Expressões Regulares em Python

Programação II

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática
Licenciatura em Tecnologias da Informação

Vasco Thudichum Vasconcelos

Maio 2016

Introdução

Expressões regulares descrevem conjuntos de palavras. Uma *palavra* é uma sequência de símbolos retirados de um dado *alfabeto*. Aos conjuntos de palavras dá-se o nome de *linguagem*. As expressões regulares que nos interessam descrevem padrões de procura, sendo frequentemente utilizadas em operações de procura-e-substituição. São construídas a partir dos símbolos do alfabeto, por meio de algumas operações básicas.

Estas notas endereçam a escrita e utilização de expressões regulares em Python, recorrendo à biblioteca `re` (Regular Expressions).

Conceitos básicos

Expressões regulares descrevem conjuntos de palavras. Um modo simples de descrever um *conjunto finito* de palavras (ou de quaisquer outros objectos) é por *extensão*, isto é, por enumeração dos seus objetos, como por exemplo, {`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`}. Podemos também descrever conjuntos por *compreensão* utilizando *expressões regulares*. Por exemplo, o conjunto acima pode ser descrito de uma forma mais concisa através da expressão regular `[0-9]`. Para além de descrever conjuntos finitos, as expressões regulares podem também descrever conjuntos infinitos. Por exemplo, `ab*` descreve de um modo sucinto o conjunto de todas as palavras que começam pela letra `a` e que são seguidas de zero ou mais ocorrências da letra `b`.

As expressões regulares são definidas sobre um alfabeto de símbolos. No caso da linguagem Python estamos interessados no alfabeto Unicode, cujos símbolos chamamos geralmente de carateres. As expressões regulares são definidas pelas seguintes regras:

C Qualquer carater é uma expressão regular. De notar que `C` denota um símbolo arbitrário e não necessariamente uma letra.

RS Se `R` e `S` são duas expressões regulares então a sua concatenação (justaposição) é uma expressão regular. Por exemplo `ab` é uma expressão regular que denota a linguagem com uma só palavra: {`ab`}.

R|S Uma barra vertical separa duas alternativas. Por exemplo, à expressão regular `ab|bc` está associada a linguagem com duas palavras: {`ab`, `bc`}.

R* O asterisco indica zero ou mais ocorrências do elemento `R`. Por exemplo, `ab*c` é uma expressão regular que denota uma linguagem infinita: {`ac`, `abc`, `abbc`, `abbbc`, ...}.

Há ainda a expressão regular ε que denota a palavra vazia, mas que não é utilizada diretamente em Python.

Por exemplo, o conjunto de todas as palavras compostas unicamente por algarismos pares pode ser descrita pela expressão regular `(0|2|4|6|8)*`. Notem a presença dos parentesis. Sem estes, a linguagem gerada pela expressão regular seria {`0`, `2`, `4`, `6`, ε, `8`, `88`, `888`, ...}.

Expressões regulares em Python

Como descobrir se uma dada palavra contém apenas algarismos pares? Começamos por importar o módulo `re`:

```
import re
```

Uma das funções presentes no módulo, `re.search(padrao,string)`, permite verificar se uma dada palavra (representada por uma *string* Python) contém um dado padrão. O padrão é dado por uma expressão regular, que é escrita também na forma de *string*, por exemplo: `'(0|2|4|6|8)*'`.

A função `re.search` devolve `None` se a *string* não tiver partes que pertençam à (linguagem associada à) expressão regular; caso contrário devolve um objeto `re.MatchObject` com informação adicional sobre as partes da *string* em que o padrão foi encontrado. Por exemplo, a instrução:

```
if re.search('(0|2|4|6|8)*', '64663422'):
    print 'Encontra'
else:
    print 'Não encontra'
```

imprime `Encontra`, apesar da *string* conter um `3`. É que a a função `re.search` procura ocorrências do padrão dado pela expressão regular, e existem várias ocorrências da expressão regular na *string* `'64663422'`, como por exemplo `'6466'` e `'422'`. Na realidade a instrução imprime `Encontra` seja qual for a *string*. Isto porque palavra vazia pertence à linguagem gerada pela expressão regular em causa e toda a *string* contém pelo menos uma *string* vazia.

Se estivermos interessados em que *toda* a string esteja coberta pela expressão regular temos de dizer que:

- Pretendemos que o padrão ocorra no início da *string*, para o qual utilizamos a expressão regular `^`, e que
- Gostariamos que o padrão termine no fim da *string*, para o qual utilizamos a expressão regular `$`.

Deste modo a chamada `re.search('^(0|2|4|6|8)*$','64663422')` devolve `None`, porque o padrão requer que a procura comece no início da *string* (`^`) e que termine no fim da *string* (`$`).

