Capítulo 9: Estudo de caso: jogos de palavras

Este capítulo apresenta o segundo estudo de caso que envolve solucionar quebra-cabeças usando palavras com certas propriedades. Por exemplo, encontraremos os palíndromos mais longos em inglês e procuraremos palavras cujas letras apareçam em ordem alfabética. E apresentarei outro plano de desenvolvimento de programa: a redução a um problema resolvido anteriormente.

9.1 - Leitura de listas de palavras

Para os exercícios deste capítulo vamos usar uma lista de palavras em inglês. Há muitas listas de palavras disponíveis na internet, mas a mais conveniente ao nosso propósito é uma das listas de palavras disponibilizadas em domínio público por Grady Ward como parte do projeto lexical Moby (ver http://wikipedia.org/wiki/Moby_Project). É uma lista de 113.809 palavras cruzadas oficiais; isto é, as palavras que se consideram válidas em quebra-cabeças de palavras cruzadas e outros jogos de palavras. Na coleção Moby, o nome do arquivo é 113809of.fic; você pode baixar uma cópia, com um nome mais simples como words.txt, de http://thinkpython2.com/code/words.txt.

Este arquivo está em texto simples, então você pode abri-lo com um editor de texto, mas também pode lê-lo no Python. A função integrada open recebe o nome do arquivo como um parâmetro e retorna um objeto de arquivo que você pode usar para ler o arquivo.

```
>>> fin = open('words.txt')
```

`fin` é um nome comum de objeto de arquivo usado para entrada de dados. O objeto de arquivo oferece vários métodos de leitura, inclusive readline, que lê caracteres no arquivo até chegar a um comando de nova linha, devolvendo o resultado como uma string:

```
>>> fin.readline()
'aa\r\n'
```

A primeira palavra nesta lista específica é “aa”, uma espécie de lava. A sequência `'\r\n'` representa dois caracteres que representam espaços em branco (whitespace), um retorno de

carro e uma nova linha, que separa esta palavra da seguinte.

O objeto de arquivo grava a posição em que está no arquivo, então se você chamar readline mais uma vez, receberá a seguinte palavra:

```
>>> fin.readline()
'aah\r\n'
```

A palavra seguinte é "aah", uma palavra perfeitamente legítima, então pare de olhar para mim desse jeito. Ou, se é o whitespace que está incomodando você, podemos nos livrar dele com o método de string `strip`:

```
>>> line = fin.readline()
>>> word = line.strip()
>>> word
'aahed'
```

Você também pode usar um objeto de arquivo como parte de um loop `for`. Este programa lê words.txt e imprime cada palavra, uma por linha:

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

9.2 - Exercícios

Há soluções para estes exercícios na próxima seção. Mas é bom você tentar fazer cada um antes de ver as soluções.

Exercício 9.1

Escreva um programa que leia words.txt e imprima apenas as palavras com mais de 20 caracteres (sem contar whitespace).

Exercício 9.2

Em 1939, Ernest Vincent Wright publicou uma novela de 50.000 palavras, chamada Gadsby, que não contém a letra "e". Como o "e" é a letra mais comum em inglês, isso não é algo fácil de fazer.

Na verdade, é difícil até construir um único pensamento sem usar o símbolo mais comum do idioma. No início é lento, mas com prudência e horas de treino, vai ficando cada vez mais fácil.

```
Muito bem, agora eu vou parar.
```

Escreva uma função chamada `has_no_e` que retorne `True` se a palavra dada não tiver a letra “e” nela.

Altere seu programa na seção anterior para imprimir apenas as palavras que não têm “e” e calcule a porcentagem de palavras na lista que não têm “e”.

Exercício 9.3

Escreva uma função chamada avoids que receba uma palavra e uma série de letras proibidas, e retorne True se a palavra não usar nenhuma das letras proibidas.

Altere o código para que o usuário digite uma série de letras proibidas e o programa imprima o número de palavras que não contêm nenhuma delas. Você pode encontrar uma combinação de cinco letras proibidas que exclua o menor número possível de palavras?

Exercício 9.4

Escreva uma função chamada `uses_only` que receba uma palavra e uma série de letras e retorne `True`, se a palavra só contiver letras da lista. Você pode fazer uma frase usando só as letras acefhlo? Que não seja “Hoe alfalfa?”

Exercício 9.5

Escreva uma função chamada `uses_all` que receba uma palavra e uma série de letras obrigatórias e retorne `True` se a palavra usar todas as letras obrigatórias pelo menos uma vez. Quantas palavras usam todas as vogais (aeiou)? E que tal aeiouy?

