**CHAPTER III -- REGULAR EXPRESSIONS**

Expressões regulares permitem técnicas de processamento de texto extremamente valiosas, mas os que merecem cuidadosa explicação. Python [re] módulo, em particular, permite diversos aperfeiçoamentos para básico expressões regulares (como referências anteriores nomeados, lookahead afirmações referência anterior, de pular, não gananciosos quantificadores e outros). Uma sólida introdução para as sutilezas das expressões regulares é valiosa para programadores envolvidos em tarefas de processamento de texto.

O prologo deste capítulo contém um tutorial de expressões regulares que permite que um leitor não familiarizado com expressões regulares para se mover rapidamente de simples a complexos elementos sintaxe de expressão regular. Este tutorial é direcionado principalmente para iniciantes, mas programadores familiarizados com expressões regulares em outras ferramentas de programação podem se beneficiar de uma leitura rápida do tutorial, que explica a expressão regular particular dialeto em Python.

É importante notar-se que a parte inicial das expressões regulares, ao mesmo tempo muito poderosas, também têm limitações. Em resumo, expressões regulares podem não corresponder aos padrões de agrupamentos para arbitrárias profundidades. Se essa afirmação não faz sentido, leia o Capítulo 4, que discute analisadores - em grande medida, a análise de existir resolver as limitações de expressões regulares. Em geral, se você tem dúvidas sobre se uma expressão regular é suficiente para a tarefa, tentar entender os exemplos Capítulo 4, em particular a discussão de como você pode soletrar um número de ponto flutuante.

Seção 3.1 examina uma série de problemas de processamento de texto que são resolvidos mais naturalmente usando expressão regular. Como em outros capítulos, as soluções para os problemas apresentados podem geralmente ser adotadas diretamente como pequenos utilitários para a realização de tarefas. No entanto, como em outros lugares, o objetivo maior na apresentação problemas e soluções é tratar de um estilo de pensar sobre uma classe mais ampla de problemas do que aqueles cujas soluções são apresentadas diretamente neste livro. Os leitores que estiverem interessados em uma série de os utilitários e módulos prontos provavelmente irá querer verificar recursos adicionais na Web, tais como os Vaults de Parnassus <http://www.vex.net/parnassus/> e o Python Cookbook <http://aspn.activestate.com/ASPN/Python/Cookbook/>.

Seção 3.2 é uma "referência com o comentário" na biblioteca padrão do Python módulos para fazer as tarefas de expressões regulares. Vários módulos de utilidade e compatibilidade com versões anteriores motores de expressões regulares estejam disponíveis, mas para a maioria dos leitores, a única importância será o módulo [re] em si. As discussões intercaladas com cada módulo de tentar dar alguma orientação sobre o porquê que você gostaria de usar um determinado módulo ou função, e documentação de referência tenta conter mais exemplos de real uso típico do que uma referência simples. Em muitos casos, os exemplos e discussão de funções individuais de endereço comum e padrões de projeto produtivas em Python. As referências cruzadas destinam-se a contextualizar uma determinada função (ou outra coisa) em termos de entes relacionados (e para ajudar o leitor a decidir qual é certo para ela). A listagem real de funções, constantes, classes, e outros, são por ordem alfabética em cada categoria.

**SECÇÃO 0 - Um tutorial de Expressões Regulares**

Algumas pessoas, quando confrontados com um problema, pensam "eu sei, Vou usar expressões regulares. "Agora elas têm dois problemas. - Jamie Zawinski, '<alt.religion.emacs> "(1997/08/12)

**TÓPICO - Apenas que é uma expressão regular, de qualquer maneira?**

Muitos leitores têm algum conhecimento de expressões regulares, mas alguns não tem nenhuma. Aqueles que já experimentaram o uso de expressões regulares em outras linguagens (ou em Python) provavelmente podem pular esta seção deste tutorial. Mas os leitores novos de expressões regulares (carinhosamente chamados de "regex" por usuários) deve ler esta seção, mesmo alguns com experiência podem se beneficiar de uma reciclagem.

Uma expressão regular é uma forma compacta de descrever complexo padrão de textos. Você pode usá-las para procurar padrões e, uma vez encontrada, para modificar os padrões de formas complexas. Elas também podem ser usadas para levar a cabo ações de programação que dependem de padrões.

Jamie Zawinski comenta com ironia sobre o epigrama que vale a pena pensar. As expressões regulares são incrivelmente poderosas e profundamente expressivas. Essa é a razão que escrevê-las é tão propenso a erros como a escrita de qualquer outro código de programação complexo. É sempre melhor para resolver um problema genuinamente simples de uma forma simples, quando você vai além de simples, pense sobre expressões regulares.

Um grande número de ferramentas além Python incorporam expressões regulares como parte de sua funcionalidade. Unix orientado a linha de comando usa ferramentas como o 'grep', 'sed', e 'awk' são na sua maioria invólucros para processamento de expressões regulares. Muitos editores de texto permitem pesquisa e/ou de substituição com base em expressões regulares.

Muitas linguagens de programação, especialmente outras linguagens de script como Perl e TCL, constroem expressões regulares para o coração da linguagem. Mesmo a maioria de linha de comando shells, como o Bash ou o console do Windows, permite restritas expressões regulares, como parte de sua sintaxe de comando.

Há algumas variações na sintaxe da expressão regular entre diferentes ferramentas que usam, mas normalmente para a maior parte das expressões regulares são uma "pequena linguagem" que ficam embutidas dentro de línguas maiores, como Python. Os exemplos neste tutorial modelo (e na documentação no resto do capítulo) irão concentrar em sintaxe Python, mas a maior parte deste capítulo transferências e facilidades para trabalhar com outras linguagens de programação e ferramentas.

Tal como acontece com a maior parte deste livro, os exemplos são ilustrados pela utilização de sessões interativas do shell do Python que os leitores podem digitar, de modo que possam jogar com variações sobre os exemplos. No entanto, o [re] módulo tem pouca razão para incluir uma função que simplesmente ilustra jogos na casca. Portanto, a disponibilidade do programa de invólucro pequeno abaixo está implícito nos exemplos:

**#---------- re\_show.py ----------#**

**import re**

**def re\_show(pat, s):**

**print re.compile(pat, re.M).sub("{\g<0>}", s.rstrip()),'\n'**

**s = '''Mary had a little lamb**

**And everywhere that Mary**

**went, the lamb was sure**

**to go'''**

Coloque o código em um módulo externo e dê «import» nele. As novas expressões regulares não precisam se preocupar sobre o que a função acima faz por agora. É o suficiente para saber que o primeiro argumento para 're\_show ()’ será um padrão de expressão regular, e o segundo argumento será uma string a ser procurada. As partidas serão tratar cada linha da cadeia como um padrão separado para fins de correspondência começos e fins de linhas. As partidas ilustrados será o que está contido entre chaves (e é tipograficamente marcado para ênfase).

**TOPICO - Correspondência de padrões no texto: o básico**

O padrão muito mais simples acompanhado por uma expressão regular é um caractere literal ou uma sequência de caracteres literais. Nada no texto-alvo que consiste de exatamente essas personagens em exatamente a ordem listada irá corresponder. Um personagem não é minúscula idêntica à sua versão em maiúsculas, e vice-versa. Um espaço em uma expressão regular, por sinal, corresponde a um espaço literal no alvo (isto é, ao contrário a maioria das linguagens de programação ou de linha de comando ferramentas, onde um número variável de espaços separar palavras-chave).

