CAPÍTULO 7

CORRESPONDÊNCIA DE PADRÕES COM EXPRESSÕES REGULARES

Talvez você já esteja acostumado a pesquisar um texto pressionando CTRL-F e digitando as palavras que estiver procurando. As *expressões regulares* vão um passo além: elas permitem especificar um *padrão* de texto a ser procurado. Talvez você não saiba exatamente o número de um telefone comercial, porém, se morar nos Estados Unidos ou no Canadá, saberá que ele contém três dígitos seguidos de um hífen e depois mais quatro dígitos

(e, opcionalmente, um código de área de três dígitos no início). É assim que você como ser humano reconhece um número de telefone quando o vê: 415-555-1234 é um número de telefone, porém 4.155.551.234 não é.

As expressões regulares são úteis, mas muitos que não são programadores as desconhecem, apesar de os editores e processadores de texto mais modernos como o Microsoft Word ou o OpenOffice terem recursos de pesquisa e de pesquisa e substituição que possam fazer buscas baseadas em expressões regulares. As expressões regulares permitem economizar bastante tempo não só para os usuários de software, mas também para os programadores. Com efeito, o autor de obras técnicas Cory Doctorow argumenta que, mesmo antes de ensinar programação, devíamos ensinar expressões regulares:

> *Conhecer [as expressões regulares] pode significar a diferença entre resolver um problema em três passos e resolvê-lo em 3 mil passos. Quando se é um nerd, você esquece que os problemas que resolvemos com o pressionamento de algumas teclas podem exigir dias de trabalho lento, maçante e suscetível a erros de outras pessoas.*[1] [2]

Neste capítulo, começaremos criando um programa para encontrar padrões de texto *sem* usar expressões regulares e então veremos como usar essas expressões para deixar o código muito mais compacto. Mostrarei como fazer

correspondências básicas usando expressões regulares e, em seguida, prosseguirei apresentando alguns recursos mais eficazes como substituição de strings e criação de suas próprias classes de caracteres. Por fim, no final do capítulo, criaremos um programa que poderá extrair automaticamente números de telefone e endereços de email de um bloco de texto.

Encontrando padrões de texto sem usar expressões regulares

Suponha que você queira encontrar um número de telefone em uma string. Você conhece o padrão: três números, um hífen, três números, um hífen e quatro números. Aqui está um exemplo: 415-555-4242.

Vamos usar uma função chamada `isPhoneNumber()` para verificar se uma string corresponde a esse padrão, retornando `True` ou `False`. Abra uma nova janela no editor de arquivo e insira o código a seguir; salve o arquivo como *isPhoneNumber.py*:

```
  def isPhoneNumber(text):
u     if len(text) != 12:
          return False
      for i in range(0, 3):
v         if not text[i].isdecimal():
              return False
w     if text[3] != '-':
          return False
      for i in range(4, 7):
x         if not text[i].isdecimal():
              return False
y     if text[7] != '-':
          return False
      for i in range(8, 12):
z         if not text[i].isdecimal():
              return False
{     return True

  print('415-555-4242 is a phone number:')
  print(isPhoneNumber('415-555-4242'))
  print('Moshi moshi is a phone number:')
```

```
print(isPhoneNumber('Moshi moshi'))
```

Quando esse programa for executado, a saída terá o seguinte aspecto:

```
415-555-4242 is a phone number:
True
Moshi moshi is a phone number:
False
```

A função isPhoneNumber() contém um código que realiza diversas verificações para saber se a string em text é um número de telefone válido. Se alguma dessas verificações falhar, a função retornará False. Inicialmente, o código verifica se a string tem exatamente 12 caracteres u. Em seguida, verifica se o código de área (ou seja, os três primeiros caracteres em text) é constituído somente de caracteres numéricos v. O restante da função verifica se a string está de acordo com o padrão de um número de telefone: o número deve ter um primeiro hífen após o código de área w, deve ter mais três caracteres numéricos x, depois outro hífen y e, por fim, mais quatro números z. Se a execução do programa conseguir passar por todas essas verificações, True será retornado {.

Chamar isPhoneNumber() com o argumento '415-555-4242' retornará True. Chamar isPhoneNumber() com 'Moshi moshi' retornará False; o primeiro teste falha, pois 'Moshi moshi' não tem um tamanho igual a 12 caracteres.

Será necessário adicionar mais código ainda para encontrar esse padrão de texto em uma string maior. Substitua as quatro últimas chamadas à função print() em *isPhoneNumber.py* por:

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'
for i in range(len(message)):
u    chunk = message[i:i+12]
v    if isPhoneNumber(chunk):
        print('Phone number found: ' + chunk)
print('Done')
```

Quando esse programa for executado, a saída terá o seguinte aspecto:

```
Phone number found: 415-555-1011
Phone number found: 415-555-9999
Done
```

A cada iteração do loop for, uma nova porção de 12 caracteres de message é atribuída à variável chunk u. Por exemplo, na primeira iteração, i será 0 e chunk receberá message[0:12] (ou seja, a string 'Call me at 4'). Na próxima iteração, i será 1 e chunk receberá message[1:13] (a string 'all me at 41').

Passamos chunk a isPhoneNumber() para verificar se ela corresponde ao padrão de número de telefone v e, em caso afirmativo, essa porção da string será exibida.

Continue a percorrer message em um loop e, em algum momento, os 12 caracteres em chunk corresponderão a um número de telefone. O loop percorrerá a string toda, testando cada porção de 12 caracteres e exibindo qualquer chunk que satisfaça isPhoneNumber(). Após termos acabado de percorrer message, exibimos Done.

Embora a string em message seja curta nesse exemplo, ela poderia ter milhões de caracteres de tamanho e o programa continuaria executando em menos de um segundo. Um programa semelhante que encontra números de telefone usando expressões regulares também seria executado em menos de um segundo, porém as expressões regulares agilizam a escrita desses programas.

Encontrando padrões de texto com expressões regulares

O programa anterior para encontrar números de telefone funciona, porém utiliza bastante código para fazer algo limitado: a função isPhoneNumber() contém 17 linhas, porém é capaz de identificar somente um padrão de número de telefone. O que aconteceria se um número de telefone estivesse formatado como 415.555.4242 ou como (415) 555-4242? E se o número de telefone tivesse uma extensão, por exemplo, 415-555-4242 x99? A função isPhoneNumber() falharia ao validar esses números. Poderíamos acrescentar mais código para esses padrões adicionais, porém há uma maneira mais simples de resolver esse problema.

As expressões regulares (regular expressions), chamadas de *regexes* por questões de concisão, correspondem a descrições para um padrão de texto. Por exemplo, um \d é uma regex que representa um dígito – ou seja, qualquer numeral único entre 0 e 9. A regex \d\d\d-\d\d\d-\d\d\d\d é usada pelo Python

para fazer a correspondência do mesmo texto conforme feito pela função isPhoneNumber(): uma string com três números, um hífen, mais três números, outro hífen e quatro números. Qualquer outra string não corresponderá à regex \d\d\d-\d\d\d-\d\d\d\d.

No entanto as expressões regulares podem ser muito mais sofisticadas. Por exemplo, acrescentar um 3 entre chaves ({3}) após um padrão é como dizer "faça a correspondência desse padrão três vezes". Desse modo, a regex \d{3}-\d{3}-\d{4}, um pouco mais concisa, também corresponde ao formato correto de número de telefone.

Criando objetos Regex

Todas as funções de regex em Python estão no módulo re. Digite o seguinte no shell interativo para importar esse módulo:

