

Python: Exceções, Iteradores e Geradores

Claudio Esperança

Exceções

- Quando um programa encontra dificuldades não previstas, diz-se que uma condição excepcional ou uma *exceção* ocorreu
 - Um erro é uma exceção mas nem toda exceção é um erro
- Para poder representar tais eventos, Python define os chamados objetos de exceção (*exception objects*)
- Se a condição excepcional não é prevista (e tratada), o programa termina com uma mensagem de rastreamento:

```
>>> 1/0
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#0>", line 1, in -toplevel-
```

```
    1/0
```

```
ZeroDivisionError: integer division or modulo by  
zero
```

Objetos de Exceção

- Cada exceção individual corresponde a um *objeto de exceção*, que por sua vez é uma instância de alguma *classe de exceção*
 - No exemplo anterior, tal objeto é instância da classe `ZeroDivisionError`
- Diz-se que o programa gerou ou levantou (*raised*, em inglês) uma condição de exceção na forma de um objeto
- Um programa bem elaborado precisa capturar (*catch*, em inglês) tais objetos e tratá-los para que a execução não seja abortada

Avisos

- Existem condições excepcionais menos sérias que não provocam o levantamento de um objeto de exceção, mas apenas são exibidas sob a forma de um aviso
- Por exemplo,

```
>>> import regex
```

```
Warning (from warnings module):
```

```
File "__main__", line 1
```

```
DeprecationWarning: the regex module is  
deprecated; please use the re module
```

- Neste caso, o intepretador nos sinaliza que o módulo regex é antigo e que foi substituído por outro mais atualizado chamado re
- O programa não falha, mas o programador fica ciente que provavelmente deve reescrever seu programa usando o módulo re para evitar obsolescência

O comando *raise*

- Para sinalizar a ocorrência de uma condição excepcional, pode-se usar o comando `raise` que tem uma das formas:
 - `raise classe`
 - `raise classe, mensagem`
 - `raise classe (mensagem)`
- Onde classe é uma das classes de exceção definidas pelo Python
 - Para saber todos os tipos de exceção consulte o manual
 - Se quiser uma classe genérica use a classe `Exception`
 - Uma listagem pode ser obtida escrevendo

```
>>> import exceptions
>>> dir(exceptions)
['ArithmeticError', 'AssertionError',
 'AttributeError', ...
```

Exemplo

```
>>> raise Exception
```

```
Traceback (most recent call last):  
  File "<pyshell#3>", line 1, in -toplevel-  
    raise Exception
```

```
Exception
```

```
>>> raise Exception, "Deu bode"
```

```
Traceback (most recent call last):  
  File "<pyshell#5>", line 1, in -toplevel-  
    raise Exception, "Deu bode"
```

```
Exception: Deu bode
```

```
>>> raise Exception("Deu Bode")
```

```
Traceback (most recent call last):  
  File "<pyshell#7>", line 1, in -toplevel-  
    raise Exception("Deu Bode")
```

```
Exception: Deu Bode
```

Algumas Classes de Exceção

Classe	Descrição
Exception	Classe base para todas as exceções
AttributeError	Falha no acesso ou atribuição a atributo de classe
IOError	Falha no acesso a arquivo inexistente ou outros de E/S
IndexError	Índice inexistente de seqüência
KeyError	Chave inexistente de dicionário
NameError	Variável inexistente
SyntaxError	Erro de sintaxe (código errado)
TypeError	Operador embutido aplicado a objeto de tipo errado
ValueError	Operador embutido aplicado a objeto de tipo certo mas valor inapropriado
ZeroDivisionError	Divisão ou módulo por zero

Criando uma Classe de Exceção

- Basta criar uma classe da forma habitual derivando-a da classe `Exception`
- Não é preciso redefinir qualquer método
- Ex.:

```
>>> class MinhaExcecao(Exception): pass
```

```
>>> raise MinhaExcecao("Deu bode!")
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#11>", line 1, in -toplevel-  
    raise MinhaExcecao("Deu bode!")
```

```
MinhaExcecao: Deu bode!
```

Capturando Exceções

- Para capturar uma exceção possivelmente levantada por um trecho de código, pode-se usar a construção `try/except`:

`try:`

Código

`except Exceções:`

Código de tratamento da exceção

- Sendo que *Exceções* pode ser:

- *Classe*

- *Classe, var*

- *(Classe1, ..., ClasseN)*

- *(Classe1, ..., ClasseN), var*

- Onde:

- *Classe*, *Classe1* e *ClasseN* são nomes de classes de exceção

- *Var* é uma variável à qual é atribuída um objeto de exceção

Exemplo 1

```
>>> try:
    a = input("Entre com um numero ")
    b = input("Entre com outro numero ")
    print a, "/", b, "=", a/b
except ZeroDivisionError:
    print "0oops, segundo numero não pode ser\
zero!"
```

Entre com um numero 1

Entre com outro numero 0

1 / 0 = 0oops, segundo numero não pode ser zero!

Exemplo 2

```
>>> try:
    a = input("Entre com um numero ")
    b = input("Entre com outro numero ")
    print a, "/", b, "=", a/b
except (ZeroDivisionError,TypeError):
    print "0oops, tente novamente!"
```

Entre com um numero 1

Entre com outro numero "a"

1 / a = 0oops, tente novamente!

