# Capítulo 9

# Dicionários

Um *dicionário* é como uma lista, porém mais geral. Em uma lista, o índice de posição deve ser um inteiro; em um dicionário, o índice pode ser de (quase) qualquer tipo.

Você pode pensar em um dicionário como o mapeamento entre um conjunto de índices (chamados de *chaves*) e um conjunto de valores. Cada chave localiza um valor. A associação entre uma chave e um valor é chamado de *par chave-valor* ou, às vezes, de *item*.

Como exemplo, vamos construir um dicionário que mapeia palavras do Inglês para o Português, então os valores e chaves são todos strings.

A função `dict` cria um novo dicionário com nenhum item. Como `dict` é o nome de uma função interna da linguagem, deve-se evitar usá-lo como nome de variável.

```
>>> ing2port = dict()
>>> print(ing2port)
{}
```

As chaves, `{}`, representam um dicionário vazio. Para adicionar itens ao dicionário, colchetes, `[]`, podem ser utilizados:

```
>>>  ing2port['one'] = 'um'
```

Esta linha cria um item que localiza, a partir da chave `'one'`, o valor "um". Se pedirmos para o conteúdo do dicionário ser mostrado novamente, veremos o par chave-valor separados por dois pontos:

```
>>> print(ing2port)
{'one': 'um'}
```

Este formato de saída também é um formato de entrada. Por exemplo, um dicionário pode ser criado com três itens. Contudo, se `ing2port` for mostrado, você pode se surpreender:

```
>>> ing2port = {'one': 'um', 'two': 'dois', 'three': 'três'}
>>> print(ing2port)
{'one': 'um', 'three': 'três', 'two': 'dois'}
```

A ordem dos pares chave-valor não é a mesma. Na verdade, se o mesmo exemplo for executado em seu computador, o resultado pode ser diferente. Em geral, a ordem dos itens em um dicionário é imprevisível.

Contudo, isso não é um problema já que os elementos em um dicionário nunca são indexados usando inteiros como índice. Ao invés disso, chaves são utilizadas para consultar os valores correspondentes:

```
>>> print(ing2port['two'])
'dois'
```

A chave `'two'` sempre localizará o valor "dois" e, portanto, a ordem dos itens não importa.

Se a chave não estiver contida no dicionário, ocorrerá um erro:

```
>>> print(ing2port['four'])
KeyError: 'four'
```

A função `len` funciona em dicionários; ele retorna o número de pares chave-valor:

```
>>> len(ing2port)
3
```

O operador `in` também funciona em dicionários; ele dirá se algo aparece no dicionário como uma chave (os valores dos pares não são considerados).

```
>>> 'one' in ing2port
True
>>> 'um' in ing2port
False
```

Para ver se algo aparece como valor em um dicionário, o método `values` pode ser utilizado. Esse método retornará os valores em uma lista, e então, usando o operador `in`:

```
>>> vals = list(ing2port.values())
>>> 'um' in vals
True
```

O operador `in` faz uso de um algoritmo diferente para listas e dicionários. Em listas, ele executa um algoritmo de busca linear. A medida que a lista vai ficando mais longa, o tempo de busca aumenta proporcionalmente ao comprimento da lista. Em dicionários, Python faz uso de um algoritmo chamado *tabela de dispersão* (*hash table*), que possui uma notória propriedade: o operador `in` leva, aproximadamente,

o mesmo tempo de execução, não importando a quantidade de itens no dicionário. Não explicarei porque funções hash são tão mágicas, porém você pode ler mais sobre em: wikipedia.org/wiki/Hash_table.

**Exercício 1: Faça o download de uma cópia do arquivo**

www.py4e.com/code3/words.txt

**Escreva um programa que leia as palavras em *words.txt* e as armazena como chaves em um dicionário. Não importa quais são os valores. Então, você pode usar o operador `in` como uma maneira rápida de verificar se uma string está no dicionário.**

## 9.1 Dicionário como um conjunto de contadores

Suponha que lhe deram uma string e que você queira contar quantas vezes cada letra aparece. Existem várias formas de fazer isso:

1. Você pode criar 26 variáveis, uma para cada letra do alfabeto. Então, você pode examinar a string e, para cada caractere, incrementar o contador correspondente, provavelmente usando uma condicional encadeada.

2. Você pode criar uma lista com 26 elementos. Então, você pode converter cada caractere para um número (usando a função embutida `ord`), usar o número como um índice na lista, e incrementar o contador apropriado.