```
>>> import re
```

NOTA A maioria dos próximos exemplos neste capítulo exigirá o módulo re, portanto lembre-se de importá-lo no início de qualquer script que você criar ou sempre que reiniciar o IDLE. Caso contrário, uma mensagem de erro NameError: name 're' is not defined (NameError: nome 're' não está definido) será obtida.

Passar um valor de string que represente sua expressão regular a re.compile() fará um objeto Regex de padrão ser retornado (ou, simplesmente, um objeto Regex).

Para criar um objeto Regex que corresponda ao padrão de número de telefone, insira o seguinte no shell interativo. (Lembre-se de que \d quer dizer "um caractere correspondente a um dígito" e \d\d\d-\d\d\d-\d\d\d\d é a expressão regular para o padrão correto de número de telefone.)

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

Agora a variável phoneNumRegex contém um objeto Regex.

PASSANDO STRINGS PURAS PARA RE.COMPILE( )

Lembre-se de que os caracteres de escape em Python usam a barra invertida (\). O valor de string '\n' representa um único caractere de quebra de linha, e não uma barra invertida seguida de um *n* minúsculo. É preciso fornecer o caractere de escape \\ para exibir uma única barra invertida. Portanto '\\n' é a string que representa uma barra invertida seguida de um *n* minúsculo. Entretanto, ao colocar um r antes do primeiro caractere de aspas do valor da string, podemos marcar a string como *pura* (raw string), que não considera caracteres de escape.

Como as expressões regulares geralmente utilizam barras invertidas, é conveniente passar strings puras para a função re.compile() em vez de digitar barras invertidas extras. Digitar r'\d\d\d-\d\d\d-\d\d\d\d' é muito mais fácil do que digitar '\\d\\d\\d-\\d\\d\\d-\\d\\d\\d\\d'.

Objetos Regex de correspondência

O método search() de um objeto Regex pesquisa a string recebida em busca de qualquer correspondência com a regex. O método search() retornará None se o padrão da regex não for encontrado na string. Se o padrão *for* encontrado, o método search() retornará um objeto Match. Objetos Match têm um método group() que retornará o texto correspondente extraído da string pesquisada. (Explicarei os grupos em breve.) Por exemplo, digite o seguinte no shell interativo:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found: ' + mo.group())
Phone number found: 415-555-4242
```

O nome da variável mo é somente um nome genérico usado em objetos Match. Esse exemplo pode parecer complicado à primeira vista, porém é muito mais conciso que o programa *isPhoneNumber.py* anterior e faz o mesmo.

Nesse caso, passamos o padrão desejado a re.compile() e armazenamos o objeto Regex resultante em phoneNumRegex. Em seguida, chamamos search() em phoneNumRegex e lhe passamos a string em que queremos encontrar uma correspondência. O resultado da pesquisa será armazenado na variável mo.

Nesse exemplo, sabemos que nosso padrão será encontrado na string, portanto sabemos que um objeto Match será retornado. Sabendo que mo contém um objeto Match e não o valor nulo None, podemos chamar group() em mo para retornar a correspondência. Escrever mo.group() em nossa instrução print exibirá a correspondência completa, ou seja, 415-555-4242.

Revisão da correspondência com expressão regular

Embora haja diversos passos para usar expressões regulares em Python, cada passo é bem simples.

1. Importe o módulo de regex usando import re.
2. Crie um objeto Regex usando a função re.compile(). (Lembre-se de usar uma string pura.)
3. Passe a string que você quer pesquisar ao método search() do objeto Regex. Isso fará um objeto Match ser retornado.
4. Chame o método group() do objeto Match para retornar uma string com o texto correspondente.

NOTA Apesar de incentivá-lo a digitar o código de exemplo no shell interativo, você também deve usar ferramentas de teste de expressões regulares baseadas em web, que poderão mostrar exatamente como uma regex faz a correspondência de uma porção de texto especificada. Recomendo usar a ferramenta de teste em *http://regexpal.com/*.

Mais correspondência de padrões com expressões regulares

Agora que os passos básicos para criar e encontrar objetos correspondentes a expressões regulares em Python já são conhecidos, você está pronto para experimentar alguns dos recursos mais eficazes da correspondência de padrões.

Agrupando com parênteses

Suponha que você queira separar o código de área do restante do número de telefone. A adição de parênteses criará *grupos* na regex: (\d\d\d)-(\d\d\d-\d\d\d\d). Então você poderá usar o método group() do objeto de

correspondência para obter o texto correspondente de apenas um grupo.

O primeiro conjunto de parênteses em uma string de regex será o grupo 1. O segundo conjunto será o grupo 2. Ao passar o inteiro 1 ou 2 ao método group() do objeto de correspondência, podemos obter partes diferentes do texto correspondente. Ao passar 0 ou nada ao método group(), o texto correspondente completo será retornado. Digite o seguinte no shell interativo:

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

Se quiser obter todos os grupos de uma só vez, utilize o método groups() – observe a forma do plural no nome.

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

Como mo.groups() retorna uma tupla com diversos valores, podemos usar o truque da atribuição múltipla para atribuir cada valor a uma variável diferente, como na linha areaCode, mainNumber = mo.groups() anterior.