**>>> from re\_show import re\_show, s**

**>>> re\_show('a', s)**

**M{a}ry h{a}d {a} little l{a}mb.**

**And everywhere th{a}t M{a}ry**

**went, the l{a}mb w{a}s sure**

**to go.**

**>>> re\_show('Mary', s)**

**{Mary} had a little lamb.**

**And everywhere that {Mary}**

**went, the lamb was sure**

**to go.**

Um número de caracteres têm significados especiais para expressões regulares. Um símbolo com um significado especial pode ser igualado, mas para isso ele deve ser prefixado com o caractere de barra invertida (isso inclui o caractere de barra invertida em si: para encontrar um barra invertida no alvo, a expressão regular deve incluir '\\'). Em Python, uma maneira especial de citar uma string é disponível que não irá realizar interpolação de string. Desde expressões regulares usam muitas das mesmas barra-prefixado códigos como fazer strings do Python, é geralmente mais fácil de compor expressão regular de strings citando-os como " strings cruas"

("raw strings") com um "r" na frente.

>>> from re\_show import re\_show

>>> s = '''Special characters must be escaped.\*'''

>>> re\_show(r'.\*', s)

{Special characters must be escaped.\*}

>>> re\_show(r'\.\\*', s)

Special characters must be escaped{.\*}

>>> re\_show('\\\\', r'Python \ escaped \ pattern')

Python {\} escaped {\} pattern

>>> re\_show(r'\\', r'Regex \ escaped \ pattern')

Regex {\} escaped {\} pattern

-\*-

Dois caracteres especiais são usados ​​para marcar o início e o fim de uma linha: acento circunflexo ("^") e cifrão ("$"). Para combinar com um acento circunflexo ou cifrão como um caractere literal, deve ser escapado (ou seja, precedê-lo por uma barra invertida "\").

Uma coisa interessante sobre o acento circunflexo e cifrão é que eles combinam padrões de largura zero. Isto é, o comprimento da cadeia acompanhado por um acento circunflexo ou cifrão por si só é zero (mas o resto da expressão regular pode ainda depender da correspondência de largura zero). Muitas ferramentas de expressões regulares fornecem outro padrão de largura zero por palavra-limite ("\b"). Palavras pode ser divididas por espaços em branco como espaços, tabulações, novas linhas, ou outros personagens como nulos, o padrão de palavra-limite corresponde o ponto real em que uma palavra começa ou termina, não em particular com caracteres em branco.

>>> from re\_show import re\_show, s

>>> re\_show(r'^Mary', s)

{Mary} had a little lamb

And everywhere that Mary

went, the lamb was sure

to go

>>> re\_show(r'Mary$', s)

Mary had a little lamb

And everywhere that {Mary}

went, the lamb was sure

to go

>>> re\_show(r'$','Mary had a little lamb')

Mary had a little lamb{}

Em expressões regulares, um ponto ``.`` pode representar qualquer caractere. Normalmente, o caractere de nova linha não está incluído, mas parâmetros opcionais podem forçar a inclusão do caractere de nova linha também (veja depois documentação de [re] funções do módulo). Utilizando um ponto em um padrão é uma forma de exigir que "algo" ocorre aqui, sem a necessidade de decidir o que.

Os leitores familiarizados com linha de comando do DOS curingas vontade conhecer o ponto de interrogação como no papel de "algum personagem" em máscaras de comando. Mas em expressões regulares, o ponto de interrogação tem um significado diferente, e o ponto é usado como um curinga.

>>> from re\_show import re\_show, s

>>> re\_show(r'.a', s)

{Ma}ry {ha}d{ a} little {la}mb

And everywhere t{ha}t {Ma}ry

went, the {la}mb {wa}s sure

to go

-\*-

Uma expressão regular pode ter caracteres literais na mesma e também de largura zero padrões posicionais. Cada personagem literal ou posicional padrão é um átomo em uma expressão regular. Pode-se também agrupar vários átomos juntos em uma expressão regular que é pequeno parte de uma expressão regular. Pode-se estar inclinado a chamar um tal agrupamento uma "molécula", mas normalmente é também chamado um átomo.

Nos antigos Unix orientados ferramentas como grep, subexpressões deve ser agrupados com parênteses escapados, por exemplo, '\(Mary\)'. Em Python (como acontece com a maioria das ferramentas mais recentes), o agrupamento é feito com parênteses nua, mas corresponder um parêntese literal requer escapa-lo no padrão.

>>> from re\_show import re\_show, s

>>> re\_show(r'(Mary)( )(had)', s)

{Mary had} a little lamb

And everywhere that Mary

went, the lamb was sure

to go

>>> re\_show(r'\(.\*\)', 'spam (and eggs)')

spam {(and eggs)}

-\*-

Em vez de nome de apenas um único caractere, um padrão em uma expressão regular pode corresponder a qualquer de um conjunto de caracteres.

Um conjunto de caracteres pode ser dado como uma lista simples dentro de praça colchetes, por exemplo, '[aeiou]' irá corresponder a única qualquer vogal minúscula única. Para intervalos de letra ou número que pode também ter o primeiro e última carta de um intervalo, com um traço no meio, por exemplo, '[A-Ma-m]' irá corresponder a qualquer letra minúscula ou maiúscula na primeira metade do alfabeto.

Python (como acontece com muitas ferramentas) fornece fuga estilo de atalhos para a classe de caracteres mais comumente utilizados, tais como '\s' para um espaço em branco e caracteres '\d' para um dígito. Pode-se sempre definir essas classes de personagens com colchetes, mas os atalhos podem fazer expressões regulares mais compactas e mais legíveis.

>>> from re\_show import re\_show, s

>>> re\_show(r'[a-z]a', s)

Mary {ha}d a little {la}mb

And everywhere t{ha}t Mary

went, the {la}mb {wa}s sure

to go

-\*-

O símbolo de acento circunflexo ``^` pode realmente ter dois significados diferentes em expressões regulares. Na maioria das vezes, isso significa que a coincidir com o comprimento zero teste padrão para início de linha. Mas, se for utilizado no início de uma classe de personagem, ele inverte o sentido da classe de personagem.

Tudo não incluído no conjunto de caracteres listados é correspondido.

**>>> from re\_show import re\_show, s**

**>>> re\_show(r'[^a-z]a', s)**

**{Ma}ry had{ a} little lamb**

**And everywhere that {Ma}ry**

**went, the lamb was sure**

**to go**

Usando classes de personagens é uma forma de indicar que quer uma coisa ou outra coisa pode ocorrer em um determinado local. Mas o que se quer especificar que qualquer um dos dois inteiros sub expressões pode ocorrer numa posição na expressão regular?

Para isso, use o operador de alternância, a barra vertical ("|"). Este é o símbolo que é também usado para indicar um pipe em Unix/DOS shells e às vezes é chamado o pipe.

O caractere pipe em uma expressão regular indica uma alternância entre tudo no grupo juntando-lo. O que significa que, mesmo que haja vários grupos para a esquerda e à direita de um personagem do caractere pipe, a alternância avidamente pergunta para tudo, de ambos os lados. Para selecionar o âmbito da alternância, você deve definir um grupo que englobe os padrões que podem corresponder. O exemplo ilustra isso:

**>>> from re\_show import re\_show**

**>>> s2 = 'The pet store sold cats, dogs, and birds.'**

**>>> re\_show(r'cat|dog|bird', s2)**

**The pet store sold {cat}s, {dog}s, and {bird}s.**

**>>> s3 = '=first first= # =second second= # =first= # =second='**

**>>> re\_show(r'=first|second=', s3)**

**{=first} first= # =second {second=} # {=first}= # ={second=}**

**>>> re\_show(r'(=)(first)|(second)(=)', s3)**

**{=first} first= # =second {second=} # {=first}= # ={second=}**

**>>> re\_show(r'=(first|second)=', s3)**

**=first first= # =second second= # {=first=} # {=second=}**

Uma das coisas mais poderosas e comuns que você pode fazer com expressões regulares é especificar quantas vezes um átomo ocorre em um termo completo regular. Às vezes você quer especificar algo sobre a ocorrência de um único personagem, mas muitas vezes você está interessado em especificar a ocorrência de uma classe de personagem ou uma sub expressão agrupada.