Exercício 9.6

Escreva uma função chamada `is_abecedarian` que retorne `True` se as letras numa palavra aparecerem em ordem alfabética (tudo bem se houver letras duplas). Quantas palavras em ordem alfabética existem?

9.3 - Busca

Todos os exercícios na seção anterior têm algo em comum; eles podem ser resolvidos com o modelo de busca que vimos em Buscando. O exemplo mais simples é:

```
def has_no_e(word):
    for letter in word:
```

```
        if letter == 'e':
            return False
    return True
```

O loop for atravessa os caracteres em word. Se encontrarmos a letra “e”, podemos retornar False imediatamente; se não for o caso, temos que ir à letra seguinte. Se sairmos do loop normalmente, isso quer dizer que não encontramos um “e”, então retornamos True.

Você pode escrever esta função de forma mais concisa usando o operador in, mas comecei com esta versão porque ela demonstra a lógica do modelo de busca.

`avoids` é uma versão mais geral de `has_no_e` , mas tem a mesma estrutura:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

Podemos retornar `False` logo que encontrarmos uma letra proibida; se chegarmos ao fim do loop, retornamos `True` .

`uses_only` é semelhante, exceto pelo sentido da condição, que se inverte:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

Em vez de uma lista de letras proibidas, temos uma lista de letras disponíveis. Se encontrarmos uma letra em word que não está em `available` , podemos retornar `False` .

`uses_all` é semelhante, mas invertemos a função da palavra e a string de letras:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

Em vez de atravessar as letras em `word` , o loop atravessa as letras obrigatórias. Se alguma das letras obrigatórias não aparecer na palavra, podemos retornar `False` .

Se você realmente estivesse pensando como um cientista da computação, teria reconhecido que `uses_all` foi um exemplo de um problema resolvido anteriormente e escreveria:

```
def uses_all(word, required):
    return uses_only(required, word)
```

Esse é um exemplo de um plano de desenvolvimento de programa chamado **redução a um problema resolvido anteriormente**, ou seja, você reconhece o problema no qual está trabalhando como um exemplo de um problema já resolvido e aplica uma solução existente.

9.4 - Loop com índices

Escrevi as funções na seção anterior com loops `for` porque eu só precisava dos caracteres nas strings; não precisava fazer nada com os índices.

Para `is_abecedarian` temos que comparar letras adjacentes, o que é um pouco complicado para o loop for:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

Uma alternativa é usar a recursividade:

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

Outra opção é usar um loop `while` :

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

O loop começa com `i == 0` e termina quando `i == len(word)-1`. Cada vez que passa pelo loop, o programa compara o "i-ésimo" caractere (que você pode considerar o caractere atual) com o caractere de posição `i+1` (que pode ser considerado o caractere seguinte).

Se o próximo caractere for de uma posição anterior (alfabeticamente anterior) à atual, então descobrimos uma quebra na tendência alfabética, e retornamos `False`.

Se chegarmos ao fim do loop sem encontrar uma quebra, então a palavra passa no teste. Para convencer-se de que o loop termina corretamente, considere um exemplo como `'flossy'`. O comprimento da palavra é 6, então o loop é executado pela última vez quando i for igual a 4, que é o índice do segundo caractere de trás para frente. Na última iteração, o programa compara o penúltimo caractere com o último, que é o que queremos.

Aqui está uma versão de `is_palindrome` (veja o Exercício 6.3) que usa dois índices: um começa no início e aumenta; o outro começa no final e diminui.

```
def is_palindrome(word):
    i = 0
    j = len(word)-1
    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1
    return True
```

Ou podemos reduzir a um problema resolvido anteriormente e escrever:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

Usando `is_reverse` da seção 8.11.

9.5 - Depuração

Testar programas é difícil. As funções neste capítulo são relativamente fáceis para testar porque é possível verificar os resultados à mão. Ainda assim, pode ser difícil ou até impossível escolher um grupo de palavras que teste todos os erros possíveis.

Tomando `has_no_e` como exemplo, há dois casos óbvios para verificar: as palavras que têm um 'e' devem retornar False, e as palavras que não têm devem retornar True. Não deverá ser um problema pensar em um exemplo de cada uma.

Dentro de cada caso, há alguns subcasos menos óbvios. Entre as palavras que têm um “e”, você deve testar palavras com um “e” no começo, no fim e em algum lugar no meio. Você deve testar palavras longas, palavras curtas e palavras muito curtas, como a string vazia. A string vazia é um exemplo de um caso especial, não óbvio, onde erros muitas vezes espreitam.

