Tópicos intermediários em Python

Linguagem de programação - Professor Flávio Varejão

Universidade Federal do Espírito Santo

25 de Abril de 2011

Outline

Outline

Outline

Outline

Expressões regulares

Expressões regulares são uma forma padronizada e poderosa de fazer busca, substituição e parsing de textos com cadeias complexas de caracteres.

Expressões regulares

Expressões regulares são uma forma padronizada e poderosa de fazer busca, substituição e parsing de textos com cadeias complexas de caracteres.
O Python possui na classe *string* funções de busca (*index*, *find* e *count*), substituição (*replace*) e parsing (*split*).

Expressões regulares

Expressões regulares são uma forma padronizada e poderosa de fazer busca, substituição e parsing de textos com cadeias complexas de caracteres.
O Python possui na classe *string* funções de busca (*index*, *find* e *count*), substituição (*replace*) e parsing (*split*). Em cenários básicos o seu uso é aconselhável e até desejável: elas devem ser utilizadas se a tarefa pode ser feita sem muito esforço.

Expressões regulares

Expressões regulares são uma forma padronizada e poderosa de fazer busca, substituição e parsing de textos com cadeias complexas de caracteres.
O Python possui na classe *string* funções de busca (*index*, *find* e *count*), substituição (*replace*) e parsing (*split*). Em cenários básicos o seu uso é aconselhável e até desejável: elas devem ser utilizadas se a tarefa pode ser feita sem muito esforço.
Contudo o uso extensivo dessas funções com if's aninhados e loops pode deixar o código "sujo", difícil de ler e escrever.

Expressões regulares

Com expressões regulares pode-se criar strings de busca complexas com caracteres especiais capazes de casar com uma família de strings no lugar de uma string específica.
Os caracteres especiais possuem uma semântica própria comum a várias ferramentas. Por exemplo o bash do linux, a linguagem de programação Perl, o construtor de linguagens formais Bison...

Expressões regulares

Lista de caracteres especiais;

- Caracteres normais - são as expressões regulares mais simples possíveis. Elas representam um matching exato.

```
import re
p = re.compile('abc')
print p.findall("abc10deafc20abcghi")
```

Expressões regulares

Lista de caracteres especiais;

- Caracteres normais - são as expressões regulares mais simples possíveis. Elas representam um matching exato.

```
import re
p = re.compile('abc')
print p.findall("abc10deafc20abcghi")

//  ['abc', 'abc']
```

Expressões regulares

Lista de caracteres especiais;

- Caracteres normais - são as expressões regulares mais simples possíveis. Elas representam um matching exato.

```
import re
p = re.compile('abc')
print p.findall("abc10deafc20abcghi")

//  ['abc', 'abc']
```

- . - Qualquer caracter exceto "nova linha"

```
import re
p = re.compile('a.c')
print p.findall("abc10deafc20abcghi")
```

Expressões regulares

Lista de caracteres especiais;

- Caracteres normais - são as expressões regulares mais simples possíveis. Elas representam um matching exato.

```
import re
p = re.compile('abc')
print p.findall("abc10deafc20abcghi")
```

```
//  ['abc', 'abc']
```

- . - Qualquer caracter exceto "nova linha"

```
import re
p = re.compile('a.c')
print p.findall("abc10deafc20abcghi")
```

```
//  ['abc', 'afc', 'abc']
```

Expressões regulares

- ^ - O Começo da linha

```
import re
p = re.compile('^a.c')
print p.findall("abc10deafc20abcghi")
```

Expressões regulares

- ^ - O Começo da linha

```
import re
p = re.compile('^a.c')
print p.findall("abc10deafc20abcghi")
```

```
// ['abc']
```

Expressões regulares

- ^ - O Começo da linha

```
import re
p = re.compile('^a.c')
print p.findall("abc10deafc20abcghi")
```

```
// ['abc']
```