Há apenas um quantificador incluído com "básico" regular sintaxe de expressão, o asterisco ("\*"), em Inglês isto tem o significado "nenhum ou algum" ou "zero ou mais". Se você quiser especificar que qualquer número de um átomo pode ocorrer como parte de um padrão, siga o átomo por um asterisco.

Sem quantificadores, agrupar expressões realmente não servem muito como finalidade, mas uma vez que pode adicionar um quantificador a uma sub expressão podemos dizer algo sobre a ocorrência da sub expressão como um todo. Dê uma olhada no exemplo:

**>>> from re\_show import re\_show**

**>>> s = '''Match with zero in the middle: @@**

**... Subexpression occurs, but...: @=!=ABC@**

**... Lots of occurrences: @=!==!==!==!==!=@**

**... Must repeat entire pattern: @=!==!=!==!=@'''**

**>>> re\_show(r'@(=!=)\*@', s)**

**Match with zero in the middle: {@@}**

**Subexpression occurs, but...: @=!=ABC@**

**Lots of occurrences: {@=!==!==!==!==!=@}**

**Must repeat entire pattern: @=!==!=!==!=@**

**TÓPICO - padrões de correspondência no texto: Intermediário**

De certa maneira, a falta de qualquer símbolo quantificador após um átomo quantifica o átomo de qualquer maneira: Ele diz que o átomo ocorre exatamente uma vez.

Expressões regulares estendidas adicionam alguns outros números úteis para "uma vez que exatamente" e "zero ou mais vezes." O sinal de mais ("+") significa "uma ou mais vezes" e o ponto de interrogação ("?") significa "zero ou uma vez." Estes quantificadores são, de longe, as mais comuns enumerações que acabam usando.

Se você pensar sobre isso, você pode ver que as expressões regulares estendidas realmente não deixar você "dizer" qualquer coisa que o básico os que não fazem. Eles só permitem dizer que em um curto e mais forma legível. Por exemplo, '(ABC) +' é equivalente a '(ABC) (ABC) \*', e 'X(ABC)?Y' É equivalente a 'XABCY|XY'. Se os átomos de serem quantificados são eles próprios complicados agrupados subexpressões, o ponto de interrogação("?") e sinal de mais("+") podem fazer as coisas muito mais curtas e claras.

**>>> from re\_show import re\_show**

**>>> s = '''AAAD**

**... ABBBBCD**

**... BBBCD**

**... ABCCD**

**... AAABBBC'''**

**>>> re\_show(r'A+B\*C?D', s)**

**{AAAD}**

**{ABBBBCD}**

**BBBCD**

**ABCCD**

**AAABBBC**

Usando expressões regulares estendidas, você pode especificar um arbitrário padrão de contagem de ocorrência utilizando uma sintaxe mais detalhada do que um ponto de interrogação, sinal de mais, e quantificadores asterisco. O encaracolado chaves ("{" e "}") podem cercar uma contagem precisa do número de ocorrências que você está procurando.

A forma mais geral de quantificadores encaracolados por chave usa dois argumentos de intervalo (o primeiro não deve ser maior do que o segundo, e ambos devem ser inteiros não negativos). A contagem de ocorrência é especificada dessa maneira a situar-se entre os valores mínimo e máximo indicado (inclusive). Como taquigrafia, o argumento pode ser deixado vazio: Se assim for, o mínimo/máximo é especificado como zero/infinito, respectivamente. Se apenas um argumento é usado (sem vírgula lá), exatamente o número de ocorrências que são correspondidos.

**>>> from re\_show import re\_show**

**>>> s2 = '''aaaaa bbbbb ccccc**

**... aaa bbb ccc**

**... aaaaa bbbbbbbbbbbbbb ccccc'''**

**>>> re\_show(r'a{5} b{,6} c{4,8}', s2)**

**{aaaaa bbbbb ccccc}**

**aaa bbb ccc**

**aaaaa bbbbbbbbbbbbbb ccccc**

**>>> re\_show(r'a+ b{3,} c?', s2)**

**{aaaaa bbbbb c}cccc**

**{aaa bbb c}cc**

**{aaaaa bbbbbbbbbbbbbb c}cccc**

**>>> re\_show(r'a{5} b{6,} c{4,8}', s2)**

**aaaaa bbbbb ccccc**

**aaa bbb ccc**

**{aaaaa bbbbbbbbbbbbbb ccccc}**

Uma opção poderosa na criação de padrões de pesquisa é a especificação que uma sub-expressão que foi combinada anteriormente em uma expressão regular é compensada mais tarde na expressão. Fazemos isso usando referências anteriores. Referências anteriores são nomeados pelo números de 1 a 99, precedidos por uma barra invertida/escape caracter quando usado desta maneira. Essas referências anteriores referem-se para cada grupo sucessivo no padrão de correspondência, tal como em '(um) (dois) (três) \1\2\3'. Cada referência anterior numerada refere para o grupo que, no presente exemplo, tem a palavra correspondente para o número.

É importante notar algo ilustra o exemplo. O que fica acompanhada por uma referência anterior é a mesma sequência literal combinando a primeira vez que, mesmo se o padrão que corresponde ao string poderia ter acompanhado outras strings. Basta repetir a sub-expressão agrupados mesmo posteriormente na expressão regular faz não coincidir com os mesmos objetivos que usando uma referência anterior (mas você tem para decidir o que é que você realmente deseja corresponder em ambos os casos).

Referências anteriores remetem para o que ocorreu nas expressões anteriores agrupadas, na ordem dessas expressões agrupadas ocorreu. Até 99 referências anteriores numeradas podem ser utilizados. No entanto, Python também permite referências anteriores de nomenclatura, o que pode torná-lo muito mais claro o que as referências anteriores estão apontando. A inicial do grupo padrão deve começar com '? P <name>', e as correspondentes referências anteriores devem conter "(? P = nome)".

**>>> from re\_show import re\_show**

**>>> s2 = '''jkl abc xyz**

**... jkl xyz abc**

**... jkl abc abc**

**... jkl xyz xyz**

**... '''**

**>>> re\_show(r'(abc|xyz) \1', s2)**

**jkl abc xyz**

**jkl xyz abc**

**jkl {abc abc}**

**jkl {xyz xyz}**

**>>> re\_show(r'(abc|xyz) (abc|xyz)', s2)**

**jkl {abc xyz}**

**jkl {xyz abc}**

**jkl {abc abc}**

**jkl {xyz xyz}**

**>>> re\_show(r'(?P<let3>abc|xyz) (?P=let3)', s2)**

**jkl abc xyz**

**jkl xyz abc**

**jkl {abc abc}**

**jkl {xyz xyz}**

Quantificadores em expressões regulares são gananciosos. Isto é, eles correspondem tanto quanto possível.

Provavelmente o erro mais fácil fazer em compor expressões regulares é combinar demais. Quando você usa um quantificador, você quer combinar tudo (do tipo certo) até o ponto onde você quer terminar a sua combinação. Mas quando se utiliza o '\*', '+', Ou quantificadores numéricos, é fácil esquecer que a último pedaço que você está procurando pode ocorrer mais tarde, em uma linha do que você está interessado.