Imaginemos agora que pretendemos verificar se uma dada *string* representa uma das matrículas mais recentes dos automóveis em circulação em Portugal, isto é, matrículas compostas por dois algarismos, traço, duas letras maiúsculas, traço, dois algarismos. Poderiamos começar a escrever a expressão

```
$(0|1|2|3|4|5|6|7|8|9|)(0|1|2|3|4|5|6|7|8|9|)-...
```

mas ainda faltaria escrever a parte das duas letras e novamente a dos dois números.

Para ajudar nestes casos, as expressões regulares da linguagem Python vêm equipadas com algumas abreviaturas importantes. Uma delas é `[]`, o *conjunto de caracteres*:

- Os caracteres podem ser listados individualmente. Por exemplo podemos escrever `[02468]` em vez de `0|2|4|6|8`.
- Podemos descrever intervalos utilizando o primeiro e o último separado por um hifen. Por exemplo, um qualquer algarismo pode ser descrito com a expressão regular `[0-9]`, e um algarismo hexadecimal pode ser descrito por `[0-9A-Fa-f]`.
- Os caracteres especiais perdem o seu significado. Por exemplo, `[(+*)]` descreve qualquer um dos caracteres `(`, `+`, `*` ou `)`.
- O caracter `^` denota o complemento de um conjunto. Por exemplo a expressão regular `[^0-9]` qualquer carater diferente de `0`, ..., `9`.

Deste modo, as matrículas em que estamos interessados podem ser descritas pela seguinte expressão regular: `[0-9][0-9]-[A-Z][A-Z]-[0-9][0-9]`. Notem que o hifen tem um significado especial dentro dos conjuntos `[]`, mas que se trata de um caracter como outro qualquer fora dos conjuntos.

Há mais umas abreviaturas que podem simplificar a escrita de expressões:

- A expressão `{m}` especifica a repetição do padrão exactamente `m` vezes. Por exemplo `[0-9]{2}` abrevia `[0-9][0-9]`.
- A expressão `{m,n}` diz que o padrão pode ser repetido no mínimo `m` vezes e no máximo `n` vezes. Por exemplo, `a{3,5}` abrevia `aaa|aaaa|aaaaa`.

Agora podemos escrever uma expressão regular para matrículas de um modo mais compacto: `[0-9]{2}-[A-Z]{2}-[0-9]{2}`.

O exercício seguinte trata de escrever uma expressão regular para determinar se uma dada *string* representa ou não um número inteiro. Eis alguns exemplos: `0`, `-23`, `962`. Estamos então interessados em *strings* que começam com um hífen opcional e que contêm depois uma sequência não vazia de algarismos. Poderíamos escrever `^(-|ε)[0-9][0-9]*$`, mas há duas abreviaturas que nos podem ajudar:

- A expressão `+` especifica a repetição de um padrão *uma ou mais vezes* (relembrar que `*` indica repetição de zero ou mais vezes).
- A expressão `?` especifica a repetição de um padrão *zero ou uma vez*.

Deste modo podemos escrever a representação de números inteiros de uma forma mais compacta: `^-?[0-9]+$`. Este padrão reconhece o número `0023`, que geralmente não é aceite como número válido. Se não quisermos aceitar zeros à esquerda podemos ajustar o padrão para `^(0|-?[1-9][0-9]*)$`. Notem os parêntesis sem os quais a expressão regular reconheceria a *string* `'x1'`.

Imaginemos agora que estamos interessados em reconhecer *strings* que contenham pelo duas barras para trás, `\`. Mais precisamente procuramos uma barra, um número indeterminado de caracteres, uma barra. Acontece que a barra para trás tem um significado especial em Python. Trata-se de um caracter de fuga (um *escape caracter*), utilizado, por exemplo, para descrever mudanças de linha (`\n`) e caracteres de tabulação (`\t`). Por isso o caracter `\` aparece numa *string* com duas barras: `'\\'`. Eis alguns exemplos de *strings* que casam com o padrão: `'\\aa\\'`, `'b\\\\b'` e `'b\\a\\\\'`.

Para nos ajudar a resolver este problema, o módulo `re` apresenta algumas abreviaturas importantes:

- A expressão `.` denota qualquer caracter excepto uma mudança de linha.
- O caracter `\` é um caracter de fuga *das expressões regulares*. Por exemplo, para reconhecer uma sequência de zero ou mais `*` temos de escrever `'\**'`. Para reconhecer `\n` temos de escrever `'\\n'`, e para reconhecer uma barra para trás temos de utilizar a expressão regular `'\\\\'`.
- Para simplificar a escrita de expressões regulares o Python tem uma notação *raw string* na forma `r'...'`, onde as barras para trás não têm significado especial. Deste modo, `r'\n'` é uma expressão regular contendo *dois* caracteres (`\` e `n`), enquanto que `'\n'` é uma expressão regular contendo (*um* caracter (de mudança de linha, `\n`).
- A expressão `\d` denota um algarismo (ou dígito). Pode ser escrita `'\d'` ou `r'\d'`, porque `\d` não é sequência de fuga em Python.