- $ - O fim da linha

```
import re
p = re.compile('.h.$')
print p.findall("abc10deafc20abcghi")
```

Expressões regulares

- ^ - O Começo da linha

```
import re
p = re.compile('^a.c')
print p.findall("abc10deafc20abcghi")
```

```
// ['abc']
```

- $ - O fim da linha

```
import re
p = re.compile('.h.$')
print p.findall("abc10deafc20abcghi")
```

```
//  ['ghi']
```

- * - Zero ou mais repetições da expressão regular que a precede *(a maior string possível)*.

```
import re
p = re.compile('a.*b')
print p.findall("abc10deafc20abcghi")
```

- * - Zero ou mais repetições da expressão regular que a precede *(a maior string possível)*.

```
import re
p = re.compile('a.*b')
print p.findall("abc10deafc20abcghi")
```

```
// ['abc10deafc20ab']
```

- * - Zero ou mais repetições da expressão regular que a precede *(a maior string possível)*.

```
import re
p = re.compile('a.*b')
print p.findall("abc10deafc20abcghi")
```

```
// ['abc10deafc20ab']
```

- + - Uma ou mais repetições da expressão regular que a precede.

```
import re
p = re.compile('ca+d')
print p.findall("cdcadcaad")
```

- * - Zero ou mais repetições da expressão regular que a precede *(a maior string possível)*.

```
import re
p = re.compile('a.*b')
print p.findall("abc10deafc20abcghi")
```

```
// ['abc10deafc20ab']
```

- + - Uma ou mais repetições da expressão regular que a precede.

```
import re
p = re.compile('ca+d')
print p.findall("cdcadcaad")
```

```
//  ['cad', 'caad']
```

- ? - Zero ou uma repetições da expressão regular que a precede.

```
import re
p = re.compile('ca?d')
print p.findall("cdcadcaaad")
```

- ? - Zero ou uma repetições da expressão regular que a precede.

```
import re
p = re.compile('ca?d')
print p.findall("cdcadcaaad")
```

```
// ['cd', 'cad']
```

- **?** - Zero ou uma repetições da expressão regular que a precede.

```
import re
p = re.compile('ca?d')
print p.findall("cdcadcaaad")
```

```
// ['cd', 'cad']
```

- ***?, +? ou ??** - Transforma o modificador após a interrogação em um matching minimal.

```
import re
p = re.compile('a.*?b')
print p.findall("abc10deafc20abcghi")
```

- **?** - Zero ou uma repetições da expressão regular que a precede.

```
import re
p = re.compile('ca?d')
print p.findall("cdcadcaaad")
```

```
// ['cd', 'cad']
```

- ***?, +? ou ??** - Transforma o modificador após a interrogação em um matching minimal.

```
import re
p = re.compile('a.*?b')
print p.findall("abc10deafc20abcghi")
```

```
// ['ab', 'afc20ab']
```

- **{n}** - n repetições da expressão regular anterior.

```
import re
p = re.compile('a{4}')
print p.findall("aaaabaa")
```

- **{n}** - n repetições da expressão regular anterior.

```
import re
p = re.compile('a{4}')
print p.findall("aaaabaa")
```

```
// ['aaaa']
```

- **{n}** - n repetições da expressão regular anterior.

```
import re
p = re.compile('a{4}')
print p.findall("aaaabaa")
```

```
// ['aaaa']
```

- **[]** - usado para definir um conjunto de caracteres que podem ser aceitos

```
import re
p = re.compile('[abc]z')
print p.findall("azzzbz")
```

- **{n}** - n repetições da expressão regular anterior.

```
import re
p = re.compile('a{4}')
print p.findall("aaaabaa")
```

```
// ['aaaa']
```

- **[]** - usado para definir um conjunto de caracteres que podem ser aceitos

```
import re
p = re.compile('[abc]z')
print p.findall("azzzbz")
```

```
// ['az', 'bz']
```