Exemplo 3

```
>>> try:
    a = input("Entre com um numero ")
    b = input("Entre com outro numero ")
    print a, "/", b, "=", a/b
except (ZeroDivisionError,TypeError),e:
    print "0oops, deu erro:",e
```

Entre com um numero 1

Entre com outro numero "z"

1 / z = 0oops, deu erro: unsupported operand
type(s) for /: 'int' and 'str'

Mais *except*

- É possível tratar diferentemente as diversas exceções usando duas ou mais cláusulas `except`
- Se quisermos nos prevenir contra qualquer tipo de erro, podemos usar uma cláusula `except` sem nome de classe
 - Outra opção é usar a classe `Exception`, que é base para todas as exceções e portanto casa com qualquer exceção
- Se não quisermos tratar um erro em uma cláusula `except`, podemos passá-la adiante usando o comando `raise`
 - Nesse caso, podemos usar um `raise` sem argumentos ou passar explicitamente um objeto de exceção

Exemplo 4

```
>>> try:
    a = input("Entre com um numero ")
    b = input("Entre com outro numero ")
    print a, "/", b, "=", a/b
except ZeroDivisionError:
    print "0oops, divisão por zero"
except TypeError:
    print "0oops, você não deu um número"
except:
    print "Deu um bode qualquer"
```

Entre com um numero 2

Entre com outro numero fads2312

Deu um bode qualquer

Exemplo 5

```
>>> try:
    a = input("Entre com um numero ")
    b = input("Entre com outro numero ")
    print a, "/", b, "=", a/b
except (ZeroDivisionError,TypeError),e:
    print "Ooops, deu erro:",e
except Exception,e:
    print "Deu bode não previsto:",e
    raise
```

Entre com um numero a

Entre com outro numero

Deu bode não previsto: EOF when reading a line

Traceback (most recent call last):

File "<pyshell#52>", line 3, in -toplevel-

b = input("Entre com outro numero ")

EOFError: EOF when reading a line

A cláusula *else*

- É possível completar um comando `try` com uma cláusula `else` que introduz um trecho de código que só é executado quando *nenhuma* exceção ocorre:

`try:`

Código

`except Exceções:`

Código de tratamento da exceção

`else:`

Código executado se não ocorrem exceções

Exemplo 6

```
>>> while True:
    try:
        a = input("Entre com um numero ")
        b = input("Entre com outro numero ")
        print a, "/", b, "=", a/b
    except Exception,e:
        print "Deu bode:",e
        print "Tente novamente"
    else:
        break
```

```
Entre com um numero 1
Entre com outro numero xxx
Deu bode: name 'xxx' is not defined
Tente novamente
Entre com um numero 1
Entre com outro numero 2
1 / 2 = 0
```

A cláusula *finally*

- A cláusula `finally` pode ser usada para se assegurar que mesmo que ocorra algum erro, uma determinada seqüência de comandos vai ser executada
 - Pode ser usada para restabelecer alguma variável para um valor `default`, por exemplo
- A cláusula `finally` e cláusulas `except` são mutuamente exclusivas
 - Exceções nesse caso não são tratadas
 - É possível combinar ambas usando comandos `try` aninhados, mas normalmente não há muito uso para isso

Exemplo 7

```
>>> try:
    try:
        x = input("Entre com um número")
    finally:
        print "restabelecendo um valor para x"
        x = None
except:
    print "Deu bode"
```

```
Entre com um número 2xx
restabelecendo um valor para x
Deu bode
```

Iteradores

- São maneiras genéricas de implementar iterações com classes
 - Permite o uso do comando `for`
 - É geralmente mais econômico do que usar uma lista pois não é preciso armazenar todos os valores, mas apenas computar um por vez
- Um iterador é uma classe que implementa o método mágico `__iter__`
 - É um método que, por sua vez, retorna um objeto que implementa um método chamado `next`
 - O método `next` deve retornar o “próximo” valor a ser iterado
 - Se não há próximo valor, `next` deve “levantar” a exceção `StopIteration`

Exemplo

```
>>> class MeuIterador:
    a = 0
    def __iter__(self): return self
    def next(self):
        if self.a>10: raise StopIteration
        self.a += 1
        return self.a
```

```
>>> iter = MeuIterador()
```

```
>>> for i in iter:
    print i,
```

```
1 2 3 4 5 6 7 8 9 10 11
```

Geradores

- Geradores são funções especiais que retornam iteradores
- Em resumo, uma função geradora é uma que contém o comando `yield valor`
- Uma função geradora normalmente é chamada para obter o iterador para um comando `for`
 - O comando `for` automaticamente iterará sobre todos os valores que `yield` “retorna”
 - Observe que o iterador produzido pela função geradora é tal que o código que gera os valores e o código dentro do `for` se sucedem alternadamente
- Geradores são especialmente úteis em códigos recursivos

Exemplo

```
>>> def gerador():  
    for i in range(10):  
        print "i = ", i  
        yield i  
  
>>> for j in gerador():  
    print "j = ", j
```

```
i = 0
```

```
j = 0
```

```
i = 1
```

```
j = 1
```

```
...
```

```
i = 9
```

```
j = 9
```