3. Você pode criar um dicionário, estabelecendo caracteres como chaves e contadores como os valores correspondentes. A primeira vez em que um caractere for lido, um novo item seria adicionado ao dicionário. Após isso, o valor de um item existente seria incrementado.

Cada uma destas opções executa a mesma computação, porém cada uma delas implementa essa computação de uma forma diferente.

Uma *implementação* é uma forma de executar uma computação; algumas implementações são melhores que outras. Por exemplo, uma vantagem da implementação utilizando o dicionário é que não precisamos saber de antemão quais as letras presentes na string e precisamos apenas separar espaço para letras que realmente aparecem.

Aqui está uma ideia de como este código seria:

```
palavra = 'brontosaurus'
d = dict()
for c in palavra:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print(d)
```

Estamos, efetivamente, computando um histograma, que é um termo estatístico para um conjunto de contadores (ou frequências).

A repetição `for` examina a string. A cada repetição, se o caractere `c` não estiver no dicionário, criamos um novo item com chave `c` e valor inicial 1 (já que só vimos essa letra uma vez). Se c já estiver contido no dicionário, incrementamos `d[c]`.

Esta é a saída do programa:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

O histograma indica que as letras "a" e "b" aparecem uma vez; "o" aparece duas vezes, e assim por diante.

Dicionários possuem um método chamado `get` que recebe uma chave e um valor padrão. Se a chave estiver contida no dicionário, `get` retorna o valor correspondente; caso contrário, retorna o valor padrão. Por exemplo:

```
>>> conta = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print(conta.get('jan', 0))
100
>>> print(conta.get('tim', 0))
0
```

Podemos usar `get` para escrever o laço do histograma de forma mais concisa. Já que o método `get` automaticamente trata o caso em que a chave não está contida no dicionário, podemos reduzir quatro linhas a apenas uma e descartar a instrução `if`.

```
word = 'brontosaurus'
d = dict()
for c in palavra:
    d[c] = d.get(c,0) + 1
print(d)
```

O uso do método `get` para simplificar essa repetição é um "jargão" muito comum em Python e nós o usaremos muitas vezes pelo resto deste livro. Portanto, você deveria tomar um tempo para comparar o laço utilizando a instrução `if` e o operador `in` com o laço utilizando o método `get`. Ambos fazem a mesma coisa, porém um é mais sucinto.

## 9.2 Dicionários e Arquivos

Um dos usos mais comuns de um dicionário é a contagem da ocorrência de palavras em um texto presente em algum arquivo. Comecemos com um arquivo de palavras, muito simples, extraído do texto de *Romeu e Julieta*.

Para o primeiro conjunto de exemplos, usaremos uma versão reduzida e simplificada do texto, sem pontuação. Mais a frente, trabalharemos com o texto original, com a pontuação.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Escreveremos um programa em Python para ler as linhas do arquivo, quebrá-las em uma lista de palavras e, então, contar a ocorrência de cada palavra presente no texto, usando um dicionário.

Veremos que temos dois laços `for`. A repetição externa está lendo as linhas do arquivo e a repetição interna está iterando entre cada palavra naquela linha em particular. Esse é um exemplo de padrão chamado *laços aninhados*, pois um laço está contido dentro de outro.

Como a repetição interna executa todas as suas iterações enquanto a externa itera uma única vez, pensamos na interna iterando "mais rapidamente" enquanto a externa itera "mais lentamente".

A combinação das duas repetições aninhadas assegura que vamos contar cada palavra em cada linha do arquivo de entrada.

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Code: http://www.py4e.com/code3/count1.py
```

Em nossa instrução `else`, usamos a alternativa mais compacta para incrementar uma variável. `conta[palavra] += 1` é equivalente a `conta[palavra] = conta[palavra]+1`. Tanto um método quanto o outro pode ser utilizado para alterar o valor da variável em qualquer que seja a quantidade desejada. Alternativas similares existem para `-=`, `*=`, e `/=`.

Quando executamos o programa, vemos todas as contagens dispostas em ordem aleatória. (o arquivo *romeo.txt* está disponível em www.py4e.com/code3/romeo.txt)

```
python count1.py
Enter the file name: romeo.txt
```

```
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

É um pouco inconveniente ter que analisar o dicionário para encontrar as palavras mais comuns e a sua contagem, então precisamos adicionar um pouco mais de código Python para termos uma saída mais clara.