Utilizando estas abreviaturas podemos resolver o nosso problema com a expressão regular `r'\\.*\\'` (ou com `'\\\\.*\\\\'` se não quisermos utilizar *raw strings*). Consultem a documentação Python sobre sequências de escape em *String literals*.

A seguinte tabela sumariza as expressões regulares Python mais importantes. Consulte a documentação `re` para mais expressões.

Operador	Interpretação
`C`	Qualquer caracter não especial corresponde a si próprio
`R|R`	Alternativa entre o `R` da esquerda e da direita
`RR`	Concatenação de duas expressões regulares
`R*`	Zero ou mais ocorrências de `R`
`R+`	Uma ou mais ocorrências de `R`
`R?`	Zero ou uma ocorrências de `R`
`R{N}`	`N` ocorrências de `R` concatenadas
`R{N,M}`	Entre `N` e `M` ocorrências de `R`
`.`	Corresponde a qualquer caracter exceto mudança de linha
`^`	Corresponde ao início da *string*
`$`	Corresponde ao fim da *string*
`[...]`	Conjuntos de carateres, por exemplo `[a-z]`
`[^...]`	O complementar de um conjunto de carateres
`\`	Carater de fuga (*escape*)
`\\`	Corresponde ao carater `\`
`(R)`	Introduz um grupo
`\N`	Corresponde ao grupo `N`; o primeiro grupo tem número 1
`\d`	O mesmo que `[0-9]`

Funções de procura de padrões

A função `re.search` pára quando encontra o primeiro padrão na *string*. Podemos utilizar a função para determinar se uma string representa uma matrícula:

```
def e_matricula (string):
  return re.search (r'^\d{2}-[A-Z]{2}-\d{2}$') != None
```

Quando o padrão está presente na *string*, a função `re.search` devolve um objecto `Match`. As operações disponíveis num objecto `match` incluem as seguintes:

- `group()` que dá o texto da primeira substring que casa com o padrão,
- `start()` que dá o índice do primeiro caracter onde o padrão foi encontrado, e
- `end()` que dá o índice carater a seguir ao último.

Suponhamos que pretendemos descobrir se uma dada string *contém* uma matrícula, e em caso positivo obter um triplo com o texto e com o índice do primeiro e do último caracter. Neste caso retiramos o carater de início de *string* (`^`) e de fim de *string* (`$`) da expressão regular das matrículas.

```
def detalhes_matricula (string):
  match = re.search(r'\d{2}-[A-Z]{2}-\d{2}', string)
  return None if match == None\
```

```
    else (match.group(), match.start(), match.end())
```

Por exemplo:

```
>>> contem_matricula ('12-AF-a70')
>>> contem_matricula ('12-AF-70')
('12-AF-70', 0, 8)
>>> contem_matricula ('A minha matricula: 12-AF-70')
('12-AF-70', 19, 27)
```

Para além de reconhecer padrões, a operação `re.search` também pode classificar os vários *grupos* existentes numa expressão regular. A expressão regular `r'^\d{2}-[A-Z]{2}-\d{2}$'` tem apenas um grupo, aquele que "apanha" a expressão toda. Podemos no entanto utilizar parentesis para indentificar os grupos de interesse. No caso das matrículas estamos interessado em três grupos: dois de algarismos e um de letras. Neste caso utilizamos uma expressão regular com três grupos de parentesis: `r'^(\d{2})-([A-Z]{2})-(\d{2})$'`.

```
def decompoe_matricula (string):
  match = re.search(r'^(\d{2})-([A-Z]{2})-(\d{2})$', string)
  return None if match == None\
    else (match.group(1), match.group(2), match.group(3))
```

Por exemplo:

```
>>> decompoe_matricula ('12-AF-70')
('12', 'AF', '70')
```

A função `re.search` pára quando encontra a primeira *substring* que casa com o padrão. Por vezes estamos interessados em obter *todas* as *substrings* que casam com um dado padrão. Neste caso utilizamos a função `re.findall`. Os parâmetros são os mesmos da função `re.search`, isto é, o padrão e a *string*. A função devolve uma lista com todas as *substrings* que encontrou. No caso da expressão regular conter mais do que um grupo, cada elemento na lista é um tuplo contendo os vários grupos. Por exemplo:

```
>>> acidente = '12-AF-70 colidiu com 43-PQ-98 que embateu em
    98-EF-00'
>>> re.findall(r'\d{2}-[A-Z]{2}-\d{2}', acidente)
['12-AF-70', '43-PQ-98', '98-EF-00']
>>> re.findall(r'(\d{2})-([A-Z]{2})-(\d{2})', acidente)
[('12', 'AF', '70'), ('43', 'PQ', '98'), ('98', 'EF', '00')]
```

