

CAPÍTULO 9

Arquivos

Precisamos de uma forma de armazenar dados permanentemente. Arquivos são uma excelente forma de entrada e saída de dados para programas. Com eles, poderemos ler dados de outros programas e mesmo da internet.

Mas o que é um arquivo? Um arquivo é uma área em disco onde podemos ler e gravar informações. Essa área é gerenciada pelo sistema operacional do computador, ou seja, não precisamos nos preocupar em como esse espaço é organizado em disco. Do ponto de vista de programas, um arquivo é acessado por nome e é onde podemos ler e escrever linhas de texto ou dados em geral.

Para acessar um arquivo, precisamos abri-lo. Durante a abertura, informamos o nome do arquivo, com o nome do diretório onde ele se encontra (se necessário) e que operações queremos realizar: leitura e/ou escrita. Em Python, abrimos arquivos com a função `open`.

A função `open` utiliza os parâmetros nome e modo. O nome é o nome do arquivo em si, por exemplo, `leiamen.txt`. O modo indica as operações que vamos realizar (Tabela 9.1).

Tabela 9.1 – Modos de abertura de arquivos

Modo	Operações
r	leitura
w	escrita, apaga o conteúdo se já existir
a	escrita, mas preserva o conteúdo se já existir
b	modo binário
+	atualização (leitura e escrita)

Os modos podem ser combinados (“`r+`”, “`w+`”, “`a+`”, “`r+b`”, “`w+b`”, “`a+b`”). As diferenças envolvidas serão discutidas a seguir. A função `open` retorna um objeto do tipo `file` (arquivo). É esse objeto que vamos utilizar para ler e escrever os dados no arquivo. Utilizamos o método `write` para escrever ou gravar dados no arquivo, `read` para ler e `close` para fechá-lo. Ao trabalharmos com arquivos, devemos sempre realizar o seguinte ciclo: abertura, leitura e/ou escrita, fechamento.

A abertura realiza a ligação entre o programa e o espaço em disco, gerenciado pelo sistema operacional. As etapas de leitura e/ou escrita são as operações que desejamos realizar no programa, e o fechamento informa ao sistema operacional que não vamos mais trabalhar com o arquivo.

O fechamento do arquivo é muito importante, pois cada arquivo aberto consome recursos do computador. Só o fechamento do arquivo garante a liberação desses recursos e preserva a integridade dos dados do arquivo.

Vejamos um exemplo onde vamos escrever o arquivo `números.txt` com 100 linhas. Em cada linha, vamos escrever um número (Listagem 9.1).

► Listagem 9.1 – Abrindo, escrevendo e fechando um arquivo

```
arquivo=open("números.txt", "w") ❶
for linha in range(1,101): ❷
    arquivo.write("%d\n" % linha) ❸
arquivo.close() ❹
```

Execute o programa da listagem 9.1. Se tudo correu bem, nada aparecerá na tela. O programa cria um novo arquivo no diretório atual, ou seja, no mesmo diretório em que você gravou seu programa.

Se você usa Windows, abra o Windows Explorer, escolha a pasta onde grava seus programas e procure o arquivo `números.txt`. Você pode abri-lo com o Notepad ou simplesmente clicando duas vezes em seu nome. Uma vez aberto, observe as linhas: seu programa gerou um arquivo que pode ser aberto por outros programas!

Se você utiliza o Mac OS X, ative o Finder para localizar e abrir seu arquivo. No Linux, você pode utilizar o Nautilus ou simplesmente digitar `less números.txt` na linha de comando.

Em ❶, utilizamos a função `open` para abrir o arquivo `números.txt`. Observe que o nome do arquivo é uma string e deve ser escrito entre aspas. O modo escolhido foi “`w`”, indicando escrita ou gravação. O modo “`w`” cria o arquivo se ele não existir. Caso já exista, seu conteúdo é apagado. Para verificar esse efeito, execute

o programa novamente e observe que o arquivo continua com 100 linhas. Experimente também escrever algo no arquivo usando o Notepad (ou outro editor de textos simples), grave e execute o programa novamente. Tudo que você escreveu no editor foi perdido, pois o programa apaga o conteúdo do arquivo e começa tudo de novo.

Criamos um `for` ❷ para gerar os números das linhas. Em ❸, escrevemos o número da linha no arquivo, usando o método `write`. Observe que escrevemos `arquivo.write`, pois `write` é um método do objeto arquivo. Observe que escrevemos "%d\n" % linha, onde o "%d" funciona como na função `print`, mas temos que adicionar o "\n" para indicar que queremos passar para uma nova linha.

A linha ❹ fecha o arquivo. O fechamento garante que o sistema operacional foi informado que não vamos mais trabalhar com o arquivo. Essa comunicação é importante, pois o sistema operacional realiza diversas funções para otimizar a velocidade de suas operações, sendo que uma delas é a de não gravar os dados diretamente no disco, mas em uma memória auxiliar. Quando não fechamos o arquivo, corremos o risco de essa memória auxiliar não ser transferida para o disco e, assim, perdermos o que foi escrito no programa.

Vejamos agora um programa para ler o arquivo e imprimir suas linhas na tela (Listagem 9.2).