Os parênteses têm um significado especial em expressões regulares, porém o que devemos fazer se for necessário corresponder a parênteses em seu texto? Por exemplo, talvez os números de telefone aos quais você esteja tentando corresponder tenham o código de área definido entre parênteses. Nesse caso, será necessário escapar os caracteres (e) com uma barra invertida. Digite o seguinte no shell interativo:

```
>>> phoneNumRegex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

Os caracteres de escape \(e \) na string pura passada para re.compile() corresponderão aos caracteres de parênteses propriamente ditos.

Fazendo a correspondência de vários grupos com pipe

O caractere | é chamado de *pipe*. Podemos usá-lo em qualquer lugar em que quisermos fazer a correspondência de uma entre várias expressões. Por exemplo, a expressão regular r'Batman|Tina Fey' corresponde a 'Batman' ou a 'Tina Fey'.

Quando *tanto* Batman *quanto* Tina Fey ocorrerem na string pesquisada, a primeira ocorrência do texto correspondente será retornada como o objeto Match. Digite o seguinte no shell interativo:

```
>>> heroRegex = re.compile (r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey.')
>>> mo1.group()
'Batman'

>>> mo2 = heroRegex.search('Tina Fey and Batman.')
>>> mo2.group()
'Tina Fey'
```

NOTA Podemos encontrar *todas* as ocorrências correspondentes com o método findall() que será discutido na seção “Método findall()”.

O pipe também pode ser usado para fazer a correspondência de um entre diversos padrões como parte de sua regex. Por exemplo, suponha que você queira fazer a correspondência de qualquer uma das strings 'Batman', 'Batmobile', 'Batcopter' e 'Batbat'. Como todas essas strings começam com Bat, seria interessante se você pudesse especificar esse prefixo somente uma vez. Isso pode ser feito com parênteses. Digite o seguinte no shell interativo:

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
```

```
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1)
'mobile'
```

A chamada ao método mo.group() retorna o texto correspondente 'Batmobile' completo, enquanto mo.group(1) retorna somente a parte do texto correspondente dentro do primeiro grupo de parênteses, ou seja, 'mobile'. Ao usar o caractere pipe e os parênteses de agrupamento, podemos especificar diversos padrões alternativos aos quais você gostaria que sua regex correspondesse.

Se houver necessidade de fazer a correspondência de um caractere de pipe propriamente dito, escape-o com uma barra invertida, como em \|.

Correspondência opcional usando ponto de interrogação

Às vezes, há um padrão ao qual você quer corresponder somente de forma opcional. Isso quer dizer que a regex deve encontrar uma correspondência independentemente de essa porção de texto estar ou não presente. O caractere ? marca o grupo que o antecede como sendo uma parte opcional do padrão. Por exemplo, digite o seguinte no shell interativo:

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

A parte da expressão regular que contém (wo)? significa que o padrão wo é um grupo opcional. A regex corresponderá a textos que não tenham nenhuma ou que tenham uma instância de wo. É por isso que a regex corresponde tanto a 'Batwoman' quanto a 'Batman'.

Usando o exemplo anterior com o número de telefone, podemos fazer a regex procurar números de telefone que tenham ou não um código de área.

Digite o seguinte no shell interativo:

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>> mo1 = phoneRegex.search('My number is 415-555-4242')
>>> mo1.group()
'415-555-4242'

>>> mo2 = phoneRegex.search('My number is 555-4242')
>>> mo2.group()
'555-4242'
```

Você pode pensar no ? como se dissesse “faça a correspondência de zero ou de uma ocorrência do grupo que antecede esse ponto de interrogação”.

Se houver necessidade de fazer a correspondência de um caractere de ponto de interrogação propriamente dito, escape-o com \?.

Correspondendo a zero ou mais ocorrências usando asterisco

O * (chamado de *asterisco*) quer dizer “corresponda a zero ou mais” – o grupo que antecede o asterisco pode ocorrer qualquer número de vezes no texto. Esse grupo poderá estar totalmente ausente ou ser repetido diversas vezes. Vamos dar uma olhada no exemplo contendo Batman novamente.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'

>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

Para 'Batman', a parte referente a (wo)* da regex corresponde a zero instâncias de wo na string; para 'Batwoman', (wo)* corresponde a uma instância de wo; e para 'Batwowowowoman', (wo)* corresponde a quatro instâncias de wo.

Se houver necessidade de fazer a correspondência do caractere asterisco propriamente dito, utilize uma barra invertida antes do asterisco na expressão regular, ou seja, *.

Correspondendo a uma ou mais ocorrências usando o sinal de adição

Enquanto * quer dizer "corresponda a zero ou mais", o + (ou *sinal de adição*) quer dizer "corresponda a um ou mais". De modo diferente do asterisco, que não exige que seu grupo esteja presente na string correspondente, o grupo que antecede um sinal de adição deve aparecer *pelo menos uma vez*. Ele não é opcional. Digite o seguinte no shell interativo e compare o resultado com as regexes que usam asterisco da seção anterior:

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'

>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'

>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

A regex Bat(wo)+man não identificará uma correspondência na string 'The Adventures of Batman', pois pelo menos um wo é exigido pelo sinal de adição.

Se houver necessidade de fazer a correspondência de um sinal de adição propriamente dito, insira uma barra invertida antes desse caractere para escapá-lo, ou seja, use \+.

Correspondendo a repetições específicas usando chaves

Se você tiver um grupo que deseja repetir um número específico de vezes, insira um número entre chaves após o grupo em sua regex. Por exemplo, a regex (Ha){3} corresponde à string 'HaHaHa', mas não a 'HaHa', pois essa última

tem apenas duas repetições do grupo (Ha).

Em vez de um número, podemos especificar um intervalo especificando um mínimo, uma vírgula e um máximo entre chaves. Por exemplo, a regex (Ha){3,5} corresponde a 'HaHaHa', 'HaHaHaHa' e 'HaHaHaHaHa'.

Também podemos deixar de fora o primeiro ou o segundo número nas chaves para deixar de especificar o mínimo ou o máximo. Por exemplo, (Ha){3,} corresponderá a três ou mais instâncias do grupo (Ha), enquanto (Ha){,5} corresponderá a zero até cinco instâncias. As chaves podem ajudar a deixar suas expressões regulares mais concisas. As duas expressões regulares a seguir correspondem a padrões idênticos:

```
(Ha){3}
(Ha)(Ha)(Ha)
```

As duas expressões regulares a seguir também correspondem a padrões idênticos:

```
(Ha){3,5}
((Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha)(Ha))
```

Digite o seguinte no shell interativo:

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'

>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

Nesse caso, (Ha){3} corresponde a 'HaHaHa', mas não a 'Ha'. Como não há correspondência em 'Ha', search() retorna None.

Correspondências greedy e nongreedy

Como (Ha){3,5} pode corresponder a três, quatro ou cinco instâncias de Ha na string 'HaHaHaHaHa', você pode estar se perguntando por que a chamada a group() do objeto Match no exemplo anterior com chaves retorna 'HaHaHaHaHa',

e não as possibilidades mais curtas. Afinal de contas, 'HaHaHa' e 'HaHaHaHa' também são correspondências válidas para a expressão regular (Ha){3,5}.

As expressões regulares em Python são *greedy* (gulosas) por default, o que significa que, em situações ambíguas, a correspondência será feita com a maior string possível. Na versão *nongreedy* (não gulosa) das chaves, que faz a correspondência com a menor string possível, um ponto de interrogação é usado depois da chave de fechamento.

Digite o seguinte no shell interativo e observe a diferença entre as formas greedy e nongreedy das chaves em que a mesma string é pesquisada:

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'

>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

Observe que o ponto de interrogação pode ter dois significados em expressões regulares: declarar uma correspondência nongreedy ou indicar um grupo opcional. Esses significados não têm nenhuma relação entre si.

Método findall()

Além do método search(), os objetos Regex também têm um método findall(). Enquanto search() retorna um objeto Match do *primeiro* texto correspondente na string pesquisada, o método findall() retorna as strings de *todas* as correspondências na string pesquisada. Para ver como search() retorna um objeto Match somente da primeira instância do texto correspondente, digite o seguinte no shell interativo:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>> mo.group()
'415-555-9999'
```

Por outro lado, findall() não retorna um objeto Match, mas uma lista de strings – *desde que não haja grupos na expressão regular*. Cada string da lista é uma parte do texto pesquisado que correspondeu à expressão regular. Digite o seguinte no shell interativo:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # não tem nenhum grupo
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

Se *houver* grupos na expressão regular, findall() retornará uma lista de tuplas. Cada tupla representa uma correspondência identificada, e seus itens serão as strings correspondentes a cada grupo da regex. Para ver findall() em ação, digite o seguinte no shell interativo (observe que a expressão regular sendo compilada agora contém grupos entre parênteses):

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # tem grupos
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[('415', '555', '9999'), ('212', '555', '0000')]
```

Para resumir o que o método findall() retorna, lembre-se do seguinte:

1. Quando chamado em uma regex sem grupos, por exemplo, \d\d\d-\d\d\d-\d\d\d\d, o método findall() retorna uma lista de strings correspondentes, como ['415-555-9999', '212-555-0000'].
2. Quando chamado em uma regex que tenha grupos, por exemplo, (\d\d\d)-(\d\d\d)-(\d\d\d\d), o método findall() retorna uma lista de tuplas contendo strings (uma string para cada grupo), como em [('415', '555', '1122'), ('212', '555', '0000')].

Classes de caracteres

No exemplo anterior de regex para número de telefone, aprendemos que \d pode representar qualquer dígito, ou seja, \d é uma versão abreviada da expressão regular (0|1|2|3|4|5|6|7|8|9). Há várias dessas *classes abreviadas de caracteres*, conforme mostrado na tabela 7.1.

Tabela 7.1 – Códigos abreviados para classes comuns de caracteres

Classe de caracteres	Representa

abreviada	
\d	Qualquer dígito de 0 a 9.
\D	Qualquer caractere que *não* seja um dígito de 0 a 9.
\w	Qualquer letra, dígito ou o caractere underscore. (Pense nisso como uma correspondência de caracteres de "palavra".)
\W	Qualquer caractere que *não* seja uma letra, um dígito ou o caractere underscore.
\s	Qualquer espaço, tabulação ou caractere de quebra de linha. (Pense nisso como uma correspondência de caracteres de "espaço".)
\S	Qualquer caractere que *não* seja um espaço, uma tabulação ou uma quebra de linha.

As classes de caracteres são convenientes para reduzir as expressões regulares. A classe de caracteres [0-5] corresponderá somente aos números de 0 a 5; isso é muito mais conciso do que digitar (0|1|2|3|4|5).

Por exemplo, digite o seguinte no shell interativo:

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7 swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6 geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

A expressão regular \d+\s\w+ corresponderá a textos que tenham um ou mais dígitos (\d+) seguidos de um caractere de espaço em branco (\s) seguido de um ou mais caracteres que sejam letra/dígito/underscore (\w+). O método findall() retorna todas as strings que correspondam ao padrão da regex em uma lista.

Criando suas próprias classes de caracteres

Haverá ocasiões em que você vai querer fazer a correspondência de um conjunto de caracteres, porém as classes abreviadas de caracteres (\d, \w, \s e assim por diante) serão amplas demais. Você pode definir sua própria classe de caracteres usando colchetes. Por exemplo, a classe de caracteres [aeiouAEIOU] corresponderá a qualquer vogal, tanto minúscula quanto maiúscula. Digite o seguinte no shell interativo:

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

Também é possível incluir intervalos de letras ou de números usando um hífen. Por exemplo, a classe de caracteres [a-zA-Z0-9] corresponderá a todas as letras minúsculas, às letras maiúsculas e aos números.

Observe que, nos colchetes, os símbolos normais de expressão regular não são interpretados. Isso quer dizer que não é necessário escapar os caracteres ., *, ? ou () com uma barra invertida na frente. Por exemplo, a classe de caracteres [0-5.] corresponderá aos dígitos de 0 a 5 e um ponto. Não é preciso escrever essa classe como [0-5\.].

Ao inserir um acento circunflexo (^) logo depois do colchete de abertura da classe de caracteres, podemos criar uma *classe negativa de caracteres*. Uma classe negativa de caracteres corresponderá a todos os caracteres que *não* estejam na classe de caracteres. Por exemplo, digite o seguinte no shell interativo:

```
>>> consonantRegex = re.compile(r'[^aeiouAEIOU]')
>>> consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')
['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.', ' ', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

Agora, em vez de fazer a correspondência de todas as vogais, estamos fazendo a correspondência de todos os caracteres que não sejam uma vogal.

Acento circunflexo e o sinal de dólar

O símbolo de acento circunflexo (^) também pode ser usado no início de uma regex para indicar que uma correspondência deve ocorrer no *início* de um texto pesquisado. Da mesma maneira, podemos colocar um sinal de dólar ($) no final da regex para indicar que a string deve *terminar* com esse padrão de regex. Além disso, podemos usar ^ e $ juntos para indicar que a string toda deve corresponder à regex – ou seja, não é suficiente que uma correspondência seja feita com algum subconjunto da string.