- | - usado para separar duas expressões regulares de modo que a string só precise "encaixar"em uma delas.

```
import re
p = re.compile('(abc)|(def)')
print p.findall("zdefz")
```

- | - usado para separar duas expressões regulares de modo que a string só precise "encaixar"em uma delas.

```
import re
p = re.compile('(abc)|(def)')
print p.findall("zdefz")

// [('', 'def')]
```

- | - usado para separar duas expressões regulares de modo que a string só precise "encaixar"em uma delas.

```
import re
p = re.compile('(abc)|(def)')
print p.findall("zdefz")

// [('', 'def')]
```

- **(?P<nome>(Expressão Regular))** - Usado para criar um grupo nomeado para recuperar o label posteriormente no código (exemplo abaixo).

Expressões regulares - Buscando

```
import re

p = re.compile('\d+')
print p.findall("abc10def20ghi")

iter = p.finditer("abc10def20ghi")

for match in iter:
    print match.span()

// ['10', '20']
// (3, 5)
// (8, 10)
```

Expressões regulares - Substituindo

```
import re
re.sub(r'\d+','xx', 'ss10ss')

//   ssxxss
```

Expressões regulares - Expressões rotuladas

Considere o seguinte exemplo para fazer parsing de uma operação de atribuição:

```
to_parse = " xpto = 4 * 8 "

label = to_parse.split("=")[0].strip()
operation = to_parse.split("=")[1].strip()

if(operation.find("*") != -1):
    separator = "*"
if(operation.find("-") != -1):
    separator = "-"
if(operation.find("+") != -1):
    separator = "+"
if(operation.find("/") != -1):
    separator = "/"
```

```
value1 = operation.split(separator)[0].strip()
value2 = operation.split(separator)[1].strip()

print label, separator, value1, value2

// xpto * 4 8
```

Expressões regulares - Expressões rotuladas

O mesmo programa pode ser implementado da seguinte forma:

```
import re

mask = r"(\s*)(?P<label>[a-zA-Z_]+\w*)(\s*=\s*)
␣␣␣␣(?P<value1>\d+)(\s*)(?P<separator>[*-/+])
␣␣␣␣(\s*)(?P<value2>\d+)(\s*)"
m = re.match(mask, to_parse)
print m.group('label'), m.group('separator'),
          m.group('value1'), m.group('value2')

// xpto * 4 8
```

Outline

Exceções

Exceções acontecem o tempo todo em Python. Todo módulo na biblioteca padrão as usa. E o próprio interpretador irá dispará-las em várias circunstâncias diferentes. O programador iniciante de Python já deve ter visto vários exemplos desse comportamento. Entre as exceções mais comuns podemos citar:

- Acessar uma índice não existente em uma lista vai gerar uma exceção do tipo *IndexError*

```
frutas = ['manga', 'abacate', 'laranja']
print x[4]
//  IndexError: list index out of range
```

- Procurar um valor inexistente em uma lista vai gerar uma exceção do tipo *ValueError*

```
frutas.index('melancia')
//  ValueError: list.index(x): x not in list
```

Exceções

- Chamar um método não existente vai gerar uma exceção do tipo *AttributeError*

```
frutas.fazerSalada()
//  AttributeError: 'list' object has no attribute 'fazer
```

- Referenciar uma variável não existente vai gerar uma exceção do tipo *NameError*

```
print fruta
//  NameError: name 'fruta' is not defined
```

- Fazer operações sobre tipos de dados diferentes sem realizar coerção vai gerar uma exceção do tipo *TypeError*

```
fruta = frutas + 'goiaba'
// TypeError: can only concatenate list (not "str")
    to list
```

Exceções

Quando não tratamos uma exceção o comportamento padrão do Python é executado: Uma linha de erro é impressa e o programa tem a sua execução abortada. Isso significa que o programa inteiro pode parar a sua execução por causa de uma exceção que poderia ter sido prevista.

