Computação II (MAB 225)

Fabio Mascarenhas - 2015.1

http://www.dcc.ufrj.br/~fabiom/pythonoo

Tratamento de Erros

- Até agora não nos preocupamos com erros em nosso programa, apenas assumimos que tudo sempre dá certo
  - Todas as entradas para nossos construtores e métodos estão corretas
  - Toda operação é bem sucedida
- Na prática, erros acontecem, e operações falham
- Uma maneira de indicar erros é *em banda*: um método retorna um ou mais valores especiais para indicar um erro, ao invés de seu retorno normal

Exceções

- O tratamento de erros também pode ser *fora de banda*: um erro é sinalizado e tratado por um mecanismo diferente da chamada e retorno de método

- Esse mecanismo são as *exceções*

- Uma *exceção* é um objeto que sinaliza que uma falha aconteceu

- Uma exceção é *lançada* quando ocorre uma falha, o lançamento interrompe a execução do programa, e transfere o controle para um *tratador de exceções*, um trecho de código instalado por algum dos métodos que estão executando no momento

Exceções - Definindo

- Exceções são objetos como qualquer outro, mas sempre são instâncias de `Exception`, ou de uma *subclasse* de `Exception`
  - Uma classe que extende `Exception`, ou uma classe que extende uma classe que extende `Exception`, etc.
- Geralmente, a classe da exceção deve indicar a *categoria* da falha que aconteceu
- `ArithmeticError`, `KeyError`, `NameError`, `FloatingPointError`, `IOError`, `ImportError`, `IndeError`, `NotImplementedError`, `StopIteration`, `ZeroDivisionError`, e muitas outras

Exceções - Lançando

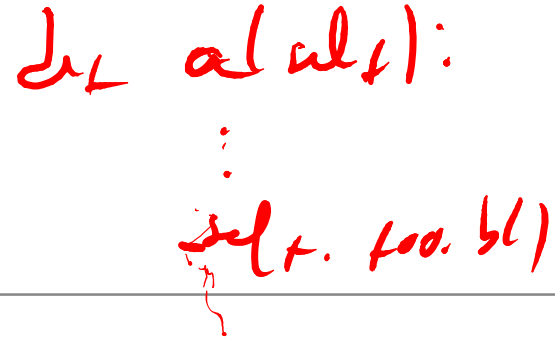
- Lançamos uma exceção (lembrando: uma *instância* de uma classe de exceção) usando o comando `raise`, passando para ela a exceção:

```
def next(self):
    if self.a > self.b:
        raise StopIteration()
    else:
        atual = self.a
        self.a = self.a + 1
        return atual
```

- O mais comum é instanciar a exceção no próprio comando raise, mas isso não é necessário:

```
si = StopIteration();
raise si;
```

Pilha de Execução

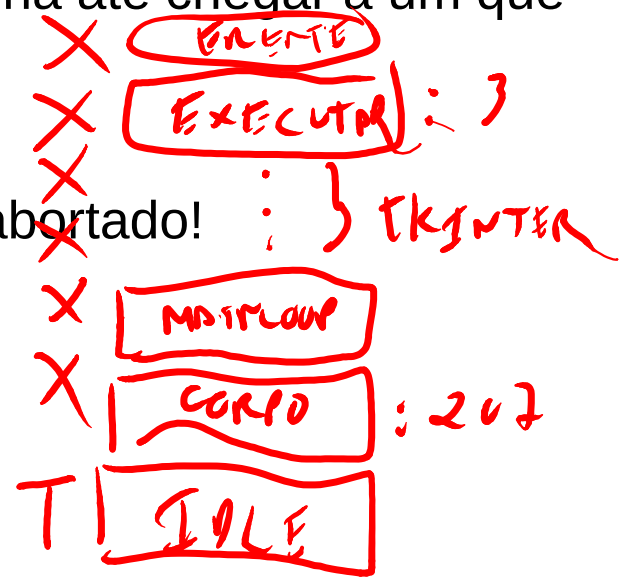


- Quando um método chama outro, a execução do primeiro fica suspensa no ponto da chamada até que ela termine