Por exemplo, a string r'^Hello' de expressão regular corresponde a strings que comecem com 'Hello'. Digite o seguinte no shell interativo:

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

A string r'\d$' de expressão regular corresponde a strings que terminem com um caractere numérico de 0 a 9. Digite o seguinte no shell interativo:

```
>>> endsWithNumber = re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')
<_sre.SRE_Match object; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None
True
```

A string r'^\d+$' de expressão regular corresponde a strings que comecem e terminem com um ou mais caracteres numéricos. Digite o seguinte no shell interativo:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')
>>> wholeStringIsNum.search('1234567890')
<_sre.SRE_Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12  34567890') == None
True
```

As duas últimas chamadas a search() no exemplo anterior com o shell interativo mostram como a string toda deve corresponder à regex se ^ e $ forem utilizados.

Sempre confundo o significado desses dois símbolos; sendo assim, utilizo o mnemônico “Carrots cost dollars” para me lembrar de que o acento circunflexo (caret) vem antes e o sinal de dólar vem depois.

Caractere-curinga

O caractere . (ou *ponto*) em uma expressão regular é chamado de *caractere-curinga* e corresponde a qualquer caractere, exceto uma quebra de linha. Por exemplo, digite o seguinte no shell interativo:

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

Lembre-se de que o caractere ponto corresponderá somente a um caractere, motivo pelo qual a correspondência para o texto flat no exemplo anterior foi feita somente com lat. Para fazer a correspondência de um ponto propriamente dito, escape-o com uma barra invertida, ou seja, use \..

Correspondendo a tudo usando ponto-asterisco

Às vezes, vamos querer fazer uma correspondência de tudo. Por exemplo, suponha que você queira fazer a correspondência da string 'First Name:' seguida de todo e qualquer texto seguido de 'Last Name:' e, por fim, seguida de qualquer caractere novamente. Podemos usar ponto-asterisco (.*) para indicar "qualquer caractere". Lembre-se de que o caractere ponto quer dizer "qualquer caractere único, exceto a quebra de linha" e o caractere asterisco quer dizer "zero ou mais ocorrências do caractere anterior".

Digite o seguinte no shell interativo:

```
>>> nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')
>>> mo.group(1)
'Al'
>>> mo.group(2)
'Sweigart'
```

O ponto-asterisco utiliza o modo *greedy*: ele sempre tentará fazer a correspondência do máximo de texto possível. Para corresponder a todo e qualquer texto em modo *nongreedy*, utilize ponto, asterisco e ponto de interrogação (.*?). Assim como no caso das chaves, o ponto de interrogação diz ao Python para fazer a correspondência em modo nongreedy.

Digite o seguinte no shell interativo para ver a diferença entre as versões greedy e nongreedy:

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>'
```

```
>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man> for dinner.>'
```

Ambas as regexes, de modo geral, podem ser traduzidas como “faça a correspondência de um sinal de ‘menor que’ seguido de qualquer caractere seguido de um sinal de ‘maior que’”. Porém a string '<To serve man> for dinner.>' tem duas correspondências possíveis para o sinal de ‘maior que’. Na versão nongreedy da regex, o Python faz a correspondência com a menor string possível: '<To serve man>'. Na versão greedy, o Python faz a correspondência com a maior string possível: '<To serve man> for dinner.>'.

Correspondendo a quebras de linha com o caractere ponto

O ponto-asterisco corresponderá a qualquer caractere, exceto uma quebra de linha. Ao passar re.DOTALL como segundo argumento de re.compile(), podemos fazer o caractere ponto corresponder a *todos* os caracteres, incluindo o caractere de quebra de linha.

Digite o seguinte no shell interativo:

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'

>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

A regex noNewlineRegex, criada sem que re.DOTALL tenha sido passado para a chamada a re.compile(), corresponderá a qualquer caractere até o primeiro caractere de quebra de linha, enquanto newlineRegex, que *teve* re.DOTALL passado para re.compile(), corresponderá a todos os caracteres. É por isso que a chamada a newlineRegex.search() corresponde à string completa, incluindo seus

caracteres de quebra de linha.

Revisão dos símbolos de regex

Este capítulo discutiu bastante a notação; sendo assim, apresentaremos uma revisão rápida do que aprendemos:

- ? corresponde a zero ou uma ocorrência do grupo anterior.
- * corresponde a zero ou mais ocorrências do grupo anterior.
- + corresponde a uma ou mais ocorrências do grupo anterior.
- {n} corresponde a exatamente *n* ocorrências do grupo anterior.
- {n,} corresponde a *n* ou mais ocorrências do grupo anterior.
- {,m} corresponde a zero até *m* ocorrências do grupo anterior.
- {n,m} corresponde a no mínimo *n* e no máximo *m* ocorrências do grupo anterior.
- {n,m}? ou *? ou +? faz uma correspondência nongreedy do grupo anterior.
- ^spam quer dizer que a string deve começar com *spam*.
- spam$ quer dizer que a string dever terminar com *spam*.
- . corresponde a qualquer caractere, exceto os caracteres de quebra de linha.
- \d, \w e \s correspondem a um dígito, um caractere de palavra ou um caractere de espaço, respectivamente.
- \D, \W e \S correspondem a qualquer caractere, exceto um dígito, um caractere de palavra ou um caractere de espaço, respectivamente.
- [abc] corresponde a qualquer caractere que estiver entre os colchetes (por exemplo, *a*, *b* ou *c*).
- [^abc] corresponde a qualquer caractere que não esteja entre os colchetes.

Correspondências sem diferenciar letras maiúsculas de minúsculas

Normalmente, as expressões regulares fazem correspondência de textos com o tipo exato de letra, ou seja, maiúscula ou minúscula, que você especificar. Por exemplo, as regexes a seguir fazem a correspondência de strings totalmente diferentes:

```
>>> regex1 = re.compile('RoboCop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
>>> regex4 = re.compile('RobocOp')
```

Entretanto, às vezes, você estará preocupado somente em fazer a correspondência das letras, sem se importar se elas são maiúsculas ou minúsculas. Para fazer sua regex ignorar as diferenças entre letras maiúsculas e minúsculas (ser case-insensitive), re.IGNORECASE ou re.I pode ser passado como segundo argumento de re.compile(). Digite o seguinte no shell interativo:

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('RoboCop is part man, part machine, all cop.').group()
'RoboCop'

>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'

>>> robocop.search('Al, why does your programming book talk about robocop so much?').group()
'robocop'
```

Substituindo strings com o método sub()

As expressões regulares não só podem identificar padrões de texto como também podem substituir esses padrões por novos textos. O método sub() dos objetos Regex recebe dois argumentos. O primeiro argumento é uma string para substituir qualquer correspondência. O segundo é a string para a expressão regular. O método sub() retorna uma string com as substituições aplicadas.

Por exemplo, digite o seguinte no shell interativo:

```
>>> namesRegex = re.compile(r'Agent \w+')
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
'CENSORED gave the secret documents to CENSORED.'
```

Às vezes, pode ser necessário utilizar o próprio texto correspondente como parte da substituição. No primeiro argumento de sub(), podemos digitar \1, \2, \3 e assim por diante para dizer "insira o texto do grupo 1, 2, 3 e assim por diante na substituição".

Por exemplo, suponha que você queira censurar os nomes dos agentes secretos mostrando apenas as primeiras letras de seus nomes. Para isso, podemos usar a regex Agent (\w)\w* e passar r'\1****' como o primeiro argumento de sub(). \1 nessa string será substituído por qualquer texto correspondente no grupo 1 – ou seja, o grupo (\w) da expressão regular.

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')
>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent Eve
knew Agent Bob was a double agent.')
A**** told C**** that E**** knew B**** was a double agent.'
```

Administrando regexes complexas

As expressões regulares serão convenientes se o padrão de texto para a correspondência for simples. Porém fazer a correspondência de padrões complicados de texto pode exigir expressões regulares longas e confusas. Podemos atenuar esse problema dizendo à função re.compile() que ignore espaços em branco e comentários na string de expressão regular. Esse "modo verbose" pode ser habilitado se a variável re.VERBOSE for passada como segundo argumento de re.compile().

Agora, em vez de uma expressão regular difícil de ler como:

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}(\s*
(ext|x|ext.)\s*\d{2,5})?)')
```

podemos distribuir a expressão regular em várias linhas usando comentários como:

```
phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?          # código de área
    (\s|-|\.)?                  # separador
    \d{3}                       # primeiros 3 dígitos
    (\s|-|\.)                   # separador
    \d{4}                       # últimos 4 dígitos
```

```
    (\s*(ext|x|ext.)\s*\d{2,5})?  # extensão
    )'", re.VERBOSE)
```

Observe como o exemplo anterior utiliza a sintaxe de aspas triplas (''') para criar uma string de múltiplas linhas de modo que a definição da expressão regular possa ser distribuída em diversas linhas, tornando-a muito mais legível.

As regras para comentários em uma string de expressão regular são as mesmas usadas no código Python normal: o símbolo # e tudo que estiver depois dele até o final da linha serão ignorados. Além disso, os espaços extras na string de múltiplas linhas da expressão regular não serão considerados como parte do padrão de texto para a correspondência. Isso permite organizar a expressão regular para que ela se torne mais fácil de ler.

Combinando re.IGNORECASE, re.DOTALL e re.VERBOSE

O que aconteceria se você quisesse usar re.VERBOSE para escrever comentários em sua expressão regular, mas também quisesse utilizar re.IGNORECASE para ignorar as diferenças entre letras maiúsculas e minúsculas? Infelizmente, a função re.compile() aceita apenas um único valor como segundo argumento. Podemos contornar essa limitação combinando as variáveis re.IGNORECASE, re.DOTALL e re.VERBOSE utilizando o caractere pipe (|) que, nesse contexto, é conhecido como o operador *ou bit a bit* (bitwise or).

Portanto, se quiser uma expressão regular que ignore as diferenças entre letras maiúsculas e minúsculas *e* inclua quebras de linha para que correspondam ao caractere ponto, sua chamada a re.compile() deverá ser feita da seguinte forma:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

As três opções para o segundo argumento terão o aspecto a seguir:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL |
re.VERBOSE)
```

Essa sintaxe é um pouco antiga e tem origem nas primeiras versões do Python. Os detalhes sobre os operadores bit a bit estão além do escopo deste livro, porém dê uma olhada nos recursos em *http://nostarch.com/automatestuff/* para obter mais informações. Também podemos passar outras opções para o segundo argumento; elas são incomuns, porém você poderá ler mais sobre elas também no site de recursos.

Projeto: extrator de números de telefone e de endereços de email

Suponha que você tenha a tarefa maçante de localizar todos os números de telefone e endereços de email em uma página web ou um documento extenso. Se fizer rolagens manualmente pela página, você poderá acabar fazendo a pesquisa por bastante tempo. Porém, se você tivesse um programa que pudesse pesquisar o texto em seu clipboard em busca de números de telefone e de endereços de email, seria possível simplesmente pressionar CTRL-A para selecionar todo o texto, CTRL-C para copiá-lo para o clipboard e então executar o seu programa. Ele poderia substituir o texto no clipboard somente pelos números de telefone e pelos endereços de email encontrados.

Sempre que estiver diante de um novo projeto, pode ser tentador mergulhar diretamente na escrita do código. No entanto, com muita frequência, será melhor dar um passo para trás e considerar o quadro geral. Recomendo inicialmente definir um plano geral para o que seu programa deverá fazer. Não pense ainda no código propriamente dito – você poderá se preocupar com ele depois. Neste momento, atenha-se aos aspectos mais gerais.

Por exemplo, seu extrator de números de telefone e de endereços de email deverá fazer o seguinte:

• Obter o texto do clipboard.

• Encontrar todos os números de telefone e os endereços de email no texto.

• Colá-los no clipboard.

Agora você poderá começar a pensar em como isso funcionará no código. O código deverá fazer o seguinte:

• Usar o módulo pyperclip para copiar e colar strings.

- Criar duas regexes: uma para corresponder a números de telefone e outra para endereços de email.
- Encontrar todas as correspondências, e não apenas a primeira, para ambas as regexes.
- Formatar as strings correspondentes de forma elegante em uma única string a ser colada no clipboard.
- Exibir algum tipo de mensagem caso nenhuma correspondência tenha sido encontrada no texto.

Essa lista é como um roadmap (mapa) do projeto. À medida que escrever o código, você poderá focar em cada um desses passos separadamente. Cada passo é razoavelmente administrável e está expresso em termos de tarefas que você já sabe fazer em Python.

Passo 1: Criar uma regex para números de telefone

Inicialmente, você deve criar uma expressão regular para procurar números de telefone. Crie um arquivo novo, digite o código a seguir e salve-o como *phoneAndEmail.py*:

```
#! python3
# phoneAndEmail.py – Encontra números de telefone e endereços de email no clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?            # código de área
    (\s|-|\.)?                    # separador
    (\d{3})                       # primeiros 3 dígitos
    (\s|-|\.)                     # separador
    (\d{4})                       # últimos 4 dígitos
    (\s*(ext|x|ext.)\s*(\d{2,5}))?    # extensão
    )''', re.VERBOSE)

# TODO: Cria a regex para email.

# TODO: Encontra correspondências no texto do clipboard.

# TODO: Copia os resultados para o clipboard.
```

Os comentários TODO são apenas um esqueleto para o programa. Eles serão substituídos à medida que você escrever o código propriamente dito.