Exceções

Quando não tratamos uma exceção o comportamento padrão do Python é executado: Uma linha de erro é impressa e o programa tem a sua execução abortada. Isso significa que o programa inteiro pode parar a sua execução por causa de uma exceção que poderia ter sido prevista.

Algumas vezes as exceções são provenientes de um bug no código, nesse caso é desejável que ela pare o programa e imprimia alguma mensagem de erro para o programador consertá-lo. Por esse motivo não é desejável criar um capturador de exceção global.

Exceções

Quando não tratamos uma exceção o comportamento padrão do Python é executado: Uma linha de erro é impressa e o programa tem a sua execução abortada. Isso significa que o programa inteiro pode parar a sua execução por causa de uma exceção que poderia ter sido prevista.

Algumas vezes as exceções são provenientes de um bug no código, nesse caso é desejável que ela pare o programa e imprimia alguma mensagem de erro para o programador consertá-lo. Por esse motivo não é desejável criar um capturador de exceção global.

Mas esse não é sempre o caso. Por exemplo: uma exceção pode ocorrer ao se abrir um arquivo. Ele pode não existir ou o usuário não possuir as permissões necessárias para abri-lo. Toda fonte de provável exceção deve ser tratada usando blocos de código do tipo try ... except.

Exceções

Exemplo de como tratar uma abertura mal sucedida de arquivo.

```
try:
    arquivo = open("/naoexiste")
except IOError:
    print "O arquivo nao existe. Fazendo alguma coisa sobre i
else:
    print "O arquivo foi aberto. Fazendo as coisas aqui"
finally:
    try:
        print "Fechando o arquivo"
        arquivo.close()
    except:
        print "Nao to nem ai"
// O arquivo foi aberto. Fazendo as coisas aqui
// Fechando o arquivo
```

Toda exceção do tipo IOError que ocorre dentro do bloco try será tratada no respectivo bloco except

Exceções

Exemplo de como tratar uma abertura mal sucedida de arquivo.

```
try:
    arquivo = open("/naoexiste")
except IOError:
    print "O arquivo nao existe. Fazendo alguma coisa sobre i
else:
    print "O arquivo foi aberto. Fazendo as coisas aqui"
finally:
    try:
        print "Fechando o arquivo"
        arquivo.close()
    except:
        print "Nao to nem ai"
// O arquivo foi aberto. Fazendo as coisas aqui
// Fechando o arquivo
```

Repare que se tudo correr bem o código dentro do bloco "else"será executado

Exceções

Exemplo de como tratar uma abertura mal sucedida de arquivo.

```
try:
    arquivo = open("/naoexiste")
except IOError:
    print "O arquivo nao existe. Fazendo alguma coisa sobre i
else:
    print "O arquivo foi aberto. Fazendo as coisas aqui"
finally:
    try:
        print "Fechando o arquivo"
        arquivo.close()
    except:
        print "Nao to nem ai"
// O arquivo foi aberto. Fazendo as coisas aqui
// Fechando o arquivo
```

O bloco finally é sempre executado e dentro dele há um outro try...except para tratar o fechamento do arquivo

Exceções

Capturando múltiplas exceções

```
arquivo = []
try:
    arquivo[1] = open("/naoexiste")
except IOError, IndexError:
    print "O arquivo nao existe ou lista muito pequena.
    Fazendo alguma coisa sobre isto ..."
```

Exceções

Propagando exceções. Basta chamar o comando raise com um objeto que herda de Exception

```
arquivo = []
try:
    arquivo[1] = open("/naoexiste")
except IOError:
    print "O␣arquivo␣nao␣existe␣ou␣lista␣muito␣pequena.
␣␣␣␣Fazendo␣alguma␣coisa␣sobre␣isto..."
except IndexError as error:
    print error
    raise error
// list assignment index out of range
// Traceback (most recent call last):
//   File "exept2.py", line 8, in <module>
//     raise error
// IndexError: list assignment index out of range
```