Além dos casos de teste que você gerar, também pode ser uma boa ideia testar seu programa com uma lista de palavras como words.txt. Ao analisar a saída, pode ser que os erros apareçam, mas tenha cuidado: você pode pegar um tipo de erro (palavras que não deveriam ser incluídas, mas foram) e não outro (palavras que deveriam ser incluídas, mas não foram).

Em geral, o teste pode ajudar a encontrar bugs, mas não é fácil gerar um bom conjunto de casos de teste, e, mesmo se conseguir, não há como ter certeza de que o programa está correto. Segundo um lendário cientista da computação:

Testar programas pode ser usado para mostrar a presença de bugs, mas nunca para mostrar a ausência deles! – Edsger W. Dijkstra

9.6 - Glossário

objeto de arquivo
: Um valor que representa um arquivo aberto.

redução a um problema resolvido anteriormente
: Um modo de resolver um problema expressando-o como uma instância de um problema resolvido anteriormente.

caso especial
: Um caso de teste que é atípico ou não é óbvio (e com probabilidade menor de ser tratado corretamente).

9.7 - Exercícios

Exercício 9.7

Esta pergunta é baseada em um quebra-cabeça veiculado em um programa de rádio chamado Car Talk (http://www.cartalk.com/content/puzzlers):

Dê uma palavra com três letras duplas consecutivas. Vou dar exemplos de palavras que quase cumprem a condição, mas não chegam lá. Por exemplo, a palavra committee, c-o-m-m-i-t-t-e-e. Seria perfeita se não fosse aquele ‘i’ que se meteu ali no meio. Ou Mississippi: M-i-s-s-i-s-s-i-p-p-i. Se pudesse tirar aqueles ‘is’, daria certo. Mas há uma palavra que tem três pares consecutivos de letras e, que eu saiba, pode ser a única palavra que existe. É claro que provavelmente haja mais umas 500, mas só consigo pensar nessa. Qual é a palavra?

Escreva um programa que a encontre.

Solução: http://thinkpython2.com/code/cartalk1.py.

Exercício 9.8

Aqui está outro quebra-cabeça do programa Car Talk (http://www.cartalk.com/content/puzzlers):

"Estava dirigindo outro dia e percebi algo no hodômetro que chamou a minha atenção. Como a maior parte dos hodômetros, ele mostra seis dígitos, apenas em milhas inteiras. Por exemplo, se o meu carro tivesse 300.000 milhas, eu veria 3-0-0-0-0-0.

"Agora, o que vi naquele dia foi muito interessante. Notei que os últimos 4 dígitos eram um palíndromo; isto é, podiam ser lidos da mesma forma no sentido correto e no sentido inverso. Por exemplo, 5-4-4-5 é um palíndromo, então no meu hodômetro poderia ser 3-1-5-4-4-5.

"Uma milha depois, os últimos 5 números formaram um palíndromo. Por exemplo, poderia ser 3-6-5-4-5-6. Uma milha depois disso, os 4 números do meio, dentro dos 6, formavam um palíndromo. E adivinhe só? Um milha depois, todos os 6 formavam um palíndromo!

"A pergunta é: o que estava no hodômetro quando olhei primeiro?"

Escreva um programa Python que teste todos os números de seis dígitos e imprima qualquer número que satisfaça essas condições.

Solução: http://thinkpython2.com/code/cartalk2.py.

Exercício 9.9

Aqui está outro problema do Car Talk que você pode resolver com uma busca (http://www.cartalk.com/content/puzzlers):

"Há pouco tempo recebi uma visita da minha mãe e percebemos que os dois dígitos que compõem a minha idade, quando invertidos, representavam a idade dela. Por exemplo, se ela tem 73 anos, eu tenho 37 anos. Ficamos imaginando com que frequência isto aconteceu nos anos anteriores, mas acabamos mudando de assunto e não chegamos a uma resposta.

"Quando cheguei em casa, cheguei à conclusão de que os dígitos das nossas idades tinham sido reversíveis seis vezes até então. Também percebi que, se tivéssemos sorte, isso aconteceria novamente dali a alguns anos, e se fôssemos muito sortudos, aconteceria mais uma vez depois disso. Em outras palavras, aconteceria 8 vezes no total. Então a pergunta é: quantos anos tenho agora?"

Escreva um programa em Python que busque soluções para esse problema. Dica: pode ser uma boa ideia usar o método de string zfill.