O número de telefone começa com um código de área *opcional*, portanto o grupo para código de área é seguido de um ponto de interrogação. Como o código de área pode ter apenas três dígitos (isto é, \d{3}) *ou* três dígitos entre parênteses (isto é, \(\d{3}\)), deve haver um pipe unindo essas partes. Você pode adicionar o comentário # código de área na regex para essa parte da string de múltiplas linhas; isso ajudará a se lembrar a que (\d{3}|\(\d{3}\))? deve corresponder.

O caractere separador do número de telefone pode ser um espaço (\s), um hífen (-) ou um ponto (.), portanto essas partes também devem ser unidas por pipes. As próximas partes da expressão regular são simples: três dígitos seguidos de outro separador seguido de quatro dígitos. A última parte é uma extensão opcional composta de qualquer quantidade de espaços seguida de ext, x ou ext. seguida de dois a cinco dígitos.

Passo 2: Criar uma regex para endereços de email

Também será necessário ter uma expressão regular que possa corresponder a endereços de email. Faça seu programa ter o seguinte aspecto:

```
#! python3
# phoneAndEmail.py – Encontra números de telefone e endereços de email no clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''(
--trecho removido--

# Cria regex para email.
emailRegex = re.compile(r'''(
    [a-zA-Z0-9._%+-]+      # nome do usuário
    @                      # símbolo @
    [a-zA-Z0-9.-]+         # nome do domínio
    (\.[a-zA-Z]{2,4})      # ponto seguido de outros caracteres
    )''', re.VERBOSE)

# TODO: Encontra correspondências no texto do clipboard.
```

```
# TODO: Copia os resultados para o clipboard.
```

A parte referente ao nome do usuário no endereço de email u tem um ou mais caracteres que podem ser: letras maiúsculas e minúsculas, números, um ponto, um underscore, um sinal de porcentagem, um sinal de adição ou um hífen. Tudo isso pode ser colocado em uma classe de caracteres: [a-zA-Z0-9._%+-].

O domínio e o nome do usuário são separados por um símbolo de @ v. O nome de domínio w tem uma classe de caracteres um pouco menos permissiva, contendo apenas letras, números, pontos e hifens: [a-zA-Z0-9.-]. Por fim, temos a parte "ponto com" (tecnicamente conhecida como *domínio de mais alto nível*) que, na verdade, pode ser um ponto seguido de qualquer caractere. Essa parte tem entre dois e quatro caracteres.

O formato dos endereços de email tem muitas regras singulares. Essa expressão regular não corresponderá a todos os endereços possíveis de email, porém corresponderá a quase todos os endereços típicos de email que você encontrar.

Passo 3: Encontrar todas as correspondências no texto do clipboard

Agora que especificamos as expressões regulares para números de telefone e endereços de email, podemos deixar o módulo re do Python fazer o trabalho pesado de localizar todas as correspondências no clipboard. A função pyperclip.paste() obterá um valor de string com o texto que está no clipboard e o método de regex findall() retornará uma lista de tuplas.

Faça seu programa ter o seguinte aspecto:

```
#! python3
# phoneAndEmail.py – Encontra números de telefone e endereços de email no clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''(
--trecho removido--
```

```
   # Encontra as correspondências no texto do clipboard.
   text = str(pyperclip.paste())

u  matches = []
v  for groups in phoneRegex.findall(text):
       phoneNum = '-'.join([groups[1], groups[3], groups[5]])
       if groups[8] != '':
           phoneNum += ' x' + groups[8]
       matches.append(phoneNum)
w  for groups in emailRegex.findall(text):
       matches.append(groups[0])
   # TODO: Copia os resultados para o clipboard.
```

Há uma tupla para cada correspondência e cada tupla contém strings para cada grupo da expressão regular. Lembre-se de que o grupo `0` corresponde à expressão regular completa, portanto o grupo no índice `0` da tupla é aquele em que estaremos interessados.

Como podemos ver em u, as correspondências serão armazenadas em uma variável de lista chamada `matches`. O programa começa com uma lista vazia e dois loops `for`. Para os endereços de email, devemos concatenar o grupo `0` de cada correspondência w. Para os números de telefone correspondentes, não queremos concatenar somente o grupo `0`. Embora o programa *detecte* números de telefone em diversos formatos, queremos que esse número seja concatenado em um formato único e padrão. A variável `phoneNum` contém uma string criada a partir dos grupos `1`, `3`, `5` e `8` do texto correspondente v. (Esses grupos são: o código de área, os três primeiros dígitos, os quatro últimos dígitos e a extensão.)

Passo 4: Reunir as correspondências em uma string para o clipboard

Agora que temos os endereços de email e os números de telefone na forma de uma lista de strings em `matches`, devemos inseri-los no clipboard. A função `pyperclip.copy()` aceita apenas um único valor de string, e não uma lista de strings; sendo assim, você deve chamar o método `join()` em `matches`.

Para fazer com que seja mais fácil ver que o programa está funcionando, vamos exibir qualquer correspondência encontrada no terminal. Se nenhum

número de telefone ou endereço de email for encontrado, o programa deverá informar isso ao usuário.

Faça seu programa ter o seguinte aspecto:

```
#! python3
# phoneAndEmail.py – Encontra números de telefone e endereços de email no clipboard.

--trecho removido--
for groups in emailRegex.findall(text):
    matches.append(groups[0])

# Copia os resultados para o clipboard.
if len(matches) > 0:
    pyperclip.copy('\n'.join(matches))
    print('Copied to clipboard:')
    print('\n'.join(matches))
else:
    print('No phone numbers or email addresses found.')
```

Executando o programa

Para ver um exemplo, abra seu navegador web na página de contato da No Starch Press em *http://www.nostarch.com/contactus.htm*, tecle CTRL-A para selecionar todo o texto da página e CTRL-C para copiá-lo para o clipboard. Ao executar esse programa, a saída será semelhante a:

```
Copied to clipboard:
800-420-7240
415-863-9900
415-863-9950
info@nostarch.com
media@nostarch.com
academic@nostarch.com
help@nostarch.com
```

Ideias para programas semelhantes

Identificar padrões de texto (e possivelmente substituí-los com o método sub()) tem várias aplicações diferentes em potencial.

• Encontrar URLs de sites que comecem com *http://* ou com *https://*.