**>>> from re\_show import re\_show**

**>>> s2 = '''-- I want to match the words that start**

**... -- with 'th' and end with 's'.**

**... this**

**... thus**

**... thistle**

**... this line matches too much**

**... '''**

**>>> re\_show(r'th.\*s', s2)**

**-- I want to match {the words that s}tart**

**-- wi{th 'th' and end with 's}'.**

**{this}**

**{thus}**

**{this}tle**

**{this line matches} too much**

Muitas vezes, se você achar que as expressões regulares estão combinando muito, um procedimento útil é reformular o problema em sua mente. Em vez de pensar, "O que estou tentando corresponder mais tarde na expressão? "E se perguntar:" O que eu preciso para evitar correspondência na próxima parte? "Isto leva muitas vezes ao padrão mais parcimonioso de combinações. Muitas vezes, a maneira de evitar um padrão é usar o operador complementar em uma classe de personagem. Olhe o exemplo, e pense sobre como ele funciona.

O truque aqui é que existem duas formas diferentes de formulação de quase a mesma sequência. Ou você pode pensar que você quer manter correspondência-até chegar ao XYZ, ou você pode pensar que você quer manter correspondência, a menos que você começa a XYZ. Estes são sutilmente diferentes.

Para as pessoas que pensaram sobre probabilidade básica, o mesmo padrão ocorre. A chance de rolar um 6 em um dado em uma lista é 1/6. Qual é a chance de rolar um 6 em seis listas? Um ingênuo cálculo coloca as chances de 1/6+1/6+1/6+1/6+1/6+1/6, ou 100%. Isso é errado, claro (afinal, a chance após 12 listas não é de 200 por cento). O cálculo correto é: "Como evito rolar um 6 para seis listas? "(ou seja, 5/6\*5/6\*5/6\*5/6\*5/6\*5/6, ou cerca de 33 por cento). A chance de obter um 6 é a mesma chance que não evitá-la (ou cerca de 66 por cento). Na verdade, se você imaginar a transcrição de uma série listas de dados, você poderia aplicar uma expressão regular para a escrita pensamento registro, e semelhante se aplica.

**>>> from re\_show import re\_show**

**>>> s2 = '''-- I want to match the words that start**

**... -- with 'th' and end with 's'.**

**... this**

**... thus**

**... thistle**

**... this line matches too much**

**... '''**

**>>> re\_show(r'th[^s]\*.', s2)**

**-- I want to match {the words} {that s}tart**

**-- wi{th 'th' and end with 's}'.**

**{this}**

**{thus}**

**{this}tle**

**{this} line matches too much**

Nem todas as ferramentas que usam expressões regulares permitem modificar strings alvo. Alguns simplesmente localizam o padrão correspondente, a ferramenta de expressão regular mais amplamente utilizada é provavelmente grep, que é uma ferramenta para pesquisa apenas. Editores de texto, por exemplo, podem ou não permitir a substituição na expressão regular nas facilidades de pesquisa.

Python, sendo uma linguagem de programação geral, permite padrões de substituição sofisticados para acompanhar as combinações. Desde que strings do Python são imutáveis, [re] funções não modificam a cadeia de objetos no lugar, mas retornam as versões modificadas. Mas como com funções no [cadeia] módulo, sempre se pode religar um variável específica para o novo objeto string que resulta de [re] a modificação.

Exemplos de substituição neste tutorial irão chamar uma função 're\_new()', que é uma casca (wrapper) para o módulo de função 're.sub()'. Strings originais serão definidos acima a chamada, e os resultados modificados aparecerão abaixo da chamada e com o mesmo estilo de marcação adicional de áreas alteradas como 're\_show()' usado. Tenha o cuidado de observar que as chaves dos resultados apresentados não serão devolvidos por padrão [re], mas são apenas adicionados aqui para dar ênfase (como é a tipografia). Basta importar a função a seguir nos exemplos a seguir:

**#---------- re\_new.py ----------#**

**import re**

**def re\_new(pat, rep, s):**

**print re.sub(pat, '{'+rep+'}', s)**

Vamos dar uma olhada em alguns exemplos de modificação que constroem sobre o que já foi coberto. Este simplesmente substitui algum texto literal para algum outro texto literal. Note que “string.replace ()” pode conseguir o mesmo resultado, e será mais rápido ao fazê-lo.

**>>> from re\_new import re\_new**

**>>> s = 'The zoo had wild dogs, bobcats, lions, and other wild cats.'**

**>>> re\_new('cat','dog',s)**

**The zoo had wild dogs, bob{dog}s, lions, and other wild {dog}s.**

Na maioria das vezes, se você estiver usando expressões regulares para modificar um texto de destino, você vai querer corresponder aos padrões mais gerais do que apenas strings literais. Tudo o que é combinado é o que é substituído (mesmo que sejam várias cadeias diferentes no destino):

**>>> from re\_new import re\_new**

**>>> s = 'The zoo had wild dogs, bobcats, lions, and other wild cats.'**

**>>> re\_new('cat|dog','snake',s)**

**The zoo had wild {snake}s, bob{snake}s, lions, and other wild {snake}s.**

**>>> re\_new(r'[a-z]+i[a-z]\*','nice',s)**

**The zoo had {nice} dogs, bobcats, {nice}, and other {nice} cats.**

É bom ser capaz de inserir uma string fixa em toda parte um padrão que ocorra em um texto de destino. Mas, francamente, fazer isso é não muito sensível ao contexto. Muitas vezes, nós não queremos apenas para inserir strings fixas, mas sim de inserir algo que tem relação muito mais com os padrões de correspondência. Felizmente, referências anteriores veem em nosso socorro aqui. Pode-se usar referências anteriores no padrão combina-se, mas é ainda mais útil para ser capaz de utilizá-los em substituição padrões. Ao usar referências anteriores de substituição, pode-se escolher e escolher entre os padrões de correspondência para usar apenas as partes do interesse.

Bem como referências recursivas, os exemplos a seguir ilustram a importância do espaço em branco em expressões regulares. Na maioria código de programação, espaço em branco é meramente estético. Mas os exemplos diferem apenas em um espaço adicional dentro dos argumentos para a segunda chamada e o valor retornado é uma diferença importante.

**>>> from re\_new import re\_new**

**>>> s = 'A37 B4 C107 D54112 E1103 XXX'**

**>>> re\_new(r'([A-Z])([0-9]{2,4})',r'\2:\1',s)**

**{37:A} B4 {107:C} {5411:D}2 {1103:E} XXX**

**>>> re\_new(r'([A-Z])([0-9]{2,4}) ',r'\2:\1 ',s)**

**{37:A }B4 {107:C }D54112 {1103:E }XXX**

Este tutorial já advertiu sobre o perigo de combinar muito com padrões de expressões regulares. Mas o perigo é tanto mais grave quando se faz modificações, que é Vale a pena repetir. Se você substituir um padrão que corresponde a uma string maior do que você pensou quando compôs o padrão, você pode potencialmente excluir alguns dados importantes do seu alvo.

É sempre uma boa ideia para experimentar expressões regulares em dados alvo diversas que são representativos do uso de produção. Verifique se você está combinando o que você acha que você está combinando. Um quantificador perdido ou curinga pode fazer uma surpreendentemente grande variedade de textos coincidirem com o que você pensou que era um padrão específico. E às vezes você só tem que olhar para o seu padrão por um tempo, ou encontrar outro par de olhos, para descobrir o que realmente está acontecendo, mesmo depois de ver o que combina. Familiaridade pode gerar desprezo, mas também incute competência.

**TÓPICO - Avançado Extensões de Expressões Regulares**

Alguns acessórios muito úteis para as expressões regulares são fundamentais incluídos no Python (e em muitas outras ferramentas). Muitos destes não estritamente aumentam o poder de expressões regulares no Python, mas que-fazer-conseguir fazer expressá-los longe mais concisa e clara.

