ocorrem ou não, sendo usada na prática tipicamente para liberar recursos alocados. A exceção ocorrida é causada novamente depois que o código na cláusula **finally** é executado. Por exemplo:

```
    def divide(x, y):
        try:
            r = x / y
        except ZeroDivisionError:
            print("divisao por zero!")
        else:
            print("resultado da divisao de {0} por {1} =  {2}".format(x,y,r))
        finally:
            print("clausula finally executada")
```

```
>>> divide(2, 1)
resultado da divisao de 2 por 1 = 2.0
clausula finally executada
>>> divide(2, 0)
divisao por zero!
clausula finally executada
>>> divide("2", "1")
clausula finally executada
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Quando uma exceção ocorre, o objeto que representa a exceção pode conter valores, especificados na criação da exceção. O acesso a esse objeto pode ser feito usando opção (opcional) `as` especificada na cláusula **except**. A opção `as v` especifica `v` como nome deé variável que contém a instância correspondente à exceção ocorrida. Os argumentos passados quando a exceção é causada podem ser usados via `v.args` ou via a transformação direta dos argumentos em uma cadeia de caracteres (via o nome mágico `__str__`, criado automaticamente quando uma exceção é causada (sem necessidade de uso de `v.args`). Por exemplo:

```
>>> try:
...     raise Exception('abcd', '123')
... except Exception as inst:
...     print(type(inst))    # tipo da excecao causada
...     print(inst.args)     # argumentos armazenados na instancia
...     print(inst)          # __str__ permite acesso aos argumentos diretamente
...     x, y = inst.args     # argumentos convertidos em uma dupla
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('abcd', '123')
('abcd', '123')
x = abcd
y = 123
```

Exercícios

1. Escreva em Python programa que leia vários caracteres, que devem ser dígitos de `'0'` a `'9`) e imprima o maior valor inteiro que se pode formar com esses dígitos, de modo que: se um caracterea lido não for um dígito uma mensagem (contendo o caractere digitado que não é um dígito) deve ser emitida, e um novo caractere deve ser lido; a leitura deve terminar quando um caractere `'-'` for lido.

   O programa deve definir e usar função `charPraInt` que recebe um caractere, retorna o inteiro correspondente, se o caractere for um dígito entre `'0'` e `'9`, senão causa uma exceção, que contém o caractere digitado que não é dígito.

2. Defina classe `Data`, que represente uma data do calendário — i.e. cujos objetos têm variáveis que representam dia, mês e ano —, de modo que a inicialização de objetos da classe cause uma exceção se o dia, mês e ano passados como argumentos para a inicialização de variáveis do objeto não representarem uma data válida.

   Considere anos bissextos. O tratamento da exceção pode apenas emitir mensagem de data inválida, após armazenamento dos valores passados para inicialização do objeto.