Exceções

Criando a sua própria exceção;

```
class MyExpt (Exception):
    def __str__(self):
        return 'Erro!'

class MyObj():
    def doStuff(self):
        raise MyExpt()

my = MyObj()
my.doStuff()

//  File "exept3.py", line 10, in <module>
//    my.doStuff()
//  File "exept3.py", line 7, in doStuff
//    raise MyExpt()
//__main__.MyExpt: Erro!
```

Outline

Orientação à objetos

Python é uma linguagem com suporte a orientação a objetos. Pode-se criar novas classes, instanciar objetos, fazer sobrecarga de operadores e realizar herança de classes. Python possui um coletor de lixo que funciona com contagem de referência quanto um objeto não possui mais referências ele é automaticamente destruído. Leaks de memória são raros em Python. Acontecem, na maioria das vezes durante a utilização de bibliotecas implementadas em linguagens de baixo nível.

Outline

Usando Classes

A definição de classes em Python é muito simples. Segue a definição básica de uma classe genérica "Interpretador".

```
class Interpretador():
    """Classe generica interpretador"""
    variavelClasse = 0

    def _metodoclasse():
        print 'Metodo de classe'

    metodoClasse = staticmethod(_metodoclasse)
```

Usando Classes

```
    def __init__(self, filename=""):
        """Construtor"""
        self.filename = filename

    def load_file(self):
        """Carrega arquivo"""
        self.lines = open(filename).readlines()

    def __del__(self):
        """Destrutor"""
        print 'destrutor'
def teste():
    i = Interpretador()
    i2 = Interpretador()
    del i
    print "out"
```

Usando Classes

```
>>> print Interpretador.__doc__
//   classe generica interpretador
>>> f()
//   destrutor
//   out
//   destrutor
>>> Interpretador.metodoClasse()
//   Metodo de classe
```

Outline

Herança

Expandindo um pouco o exemplo anterior, vamos construir mais duas classes: a classe Sucuri que implementa coisas relacionadas à linguagem sucuri especificamente e a classe Interpretador_Sucuri que une as duas usando herança múltipla.

```
class Sucuri():
    def load_file(self):
        print 'load_file sucuri'

class Interpretador_Sucuri(Sucuri, Interpretador):
    pass
```

Quando existe conflito de nomes nos métodos ou atributos das classes da herança o Python sempre escolhe aqueles que aparecem primeiro na lista das superclasses.

Outline

Encapsulamento

Python não possui encapsulamento real. Ou seja, sempre é possível acessar métodos e atributos privados. O pseudo-encapsulamento é feito da seguinte forma:

```
class Banco():
    def __zeraContas(self):
        print "ok"
// Banco().__zeraContas()
// AttributeError: Banco instance has no attribute
//   '__zeraContas'
// Banco()._Banco__zeraContas()
// ok
```

Outline

Sobrecarga de Operadores

Sobrecarregar operadores é muito simples em Python. A filosofia é que TUDO é um objeto. Inclusive os tipos primitivos. A classe int usa os métodos __add__, __sub__, __mul__, __div__ e __mod__ para calcular o resultado das operações +, -, *, / e % (respectivamente).
Portanto se você deseja utilizar um tipo especifico de operador basta criar uma classe que implementa o método que provem implementa aquela funcionalidade.

Por exemplo, vamos criar um objeto que aceita ser somado com outro do mesmo tipo:

```
class teste():
    def __init__(self, value):
        self.value = value
    def __add__(self,other):
        _ret = teste(self.value + other.value)
        return _ret
```

Métodos de operadores comuns

- **__add__** - Operador "+"
- **__or__** - Operador bitwise "ou"
- **__str__** - Impressão de strings
- **__getitem__** - Indexação
- **__len__** - Tamanho da lista
- **__lt__** - Comparação menor que
- **__eq__** - Comparação "igual"
- **__iadd__** - adição "in place"