Anteriormente no tutorial, os problemas de correspondência eram muito discutidos, e algumas soluções foram sugeridas. Python é bom o suficiente para tornar isso mais fácil, fornecendo opcional "não-gananciosos" quantificadores. Estes quantificadores pegar o mínimo possível enquanto ainda combinar o que vem a seguir no padrão (Em vez de, tanto quanto possível).

Quantificadores não gananciosos têm a mesma sintaxe como os regulares gananciosos, exceto com o quantificador seguido por um ponto de interrogação. Por exemplo, um padrão não-ganancioso pode parecer: 'A[A-Z]\*?B'. Em Inglês, isso significa "corresponder a um A, seguido por apenas letras maiúsculas quantos forem necessárias até encontrar um B. "

Uma pequena coisa de olhar para fora é o fato de que o padrão "[A-Z]\*? '. sempre corresponderá letras maiúsculas zero. Já não as correspondências são sempre necessários para encontrar o seguinte "qualquer caractere" padrão. Se você usar não-gananciosos quantificadores, atente para correspondência muito pouco, o que constitui um perigo simétrico.

**>>> from re\_show import re\_show**

**>>> s = '''-- I want to match the words that start**

**... -- with 'th' and end with 's'.**

**... this line matches just right**

**... this # thus # thistle'''**

**>>> re\_show(r'th.\*s',s)**

**-- I want to match {the words that s}tart**

**-- wi{th 'th' and end with 's}'.**

**{this line matches jus}t right**

**{this # thus # this}tle**

**>>> re\_show(r'th.\*?s',s)**

**-- I want to match {the words} {that s}tart**

**-- wi{th 'th' and end with 's}'.**

**{this} line matches just right**

**{this} # {thus} # {this}tle**

**>>> re\_show(r'th.\*?s ',s)**

**-- I want to match {the words }that start**

**-- with 'th' and end with 's'.**

**{this }line matches just right**

**{this }# {thus }# thistle**

Modificadores podem ser usados ​​em expressões regulares ou como argumentos para muitas das funções de [re]. Um modificador afeta, de uma forma ou outra a interpretação de um padrão de expressão regular. Um modificador, ao contrário de um átomo, é global para o particular corresponder - em si mesmo, um modificador não corresponde qualquer coisa, em vez restringe ou direciona o que os átomos da correspondência.

Quando usado diretamente dentro de um padrão de expressão regular, um ou mais modificadores começar todo o padrão, tal como em "(? Limsux) '. por exemplo, para combinar com a palavra 'cat' independentemente do caso das letras, pode-se usar '(?i)cat'. Os mesmos modificadores podem ser passados como o último argumento como bitmasks (ou seja, com um '|' entre cada modificador), mas apenas para algumas funções no módulo[re], e não para todos. Por exemplo, as duas chamadas abaixo são equivalentes:

**>>> import re**

**>>> re.search(r'(?Li)cat','The Cat in the Hat').start()**

**4**

**>>> re.search(r'cat','The Cat in the Hat',re.L|re.I).start()**

**4**

No entanto, algumas funções chamadas em [re] não têm argumento para modificadores. Em tais casos, você deve usar o modificador prefixo pseudo-grupo ou pré-compilar a expressão regular em vez de usá-lo em forma de string. Por exemplo:

**>>> import re**

**>>> re.split(r'(?i)th','Brillig and The Slithy Toves')**

**['Brillig and ', 'e Sli', 'y Toves']**

**>>> re.split(re.compile('th',re.I),'Brillig and the Slithy Toves')**

**['Brillig and ', 'e Sli', 'y Toves']**

Veja a documentação do módulo [re] para obter detalhes sobre quais funções que podem tomar argumentos.

Os modificadores listados abaixo são utilizados em expressões[re]. Usuários de outras ferramentas de expressões regulares podem estar acostumados a um 'g' opção para o casamento de "global". Esses outros instrumentos tenham uma linha de texto como unidade padrão, e "globais" significa corresponder a várias linhas. Python tem a sequência real passado como unidade, de modo "global" é simplesmente o padrão. Para operar em um única linha, ou as expressões regulares têm de ser adaptados procurar apropriadas personagens que começam linha e fim de linha ou de strings  sendo operados deve ser dividido o primeiro a utilizar 'string.split ()' ou de outros meios.

**#\*--------- Regular expression modifiers ---------------#**

**\* L (re.L) - Locale customization of \w, \W, \b, \B**

**\* i (re.I) - Case-insensitive match**

**\* m (re.M) - Treat string as multiple lines**

**\* s (re.S) - Treat string as single line**

**\* u (re.U) - Unicode customization of \w, \W, \b, \B**

**\* x (re.X) - Enable verbose regular expressions**

A opção de linha única ("s") permite que o curinga para coincidir com um caractere de nova linha (que não irá de outro modo). A várias linhas opção ("m") faz com que "^" e "$" para coincidir com o início e o fim de cada linha da meta, e não apenas o início / fim do destino como um todo (o padrão). A opção insensível ("i") ignora as diferenças entre o caso de letras. O Locale e opções de Unicode ("L" e "u") dão diferentes interpretações para a palavra-limite ("\b") e alfanuméricos ("\w") escapou padrões - e suas formas inversas ("\B" e "\W").

A opção verbose ("x") é um pouco diferente das outras. Expressões regulares Verbose podem conter não significativa espaços em branco e comentários inline. Em certo sentido, isso também é apenas uma interpretação diferente de padrões de expressões regulares, mas ele permite que você produzir muito mais complexo facilmente legível padrões. Alguns exemplos a seguir nas seções abaixo.

   - \* -

Vamos dar uma primeira olhada em como diferenciar maiúsculas de minúsculas e single-line opções de mudar o comportamento partida.

**>>> from re\_show import re\_show**

**>>> s = '''MAINE # Massachusetts # Colorado #**

**... mississippi # Missouri # Minnesota #'''**

**>>> re\_show(r'M.\*[ise] ', s)**

**{MAINE # Massachusetts }# Colorado #**

**mississippi # {Missouri }# Minnesota #**

**>>> re\_show(r'(?i)M.\*[ise] ', s)**

**{MAINE # Massachusetts }# Colorado #**

**{mississippi # Missouri }# Minnesota #**

**>>> re\_show(r'(?si)M.\*[ise] ', s)**

**{MAINE # Massachusetts # Colorado #**

**mississippi # Missouri }# Minnesota #**

Olhando para trás, para a definição de "re\_show()", podemos vê-lo foi definida explicitamente usar a opção de várias linhas. Assim padrões exibidos com "re\_show()" será sempre de várias linhas. Vejamos alguns exemplos que usam "re.findall()' em vez disso.

**>>> from re\_show import re\_show**

**>>> s = '''MAINE # Massachusetts # Colorado #**

**... mississippi # Missouri # Minnesota #'''**

**>>> re\_show(r'(?im)^M.\*[ise] ', s)**

**{MAINE # Massachusetts }# Colorado #**

**{mississippi # Missouri }# Minnesota #**

**>>> import re**

**>>> re.findall(r'(?i)^M.\*[ise] ', s)**

**['MAINE # Massachusetts ']**

**>>> re.findall(r'(?im)^M.\*[ise] ', s)**

**['MAINE # Massachusetts ', 'mississippi # Missouri ']**

Caracteres de correspondência de texto e limites da palavra dependem exatamente o que é considerado como sendo alfanumérico. Páginas de código de caracteres para cartas fora do (US-Inglês) intervalo ASCII diferem entre alfabetos nacionais. Versões de Python são configuradas para uma localidade especial, e expressões regulares podem, opcionalmente, utilizar o atual para combinar palavras.