- Limpar datas em formatos diferentes (como 3/14/2015, 03-14-2015 e 2015/3/14) substituindo-as por datas em um único formato-padrão.
- Remover informações críticas como números de seguridade social (Social Security Number) ou números de cartões de crédito.
- Encontrar erros comuns de digitação como vários espaços entre palavras, repetição acidental acidental de palavras ou vários pontos de exclamação no final das sentenças. São irritantes!!

Resumo

Embora um computador possa procurar um texto rapidamente, devemos dizer-lhe exatamente o que deverá ser procurado. As expressões regulares permitem especificar padrões exatos de caracteres que estivermos procurando. Com efeito, alguns processadores de texto e aplicativos de planilhas oferecem funcionalidades para pesquisar e substituir que permitem fazer pesquisas usando expressões regulares.

O módulo re que acompanha o Python permite compilar objetos Regex. Esses valores têm diversos métodos: search() para encontrar uma única correspondência, findall() para encontrar todas as instâncias correspondentes e sub() para pesquisar e substituir texto.

Há mais sobre a sintaxe de expressões regulares do que foi descrito neste capítulo. Você poderá descobrir mais informações na documentação oficial do Python em *http://docs.python.org/3/library/re.html*. O site de tutoriais em *http://www.regular-expressions.info/* também é um recurso útil.

Agora que você tem expertise para manipular e fazer correspondência de strings, é hora de mergulhar de cabeça na leitura e na escrita de arquivos no disco rígido de seu computador.

Exercícios práticos

1. Qual é a função que cria objetos Regex?
2. Por que as strings puras (raw strings) geralmente são usadas na criação de objetos Regex?

3. O que o método search() retorna?
4. Como podemos obter as strings correspondentes ao padrão a partir de um objeto Match?
5. Na regex criada a partir de r'(\d\d\d)-(\d\d\d-\d\d\d\d)', o que o grupo 0 inclui? E o grupo 1? E o grupo 2?
6. Os parênteses e os pontos têm significados específicos na sintaxe das expressões regulares. Como podemos especificar uma regex que corresponda aos caracteres que representam parênteses e pontos?
7. O método findall() retorna uma lista de strings ou uma lista de tuplas de strings. O que faz com que uma ou outra opção seja retornada?
8. O que o caractere | representa em expressões regulares?
9. Quais são os dois significados do caractere ? em expressões regulares?
10. Qual é a diferença entre os caracteres + e * em expressões regulares?
11. Qual é a diferença entre {3} e {3,5} em expressões regulares?
12. O que as classes abreviadas de caracteres \d, \w e \s representam em expressões regulares?
13. O que as classes abreviadas de caracteres \D, \W e \S representam em expressões regulares?
14. Como podemos fazer uma expressão regular ignorar as diferenças entre letras maiúsculas e minúsculas (ser case-insensitive)?
15. A que o caractere . normalmente corresponde? A que ele corresponderá se re.DOTALL for passado como segundo argumento de re.compile()?
16. Qual é a diferença entre .* e .*??
17. Qual é a sintaxe da classe de caracteres que corresponde a todos os números e a todas as letras minúsculas?
18. Se numRegex = re.compile(r'\d+'), o que numRegex.sub('X', '12 drummers, 11 pipers, five rings, 3 hens') retornará?
19. O que a especificação de re.VERBOSE como segundo argumento de re.compile() permite fazer?
20. Como você poderá escrever uma regex que corresponda a um número com vírgulas a cada três dígitos? Ela deverá corresponder a:
    - '42'

- '1,234'
- '6,368,745'

mas não a:

- '12,34,567' (que tem somente dois dígitos entre as vírgulas)
- '1234' (que não tem vírgulas)

21. Como você poderá escrever uma regex que corresponda ao nome completo de alguém cujo sobrenome seja Nakamoto? Suponha que o primeiro nome que vem antes dele sempre seja uma única palavra que comece com uma letra maiúscula. A regex deverá corresponder a:
    - 'Satoshi Nakamoto'
    - 'Alice Nakamoto'
    - 'RoboCop Nakamoto'

    mas não a:

    - 'satoshi Nakamoto' (em que o primeiro nome não começa com letra maiúscula)
    - 'Mr. Nakamoto' (em que a palavra anterior tem um caractere que não é uma letra)
    - 'Nakamoto' (que não tem primeiro nome)
    - 'Satoshi nakamoto' (em que Nakamoto não começa com letra maiúscula)
22. Como você poderá escrever uma regex que corresponda a uma frase em que a primeira palavra seja *Alice*, *Bob* ou *Carol*, a segunda palavra seja *eats*, *pets* ou *throws*, a terceira palavra seja *apples*, *cats* ou *baseballs* e a frase termine com um ponto? Essa regex não deve diferenciar letras maiúsculas de minúsculas. Ela deverá corresponder a:
    - 'Alice eats apples.'
    - 'Bob pets cats.'
    - 'Carol throws baseballs.'
    - 'Alice throws Apples.'
    - 'BOB EATS CATS.'

    mas não a:

    - 'RoboCop eats apples.'

- 'ALICE THROWS FOOTBALLS.'
- 'Carol eats 7 cats.'

Projetos práticos

Para exercitar, escreva programas que executem as tarefas a seguir.

Detecção de senhas robustas

Crie uma função que utilize expressões regulares para garantir que a string de senha recebida seja robusta. Uma senha robusta deve ter pelo menos oito caracteres, deve conter tanto letras maiúsculas quanto letras minúsculas e ter pelo menos um dígito. Talvez seja necessário testar a string em relação a diversos padrões de regex para validar sua robustez.

Versão de strip() usando regex

Crie uma função que receba uma string e faça o mesmo que o método de string strip(). Se nenhum outro argumento for passado além da string em que a remoção será feita, os caracteres de espaço em branco serão removidos do início e do fim da string. Caso contrário, os caracteres especificados no segundo argumento da função serão removidos da string.

---

1 N.T.: Tradução literal de acordo com a citação original em inglês: “Knowing [regular expressions] can mean the difference between solving a problem in 3 steps and solving it in 3,000 steps. When you’re a nerd, you forget that the problems you solve with a couple keystrokes can take other people days of tedious, error-prone work to slog through.”

2 Cory Doctorow, “Here’s what ICT should really teach kids: how to do regular expressions” (Eis o que o ICT realmente deveria ensinar às crianças: como criar expressões regulares), *Guardian*, 4 de dezembro de 2012, *http://www.theguardian.com/technology/2012/dec/04/ict-teach-kids-regular-expressions/*.