A função `findall` não dá informação sobre os índices iniciais e finais das ocorrências, bem como outra informação disponível em objectos `Match` devolvidos pela função `re.search`. Para extra flexibilidade usamos a função `re.finditer`, que devolve um iterador sobre objetos `re.MatchObject`. Os iteradores são geralmente exercitados utilizando um ciclo **`for`**. Para obter todas as matrículas e os índices de início e de fim podemos utilizar o seguinte código.

```
def detalhes_matriculas (string):
    matches = re.finditer(r'\d{2}-[A-Z]{2}-\d{2}', string)
    resultado = []
    for match in matches:
        resultado.append((match.group(), match.start(),
          match.end()))
    return resultado
```

Por exemplo:

```
>>> detalhes_matriculas(acidente)
[('12-AF-70', 0, 8), ('43-PQ-98', 21, 29),
 ('98-EF-00', 45, 53)]
```

As funções `re.search`, `re.findall` e `re.finditer` devolvem sempre ocorrências *máximas* e *sem sobreposição* do padrão. Máximas porque uma *string* pode conter muitas *subtrings* que casem com o padrão. Por exemplo, a *string* `'ab'` tem quatro *subtrings* que casam com o padrão `'(a|b)*'`. São elas `''`, `'a'`, `'b'` e `'ab'`; a função `re.findall` devolve apenas a ocorrência `'ab'`. Sem sobreposição porque as funções não voltam a trás para procurar mais padrões. Por exemplo `re.search('\d\d', '123').group()` devolve `'12'` apenas.

Substituição de padrões

Outra tarefa comum é procurar e alterar parte de uma *string* utilizando expressões regulares. Podemos por exemplo, normalizar endereços postais, alterar endereços de email antigos ou trocar a ordem de algum texto. Tudo isto é possível utilizando a função `re.sub`. A função recebe três parâmetros obrigatórios: o padrão, o padrão de substituição e a *string*. Devolve uma nova *string* com a alteração efectuada.

Imaginemos que queremos alterar datas da forma `'maio 3'` para `'3 de maio`. A expressão regular `r'([a-zA-Z]+) (\d+)'` identifica um grupo de uma ou mais letras, `([a-zA-Z]+)`, e um segundo grupo de um ou mais algarismos, `(\d+)`. Para especificar o padrão de substituição usamos as expressões `\1` e `\2` para identificar cada um destes grupos: `\1` identifica o primeiro grupo na expressão regular e `\2` o segundo.

Por exemplo:

```
>>> feira = 'A feira de decorreu entre maio 3 e junho 4'
>>> re.sub(r'([a-zA-Z]+) (\d+)', r'\2 de \1', feira)
'A feira de decorreu entre 3 de maio e 4 de junho'
```

Compilação de padrões para aumento de desempenho

A execução de qualquer uma das funções estudadas até aqui passa por uma fase inicial na qual a expressão regular é compilada numa representação (um automato) que permite mais rapidamente efectuar as operações de pesquisa e simulação.

Se pretendemos utilizar muitas vezes o mesmo padrão pode compensar compilá-lo, guardar o resultado numa variável, e depois utilizá-lo várias vezes. Para compilar uma expressão regular utilizamos a função `re.compile(padrão)`.

Por exemplo, em vez de escrever

```
print re.search(r'\d{2}-[A-Z]{2}-\d{2}', '43-PQ-98')
```

podemos escrever

```
expreg = re.compile(r'\d{2}-[A-Z]{2}-\d{2}', string)
print expreg.search('43-PQ-98')
```

Notem que neste caso utilizamos as funções `search`, `findall`, etc, sobre o objeto `expreg`, mas omitimos o parâmetro do padrão.

Imaginemos que, dado um ficheiro com endereços de correio electrónicos, pretendemos atualizar os emails dos alunos, transformando o domínio de `alunos.fc.ul.pt` para `alunos.ciencias.ulisboa.pt`. Todos os outros endereços de email não devem ser alterados. Os endereços de email estão guardados num ficheiro e a função `converte_email` deve guardar os novos números num outro ficheiro. A função `sub` irá ser chamada tantas vezes quantas o número de linhas no ficheiro, pelo que compensa compilar primeiro a expressão regular.

```
def converte_email (antes, depois):
  expreg = re.compile(r'(fc\d{5}@)alunos.fc.ul.pt')
  with open(antes) as leitor:
    with open(depois, mode = 'w') as escritor:
      for linha in leitor:
        escritor.write(expreg.sub(
          r'\1alunos.ciencias.ulisboa.pt', linha))
```

Para saber mais

- A documentação do módulo Regular Expression Operations
- Mark Lutz, Programming Python, 4th Edition, O'Reilly, 2011. Capítulo 19.