De maior significado a longo prazo é a capacidade do [re] módulo (depois de Python 2.0) para olhar para as categorias de Unicode caracteres, e decidir se um personagem é alfabética com base em nessa categoria. Configurações locais funcionam bem para diacríticos europeus, mas para não-romanos conjuntos, Unicode é o erro mais clara e menos propenso. O "u" modificador controla se caracteres alfabéticos Unicode são aqueles reconhecidos ou meramente ASCII:

**>>> import re**

**>>> alef, omega = unichr(1488), unichr(969)**

**>>> u = alef +' A b C d '+omega+' X y Z'**

**>>> u, len(u.split()), len(u)**

**(u'\u05d0 A b C d \u03c9 X y Z', 9, 17)**

**>>> ':'.join(re.findall(ur'\b\w\b', u))**

**u'A:b:C:d:X:y:Z'**

**>>> ':'.join(re.findall(ur'(?u)\b\w\b', u))**

**u'\u05d0:A:b:C:d:\u03c9:X:y:Z'**

Substituição de padrões em referência anterior é muito poderoso, mas é fácil de usar muitos grupos de uma expressão regular complexa, o que pode ser confuso para identificar. Muitas vezes, é mais legível para referir as partes de um padrão de substituição na ordem sequencial.

Para lidar com este problema, padrões de Python[re] permitem "agrupamento sem referências recursivas."

Um grupo que também não deveria ser tratada como uma referência anterior tem dois pontos e uma interrogação no início do grupo, tal como em '(:? padrão) ". Na verdade, você pode usar essa sintaxe, mesmo quando as referências anteriores estão no padrão de busca em si:

**>>> from re\_new import re\_new**

**>>> s = 'A-xyz-37 # B:abcd:142 # C-wxy-66 # D-qrs-93'**

**>>> re\_new(r'([A-Z])(?:-[a-z]{3}-)([0-9]\*)', r'\1\2', s)**

**{A37} # B:abcd:142 # {C66} # {D93}**

**>>> # Groups that are not of interest excluded from backref**

**...**

**>>> re\_new(r'([A-Z])(-[a-z]{3}-)([0-9]\*)', r'\1\2', s)**

**{A-xyz-} # B:abcd:142 # {C-wxy-} # {D-qrs-}**

**>>> # One could lose track of groups in a complex pattern**

**...**

Python oferece uma sintaxe particularmente útil para realmente complexa padrão referências anteriores. Ao invés de apenas jogar com o numeração dos grupos combinados, você pode dar-lhes um nome. Acima ressaltamos a sintaxe para referências anteriores nomeados na espaço de padrões, por exemplo, '(P = nome?) ". No entanto, um pouco diferentes sintaxe é necessária em padrões de substituição. Para isso, utilizamos operador '\g", juntamente com colchetes e um nome. Por exemplo:

**>>> from re\_new import re\_new**

**>>> s = "A-xyz-37 # B:abcd:142 # C-wxy-66 # D-qrs-93"**

**>>> re\_new(r'(?P<prefix>[A-Z])(-[a-z]{3}-)(?P<id>[0-9]\*)',**

**... r'\g<prefix>\g<id>',s)**

**{A37} # B:abcd:142 # {C66} # {D93}**

Outro truque de avançadas ferramentas de expressões regulares é "afirmações lookahead". Estes são semelhantes aos regulares agrupadas subexpressão, exceto eles realmente não peguam o que eles correspondem. Há duas vantagens em usar afirmações lookahead. Por um lado, uma afirmação lookahead pode funcionar em um forma semelhante a um grupo que não é backreferenced, isto é, é pode corresponder a algo, sem contar que em referências anteriores. De forma mais significativa, contudo, uma afirmação lookahead pode especificar

que a próxima parte de um padrão tem uma certa forma, mas deixe um subexpressão (mais geral) diferente, na verdade, agarrá-lo (geralmente para fins de referências recursivas que outros subexpressão).

Existem dois tipos de afirmações lookahead: positivo e negativo. Como seria de esperar, uma afirmação positiva especifica que algo vem que vem, e um negativo especifica de que algo não vem em seguida. enfatizando sua conexão com grupos não-backreferenced, a sintaxe para afirmações lookahead é semelhante: "(? = padrão)" para positiva afirmações, e '(?! padrão) "para afirmações negativas.

**>>> from re\_new import re\_new**

**>>> s = 'A-xyz37 # B-ab6142 # C-Wxy66 # D-qrs93'**

**>>> # Assert that three lowercase letters occur after CAP-DASH**

**...**

**>>> re\_new(r'([A-Z]-)(?=[a-z]{3})([\w\d]\*)', r'\2\1', s)**

**{xyz37A-} # B-ab6142 # C-Wxy66 # {qrs93D-}**

**>>> # Assert three lowercase letts do NOT occur after CAP-DASH**

**...**

**>>> re\_new(r'([A-Z]-)(?![a-z]{3})([\w\d]\*)', r'\2\1', s)**

**A-xyz37 # {ab6142B-} # {Wxy66C-} # D-qrs93**

Junto com declarações "olhar à frente (aheadassertions)", Python 2.0 + acrescenta declarações "olhar para trás (lookbehind)".A idéia é semelhante - um padrão é de interesse só se é (ou não é) precedido por outro padrão. Declarações lookbehind são um pouco mais restritas do que declarações lookahead porque eles só podem olhar para trás por um número fixo de posições de caracteres. Em outras palavras, nenhum quantificador geral é permitido em declarações lookbehind. Ainda assim, alguns padrões são mais facilmente expressos usando lookbehind afirmações.

Tal como acontece com declarações lookahead, afirmações lookbehind vêm com um sabor negativo e um positivo. O primeiro, assegura que um certo padrão não-preceder o jogo, este último assegura que o padrão-se-preceder a partida.

**>>> from re\_show import re\_show**

**>>> re\_show('Man', 'Manhandled by The Man')**

**{Man}handled by The {Man}**

**>>> re\_show('(?<=The )Man', 'Manhandled by The Man')**

**Manhandled by The {Man}**

**>>> re\_show('(?<!The )Man', 'Manhandled by The Man')**

**{Man}handled by The Man**

Nos exemplos mais tarde, começamos a ver o quão complicadas expressões regulares podem obter. Estes exemplos são não a metade. É possível fazer alguma quase absurdamente difícil de entender as coisas com expressão regular (mas eles são no entanto útil).

Há duas facilidades básicas que o modificador Python "verbose" ("x") usa para esclarecer expressões. Uma delas é permitir regulares expressões para continuar ao longo de várias linhas (ignorando espaço em branco como espaços à direita e novas linhas). A segunda é permitindo comentários dentro de expressões regulares. Quando os padrões são complicados, fazer as duas coisas!

O exemplo dado é um exemplo típico de um complicado, mas bem estruturado e bem comentado expressão, regular:

**>>> from re\_show import re\_show**

**>>> s = '''The URL for my site is: http://mysite.com/mydoc.html. You**

**... might also enjoy ftp://yoursite.com/index.html for a good**

**... place to download files.'''**

**>>> pat = r''' (?x)( # verbose identify URLs within text**

**... (http|ftp|gopher) # ter certeza de que encontrar um tipo de recurso**

**... :// # ...needs to be followed by colon-slash-slash**

**... [^ \n\r]+ # some stuff then space, newline, tab is URL**

**... \w # URL always ends in alphanumeric char**

**... (?=[\s\.,]) # assert: followed by whitespace/period/comma**

**... ) # end of match group'''**

**>>> re\_show(pat, s)**

**The URL for my site is: {http://mysite.com/mydoc.html}. You**

**might also enjoy {ftp://yoursite.com/index.html} for a good**

**place to download files.**

**SEÇÃO 1 - Algumas tarefas comuns**

**PROBLEMA: Fazendo um bloco de texto alinhado à esquerda**

Para clareza visual ou para identificar o papel de texto, os blocos de texto são muitas vezes recuado - especialmente em prosa orientadas documentos (arquivos de log, mas, arquivos de configuração e como também pode não utilizados têm campos iniciais). Para fins de jusante, recuo é muitas vezes irrelevantes, ou mesmo abertamente incorreta, uma vez que a reentrância não faz parte do próprio texto mas apenas uma decoração do texto. No entanto, muitas vezes faz as coisas ainda piores para executar a muito mais ingênuo transformação do texto recuado - simplesmente remover levando espaço em branco de cada linha. Enquanto recuo do bloco pode ser decoração, os recortes relativos das linhas dentro de blocos podem servir funções importantes ou essenciais (por exemplo, a blocos de texto pode ser Python código fonte).