## 9.3 Laços e Dicionários

Se usarmos um dicionário como a sequência em uma instrução `for`, ela percorrerá as chaves do dicionário. Este laço mostra cada chave e seu valor correspondente:

```
conta = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for chave in conta:
    print(chave, conta[key])
```

Aqui está como a saída se parece:

```
jan 100
chuck 1
annie 42
```

Novamente, as chaves não estão em uma ordem particular.

Podemos usar este padrão para implementar vários "jargões" de repetição que descrevemos anteriormente. Por exemplo, se quisermos encontrar todas as entradas em um dicionário com valor maior que dez, poderíamos escrever o seguinte código:

```
conta = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for chave in conta:
    if conta[chave] > 10 :
        print(chave, conta[chave])
```

O laço `for` itera entre as *chaves* do dicionário, então devemos utilizar o operador de índice para retornar o *valor* correspondente a cada chave. A saída se pareceria com:

```
jan 100
annie 42
```

Vemos apenas as entradas com valor acima de 10.

Se quisermos mostrar as chaves em ordem alfabética, primeiramente precisamos fazer uma lista, contendo as chaves desse dicionário, usando o método `keys`, disponível no próprio objeto dicionário. A partir disso, podemos ordenar a lista e percorrê-la, analisando cada chave e mostrando os pares chave-valor de forma ordenada, como a seguir:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = list(counts.keys())
print(lst)
lst.sort()
for key in lst:
    print(key, counts[key])
```

A saída se parece com:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

Primeiro vemos a lista de chaves desordenada que retiramos pelo método `key`. Em seguida, vemos os pares chave-valor em ordem, vindos do laço `for`.

## 9.4 Métodos avançados de divisão de texto

No exemplo passado utilizando o arquivo *romeo.txt*, deixamos o texto o mais simples possível removendo todas as pontuações manualmente. O texto original tem muitas pontuações, como mostrado abaixo.

```
Mas, sutil! que luz ultrapassa aquela janela?
Esse é o leste, e Julieta é o sol.
Ascenda, honesto sol, e acabe com a invejosa lua,
Que já está doente e pálida com o luto,
```

Desde que a função `split` procura por espaços e trata as palavras como objetos separados por espaços, nós trataremos as palavras "sutilmente!" e "sutil" como palavras *diferentes* e criaremos entradas específicas no dicionário para cada palavra.

Além disso, já que o arquivo tem letras maiúsculas, nós trataremos "quem" e "Quem" como palavras diferentes, com diferentes contadores.

Podemos resolver os dois problemas utilizando os métodos de strings `lower`, `punctuation`, e
`translate`. O `translate` é o mais sutil dos métodos. Aqui está a documentação para o `translate`:

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

*Troca os caracteres em `fromstr` com o caractere na mesma posição em `tostr` e apaga todos os caracteres que estão em `deletestr`. `fromstr` e `tostr` podem ser strings vazias e o parâmetro `deletestr` pode ser omitido.*

Nós não iremos especificar o `tostr` mas utilizaremos o parâmetro `deletestr` para retirar todas as pontuações. Nós até iremos deixar o Python nos mostrar a lista de caracteres que são considerados "pontuação":

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

O parâmetro utilizado pelo `translate` era diferente no Python 2.0.

As seguintes modificações foram feitas no nosso programa:

```
import string

fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    line = line.rstrip()
    line = line.translate(line.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Code: http://www.py4e.com/code3/count2.py
```

Parte do aprendizado em "Art of Python" ou "Thinking Pythonically" é perceber que o Python geralmente tem soluções prontas para os problemas mais comuns de análise de dados. Com o tempo, você verá vários exemplos de códigos e lerá documentações de funções suficientes para saber onde procurar se alguém já escreveu algo que tornará seu trabalho mais fácil.

A seguir está uma versão abreviada da saída:

```
Entre com o nome do arquivo: romeo-full.txt
{'juro': 1, 'todos': 6, 'afeard': 1, 'deixar': 2, 'esse': 2,
'parentes': 2, 'o que': 11, 'pensador': 1, 'amor': 24, 'capa': 1,
a': 24, 'pomar': 2, 'luz': 5, 'amantes': 2, 'romeo': 40,
'donzela': 1, 'whiteupturned': 1, 'juliet': 32, 'cavalheiro': 1,
'isso': 22, 'inclina': 1, 'canst': 1, 'tendo': 1, ...}
```

Olhar para essa saída ainda é complicado e podemos usar o Python para nos dar exatamente o que estamos procurando, mas, para isso, precisamos aprender *tuplas*. Nós voltaremos para esse exemplo quando aprendermos tuplas.

## 9.5 Debugging

Como você trabalhou com maiores conjuntos de dados, torna-se difícil tratar erros mostrando tudo em tela e verificando manualmente. Aqui estão algumas sugestões para o *debugging* de grandes conjuntos de dados:

**Diminua a escala da entrada** Se possível, diminua o tamanho do conjunto de dados. Por exemplo, se o programa lê um arquivo de texto, comece com apenas 10 linhas ou com o menor exemplo que encontrar. Você ainda pode editar os arquivos ou (melhor) modificar o programa para que ele leia apenas as primeiras 'n' linhas.

Se existir um erro, reduza `n` para o menor valor que manifeste o erro, só então o aumente gradualmente enquanto encontra e corrige os erros.

**Verifique resumos e tipos** Ao invés de mostrar e verificar todo o conjunto de dados, considere apresentar resumos do conjunto: por exemplo o número de ítens num dicionário ou o total de uma lista de números.

Uma causa comum de erros em tempo de execução é a presença de um valor que não é do tipo esperado. Para tratar esse tipo de erro, normalmente basta mostrar em tela o tipo da entrada.

**Escreva auto-verificadores** Normalmente pode-se escrever códigos que verifiquem erros automaticamente. Por exemplo, se você está calculando a média de um conjunto de números, você pode verificar se o resultado não está acima do maior valor do conjunto, ou abaixo do menor. Isso é chamado de "verificação de sanidade", pois detecta resultados que são "completamente sem lógica".

Outro tipo de verificação compara o resultado de duas computações diferentes para analisar a consitência dos resultados. Isso é chamado de "verificação de consitência".

**Organize a maneira que a saída será mostrada** Analisar uma saída formatada é mais fácil de encontrar erros.

Novamente, o tempo gasto construindo os alicerces diminui o tempo gasto no *debugging*.

## 9.6 Glossario

**chave** Um objeto que aparece num dicionário na primeira parte do par chave-valor.

**dicionário** Um conjunto de uma série de chaves e seus correspondentes valores.

**função hash** Uma função utilizada pela tabela de dispersão (*hashtable*) para cacular a localização da chave.

**histograma** Um conjunto de contadores.

**implementação** A forma como o problema é resolvido (em forma de código, por exemplo).

**item** Outro nome para o par chave-valor.

**key-value pair** The representation of the mapping from a key to a value.

**laços aninhados** Quando há um ou mais laços de repetição "dentro" de outro laço. O laço interno completa todas as iterações para cada iteração do externo.

**lookup** Uma operação do dicionário que toma um chave e encontra seu valor correspondente.

**tabela de Dispersão (*hashtable*)** O algorítmo utilizado para implementar os dicionários em Python.

**valor** Um objeto que aparece num dicionário na segunda parte do par chave-valor. Isso é mais específico do que nossos usos anteriores da palavra "valor".

## 9.7 Exercícios

**Exercício 2: Escreva um programa que categorize cada mensagem de e-mail de acordo com o dia em que a mensagem foi enviada. Para isso, procure por linhas que comecem com "From", depois procure pela terceira palavra e mantenha uma contagem de ocorrência para cada dia da semana. No final do programa, mostre em tela o conteúdo do seu dicionário (a ordem não interessa).**

**linha exemplo:**

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

**Exemplo de código:**

```
python dow.py
Enter a file name: mbox-short.txt
{'Sex': 20, 'Qui': 6, 'Sab': 1}
```

**Exercício 3: Escreva um programa que leia um registro de mensagens, construa um histograma, utilizando um dicionário, para contar quantas mensagens chegaram em cada endereço de email e mostre em tela o dicionário.**

```
Enter a file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

**Exercício 4: Adicione linhas de código no programa abaixo para identificar quem possui mais mensagens no arquivo. Após todo o dado ser lido e todo o dicionário ser criado, procure no dicionário, utilizando um laço máximo (Veja o capítulo 5: Laços máximo e mínimo), quem tem o maior número de mensagens e mostre em tela quantas mensagens essa pessoa tem.**

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

**Exercício 5: Esse programa grava o domínio de email (ao invés do endereço) de onde a mensagem foi enviada ao invés de quem o email veio ( i.e., o endereço completo da mensagem). No final do programa, mostre em tela o conteúdo do seu dicionário.**

```
python schoolcount.py
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```