- Os métodos que estão suspensos em um dado momento formam a *pilha de execução*

- O lançamento de uma exceção interrompe o fluxo normal de execução, e faz a execução ir abandonando os métodos que estão na pilha até chegar a um que possua um *tratador* para aquela exceção

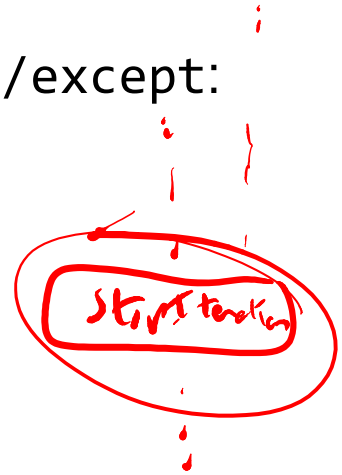
- Se não houver nenhum tratador, o programa inteiro é abortado!



Exceções - Capturando

- Instalamos um tratador de exceções usando um bloco `try/except`:

```
try:
    print it.next()
except StopIteration:
    print "acabou!"
```



- Qualquer exceção que aconteça durante a execução do bloco `try` que não tenha sido tratada e que seja subclasse da classe dada na cláusula `except` é capturada, e então o bloco `except` é executado

- Se não ocorrer nenhuma outra exceção no bloco `except`, a execução prossegue normalmente com o que venha depois do `try/except`

Exceções - Sutilezas

- O tratador de exceções mais *recente* na pilha de execução tem preferência, então não é garantido que o bloco `except` seja executado caso uma exceção ocorra, já que ela pode ser tratada antes

- A declaração que segue a cláusula `except` diz quais exceções queremos tratar: uma instância da classe dada, ou de uma classe que extende ela, ou de uma classe que extende uma classe que extende ela, etc.

- Se o tipo da exceção não bate, é como se o tratador não existisse, e continuamos a busca na pilha de execução

- Se uma exceção ocorre dentro do bloco `except`, ela não pode ser tratada pelo próprio bloco: a execução dele é interrompida, e outra busca começa

Múltiplos `except`

- Um bloco `try/except` pode ter múltiplos blocos `except`, cada um capturando um tipo diferente de exceção

- Cada `except` introduz um tratador para o seu tipo:

```
try:
    print it.next()
except StopIteration:
    print "acabou!"
except Exception:
    print "outro problema"
```

- Se queremos um bloco `except` que captura qualquer exceção é só não dar o tipo de exceção que queremos, (`except:`)

Mensagens em exceções

- Todas as exceções pré-definidas possuem um construtor que recebe uma *mensagem de erro* opcional como seu primeiro parâmetro

- Essa mensagem fica armazenada no campo `message` do objeto da exceção, e é usada para formatar uma mensagem de erro pelo interpretador se a exceção não for capturada e encerrar o programa

```
>>> e = Exception("mensagem de erro")
>>> e.message
'mensagem de erro'
>>> raise e

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    raise e
Exception: mensagem de erro
>>>
```

Capturando o objeto exceção

- Quando capturamos uma exceção em um bloco `except`, podemos dar nome ao objeto exceção capturado com `as`:

```
try:
    f = open('arquivo.txt')
    s = f.readline()
except IOError as e:
    print "Erro de I/O: %s" % e.message
```

- Várias classes de exceção têm campos que dão mais informação sobre o erro que aconteceu

- `IOError`, por exemplo, tem um campo `filename` que diz qual arquivo foi a fonte do erro

Código de Finalização

- Algumas vezes, temos código que precisa ser executado mesmo que uma exceção aconteça, independente de qual seja a exceção

- Uma maneira de fazer isso seria duplicando esse código, e usando um tratador vazio:

```
try:
    try:
        # código normal
    except ...:
        # outros tratadores de exceção...
    # finalização
except Exception as e:
    # finalização
    raise e
```

- Existe um jeito melhor: a cláusula `finally`

Usando `finally`

- Podemos terminar um comando `try/except` com um bloco `finally:`

```
try:
    # código normal
except ...:
    # outros tratadores de exceção...
finally:
    # finalização
```

- O bloco finally sempre é executado, ocorra ou não uma exceção

- Se não ocorrer uma exceção, ele é executado após o bloco try, e se ocorrer alguma exceção ele é executado antes de abandonar o contexto atual