O procedimento geral que você precisa tomar em maximamente não endentado um bloco de texto é bastante simples. Mas é fácil de colocar mais código para ele do que o necessário, e chegar a alguma deselegante e lentos loops aninhados de operações 'string.find ()' e 'String.Replace ()'. Um pouco de inteligência no uso normal de expressões, combinados com a concisão de um estilo de programação funcional(FP), pode dar-lhe uma rápida, curta e direta transformação.

**#---------- flush\_left.py ----------#**

**# Remove as many leading spaces as possible from whole block**

**from re import findall,sub**

**# What is the minimum line indentation of a block?**

**indent = lambda s: reduce(min,map(len,findall('(?m)^ \*(?=\S)',s)))**

**# Remove the block-minimum indentation from each line?**

**flush\_left = lambda s: sub('(?m)^ {%d}' % indent(s),'',s)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**import sys**

**print flush\_left(sys.stdin.read())**

A função 'flush\_left ()' assume que os blocos são recuados com espaços. Se tabs são usados ​​- ou usados combinado com espaços - uma passagem inicial através de "untabify.py 'o utilitário (que pode ser encontrada em '$PYTHONPATH/tools/scripts/') pode converter blocos para espaço somente recuo.

Um adjuvante útil para 'flush\_left ()' é provável que seja o 'reformat\_para()' função que foi apresentado no Capítulo 2, Problema 2. Entre os dois deles, você pode obter uma boa parte o caminho para um "grupo-orientada processador de texto." (Que outras capacidades seriam mais úteis?)

**PROBLEMA: Resumindo a opção de documentação da linha de comando**

Documentação de linha de comando opções para programas são geralmente em semi-formatos padrão, em lugares como manpages, docstrings, READMEs e afins. Em geral, dentro de documentação que esperar para ver as opções de linha de comando recuado um pouco, seguido por reentrância um pouco mais, seguido por uma ou mais linhas de descrição e geralmente terminava por uma linha em branco. Este estilo é legível para usuários que navegam documentação, mas é de suficientemente complexidade e variabilidade que expressões regulares são bem adaptadas para encontrar as descrições certas (métodos de cadeia simples aquém).

Um cenário específico onde você pode querer um resumo das de linha de comando é como uma ajuda para a compreensão arquivos de configuração que exigem filhos de comandos múltiplos. O arquivo '/ etc / inetd.conf' em sistemas Unix-like é um bom exemplo de tal um arquivo de configuração. Além disso, os arquivos de configuração frequentemente têm complexidade suficiente e variabilidade dentro deles de que os métodos de cadeia simples têm dificuldade em analisa-los.

O utilitário abaixo vai olhar para cada serviço lançado pela '/etc/inetd.conf' e presente a documentação resumo STDOUT de todas as opções utilizadas quando os serviços serão iniciados.

**#---------- show\_services.py ----------#**

**import re, os, string, sys**

**def show\_opts(cmdline):**

**args = string.split(cmdline)**

**cmd = args[0]**

**if len(args) > 1:**

**opts = args[1:]**

**# might want to check error output, so use popen3()**

**(in\_, out\_, err) = os.popen3('man %s | col -b' % cmd)**

**manpage = out\_.read()**

**if len(manpage) > 2: # found actual documentation**

**print '\n%s' % cmd**

**for opt in opts:**

**pat\_opt = r'(?sm)^\s\*'+opt+r'.\*?(?=\n\n)'**

**opt\_doc = re.search(pat\_opt, manpage)**

**if opt\_doc is not None:**

**print opt\_doc.group()**

**else: # try harder for something relevant**

**mentions = []**

**for para in string.split(manpage,'\n\n'):**

**if re.search(opt, para):**

**mentions.append('\n%s' % para)**

**if not mentions:**

**print '\n ',opt,' '\*9,'Option docs not found'**

**else:**

**print '\n ',opt,' '\*9,'Mentioned in below para:'**

**print '\n'.join(mentions)**

**else: # no manpage available**

**print cmdline**

**print ' No documentation available'**

**def services(fname):**

**conf = open(fname).read()**

**pat\_srv = r'''(?xm)(?=^[^#]) # lns that are not commented out**

**(?:(?:[\w/]+\s+){6}) # first six fields ignored**

**(.\*$) # to end of ln is servc launch'''**

**return re.findall(pat\_srv, conf)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**for service in services(sys.argv[1]):**

**show\_opts(service)**

As tarefas específicas realizadas por 'show\_opts()' e 'services()' são um tanto específico para sistemas Unix-like, mas o general técnicas são mais amplamente aplicáveis. Por exemplo, um caractere de comentário particular e número de campos em '/etc/inetd.conf' pode ser diferente para os scripts outros lançadores mas o uso de expressões regulares para encontrar os comandos de lançamento se aplica em outros lugares. Se o 'man' e 'col' utilitários não são no sistema relevante, você pode fazer algo equivalente, de modo como ler nas docstrings de módulos Python com semelhante descrições de opções (a maioria das amostras em '$PYTHONPATH/tools/' utilizar a documentação compatível, por exemplo).

Outra coisa a destacar é que, mesmo com expressões regulares são utilizadas na análise de alguns dados, você não precisa fazer tudo com expressões regulares. A operação simples 'string.split ()' para identificar parágrafos em 'show\_opts ()' ainda é o mais rápido e a técnica mais fácil, apesar de 're.split ()' poderia fazer o mesmo coisa.

Nota: Ao longo das linhas de divisão de parágrafo, aqui é um pensamento problema. O que é uma expressão regular que corresponde a cada conjunto parágrafo que contém em si algumas menores padrão 'pat'? para efeitos de quebra-cabeça, assumir que um parágrafo é um texto que ambos começa e termina com novas linhas duplicou ("\ n \ n").

**PROBLEMA: Detecção de palavras duplicadas**

Algums erros comuns de digitação nos textos em prosa são palavras é duplicadas (espero que eles foram editadas fora deste livro, exceto nos poucos casos quando se destinam). O mesmo erro ocorre em menor medida em código de linguagem de programação, arquivos de configuração, ou alimentações de dados. As expressões regulares são bem adequadas para detectar esta ocorrência, o que só equivale a uma referência anterior a um padrão de palavra. É fácil de embrulhar o regex em um pequeno utilitário com alguns recursos extras:

**#---------- dupwords.py ----------#**

**# Detect doubled words and display with context**

**# Include words doubled across lines but within paras**

**import sys, re, glob**

**for pat in sys.argv[1:]:**

**for file in glob.glob(pat):**

**newfile = 1**

**for para in open(file).read().split('\n\n'):**

**dups = re.findall(r'(?m)(^.\*(\b\w+\b)\s\*\b\2\b.\*$)', para)**

**if dups:**

**if newfile:**

**print '%s\n%s\n' % ('-'\*70,file)**

**newfile = 0**

**for dup in dups:**

**print '[%s] -->' % dup[1], dup[0]**

Esta versão particular, pega a linha ou linhas em que duplicatas ocorrem e as imprime para o contexto (juntamente com um prompt para a duplicar a si mesmo). Variações são simples. A suposição feita por 'dupwords.py "que é uma palavra que dobrou abrange uma linha (a partir da extremidade de um para o início de uma outra, ignorando os espaços em branco) é uma duplicação real, mas uma duplicata que parágrafos vãos não é igualmente digno de nota.

**PROBLEMA: Verificação de erros do servidor:**

Os servidores Web são uma fonte onipresente de hoje informação. Mas encontrar URLs que levam a documentos reais é largamente hit-or-miss. Cada mantenedor Web parece reorganizar seu site cada mês ou dois, quebrando assim bookmarks e hiperlinks. Tão ruim quanto o caos é para os internautas simples, é pior para robôs confrontados com a difícil tarefa de reconhecer a diferença entre conteúdo e erros. A-a-a, é fácil acumular páginas da Web baixados que consistem em erro mensagens em vez de conteúdo desejado.

Em princípio, os servidores Web podem e devem retornar códigos de erro indicando os erros do servidor. Mas, na prática, os servidores Web quase sempre retornar páginas geradas dinamicamente resultados para errônea pedidos. Tais páginas são basicamente perfeitamente normais páginas HTML que só acontecerá a conter texto como "Erro 404: O arquivo não encontrado! "Na maioria das vezes essas páginas são um pouco mais sofisticado do que este, contendo gráficos personalizados e layout, links para o site homepages, código JavaScript, cookies, meta tags, e todos os tipos de outras coisas. É realmente espantoso o quanto muitos servidores Web enviar em resposta a pedidos de inexistente URLs.

Abaixo está um script muito simples Python para examinar o que Web servidores retorno sobre solicitações válidas ou inválidas. Recebendo um erro página é geralmente tão simples como pedir uma página chamada 'Http://somewebsite.com/phony-url' ou semelhante (que nada realmente não existe). [Urllib] é discutido no Capítulo 5, mas seus detalhes não são importantes aqui.

**#---------- url\_examine.py ----------#**

**import sys**

**from urllib import urlopen**

**if len(sys.argv) > 1:**

**fpin = urlopen(sys.argv[1])**

**print fpin.geturl()**

**print fpin.info()**

**print fpin.read()**

**else:**

**print "No specified URL"**

Dada a diversidade de páginas de erro que você pode receber, é    difícil ou impossível para criar uma expressão regular (ou qualquer programa) que determine com certeza se um dado documento HTML é uma página de erro. Além disso, alguns sites escolhem gerar páginas que não são realmente bem um erro, mas tem realmente um conteúdo qualquer (por exemplo, listas genéricas de local informações com sugestões sobre como chegar ao conteúdo). Mas algumas heurísticas vêm muito perto de separar conteúdo de erros. Uma heurística digna de nota o que é interessante erros são quase sempre 404 ou 403 (não é uma coisa certa, mas bom o suficiente para fazer suposições inteligentes). Abaixo é um utilitário para classificar a "Probabilidade de erro" de documentos HTML:

**#---------- error\_page.py ----------#**

**import re, sys**

**page = sys.stdin.read()**

**# Mapping from patterns to probability contribution of pattern**

**err\_pats = {r'(?is)<TITLE>.\*?(404|403).\*?ERROR.\*?</TITLE>': 0.95,**

**r'(?is)<TITLE>.\*?ERROR.\*?(404|403).\*?</TITLE>': 0.95,**

**r'(?is)<TITLE>ERROR</TITLE>': 0.30,**

**r'(?is)<TITLE>.\*?ERROR.\*?</TITLE>': 0.10,**

**r'(?is)<META .\*?(404|403).\*?ERROR.\*?>': 0.80,**

**r'(?is)<META .\*?ERROR.\*?(404|403).\*?>': 0.80,**

**r'(?is)<TITLE>.\*?File Not Found.\*?</TITLE>': 0.80,**

**r'(?is)<TITLE>.\*?Not Found.\*?</TITLE>': 0.40,**

**r'(?is)<BODY.\*(404|403).\*</BODY>': 0.10,**

**r'(?is)<H1>.\*?(404|403).\*?</H1>': 0.15,**

**r'(?is)<BODY.\*not found.\*</BODY>': 0.10,**

**r'(?is)<H1>.\*?not found.\*?</H1>': 0.15,**

**r'(?is)<BODY.\*the requested URL.\*</BODY>': 0.10,**

**r'(?is)<BODY.\*the page you requested.\*</BODY>': 0.10,**

**r'(?is)<BODY.\*page.{1,50}unavailable.\*</BODY>': 0.10,**

**r'(?is)<BODY.\*request.{1,50}unavailable.\*</BODY>': 0.10,**

**r'(?i)does not exist': 0.10,**

**}**

**err\_score = 0**

**for pat, prob in err\_pats.items():**

**if err\_score > 0.9: break**

**if re.search(pat, page):**

**# print pat, prob**

**err\_score += prob**

**if err\_score > 0.90: print 'Page is almost surely an error report'**

**elif err\_score > 0.75: print 'It is highly likely page is an error report'**

**elif err\_score > 0.50: print 'Better-than-even odds page is error report'**

**elif err\_score > 0.25: print 'Fair indication page is an error report'**

**else: print 'Page is probably real content'**

Testado contra um bom número de sites, uma coleção como esta de pesquisas de expressões regulares e limitar confidências obras muito bem. Dentro do próprio julgamento do autor de apenas o que é realmente uma página de erro, 'erro\_page.py' não obteve nenhum falso positivo e sempre chegou a, pelo menos, o menor nível de aviso para cada página de erro verdadeiro.

Os padrões escolhidos são todos bem simples, e tanto os padrões e as respectivas ponderações foram determinadas inteiramente subjetivamente pelo autor. Mas algo assim como a ponderada a técnica de tentativa e erro pode ser usada para resolver muitos problemas de correspondência de "lógica nebulosa" (a maioria não tendo nada a ver com os erros do servidor Web).

O Código como o acima pode formar uma abordagem geral a mais para aplicações completas. Mas para o que vale a pena, nos scripts 'Url\_examine.py' e 'error\_page.py' pode ser utilizado diretamente em conjunto pelo pipe a partir do primeiro para o segundo. Por exemplo:

**#\*------ Using ex\_error\_page.py -----#**

**% python urlopen.py http://gnosis.cx/nonesuch | python ex\_error\_page.py**

**Page is almost surely an error report**

**PROBLEMA: Lendo linhas com caracteres de continuação**

Muitos arquivos de configuração e outros tipos de código de computador são orientados a linha , mas também têm uma facilidade para tratar várias linhas como se fossem uma única linha lógica. No processamento de um tal arquivo é geralmente desejável como um primeiro passo para ligar todos estes linhas lógicas em linhas delimitadas por quebras de linha reais (ou mais provavelmente, para transformar linhas tanto único e continuou como elementos da lista homogêneos para percorrer mais tarde). Um caractere de continuação é geralmente necessário para ser a -última- coisa em uma linha antes de uma nova linha, ou, possivelmente, a última coisa outras de alguns espaços em branco. Um pequeno (e muito parcial) da tabela de caracteres complementares utilizados por alguns comuns e incomuns formatos está listado abaixo:

**#\*----- Common continuation characters -----#**

**\ Python, JavaScript, C/C++, Bash, TCL, Unix config**

**\_ Visual Basic, PAW**

**& Lyris, IBIS**

**; Clipper, TOP**

**- XSPEC, NetREXX, COBOL**

**= Oracle Express**