► Listagem 9.2 – Abrindo, lendo e fechando um arquivo

```
arquivo=open("números.txt","r") ❶
for linha in arquivo.readlines(): ❷
    print(linha) ❸
arquivo.close() ❹
```

Em ❶, utilizamos a função `open` para abrir o arquivo. O nome é o mesmo utilizado durante a gravação, mas o modo agora é "r", ou seja, leitura. Em ❷ utilizamos o método `readlines`, que gera uma lista em que cada elemento é uma linha do arquivo. Em ❸ simplesmente imprimimos a linha na tela. Em ❹ fechamos o arquivo.

Veja que ❶ é muito parecida com a mesma linha do programa anterior, e que ❹ é idêntica. Isso acontece porque a abertura e o fechamento são partes essenciais na manipulação de arquivos. Sempre que manipularmos arquivos, teremos uma abertura, operações e fechamento.

Até agora estamos trabalhando com arquivos do tipo texto. A característica principal desses arquivos é que seu conteúdo é apenas texto simples, com alguns caracteres especiais de controle. O caractere de controle mais importante em um

arquivo-texto é o que marca o fim de uma linha. No Windows, o fim de linha é marcado por uma sequência de dois caracteres, cujas posições na tabela ASCII são 10 (*LF – Line Feed*, avanço de linha) e 13 (*CR – Carriage Return*, retorno de cursor). No Linux, temos apenas um caractere marcando o fim de linha, o *Line Feed* (10). Já no Mac OS X, temos apenas o *Carriage Return* (13) marcando o fim de uma linha.

9.1 Parâmetros da linha de comando

Podemos acessar os parâmetros passados ao programa na linha de comando utilizando o módulo `sys` e trabalhando com a lista `argv`. Vejamos o programa da listagem 9.3.

► Listagem 9.3 – Impressão dos parâmetros passados na linha de comando

```
import sys
print("Número de parâmetros: %d" % len(sys.argv))
for n,p in enumerate(sys.argv):
    print("Parâmetro %d = %s" % (n,p))
```

Experimente chamar o script na linha de comando usando os seguintes parâmetros:

```
fparam.py primeiro segundo terceiro
fparam.py 1 2 3
fparam.py readme.txt 5
```

Observe que cada parâmetro foi passado com um elemento da lista `sys.argv` e que os parâmetros são separados por espaços em branco na linha de comando, mas que esses espaços são removidos dos elementos em `sys.argv`. Se você precisar passar um parâmetro com espaços em branco, como um nome formado por várias palavras, escreva-o entre aspas para que ele seja considerado como um só parâmetro.

```
fparam.py "Nome grande" "Segundo nome grande"
```

Exercício 9.1 Escreva um programa que receba o nome de um arquivo pela linha de comando e que imprima todas as linhas desse arquivo.

Exercício 9.2 Modifique o programa do exercício 9.1 para que receba mais dois parâmetros: a linha de início e a de fim para impressão. O programa deve imprimir apenas as linhas entre esses dois valores (incluindo as linhas de início e fim).

9.2 Geração de arquivos

Vejamos o programa da listagem 9.4, que gera dois arquivos com 500 linhas cada. O programa distribui os números ímpares e pares em arquivos diferentes.

Observa que para gravar em arquivos diferentes, utilizamos um `if` e dois arquivos abertos para escrita. Utilizando o método `write` dos objetos `pares` e `ímpares`, fizemos a seleção de onde gravar o número. Observe que incluímos o `\n` para indicar fim de linha.

► Listagem 9.4 – Gravação de números pares e ímpares em arquivos diferentes

```
ímpares=open("ímpares.txt","w")
pares=open("pares.txt","w")
for n in range(0,1000):
    if n % 2 == 0:
        pares.write("%d\n" % n)
    else:
        ímpares.write("%d\n" % n)
ímpares.close()
pares.close()
```

9.3 Leitura e escrita

Podemos realizar diversas operações com arquivos; entre elas ler, processar e gerar novos arquivos. Utilizando o arquivo `pares.txt` criado pelo programa da listagem 9.4, vejamos como filtrá-lo de forma a gerar um novo arquivo, apenas com números múltiplos de 4 (Listagem 9.5).

► Listagem 9.5 – Filtragem exclusiva dos múltiplos de quatro

```
múltiplos4=open("múltiplos de 4.txt","w")
pares=open("pares.txt")
```

```
for l in pares.readlines():
    if int(l) % 4 == 0: ❶
        múltiplos4.write(l)
pares.close()
múltiplos4.close()
```

Veja que em ❶ convertemos a linha lida de string para inteiro antes de fazer os cálculos.

Exercício 9.3 Crie um programa que leia os arquivos `pares.txt` e `ímpares.txt` e que crie um só arquivo `pares e ímpares.txt` com todas as linhas dos outros dois arquivos, de forma a preservar a ordem numérica.

Exercício 9.4 Crie um programa que receba o nome de dois arquivos como parâmetros da linha de comando e que gere um arquivo de saída com as linhas do primeiro e do segundo arquivo.

Exercício 9.5 Crie um programa que inverta a ordem das linhas do arquivo `pares.txt`. A primeira linha deve conter o maior número; e a última, o menor.

9.4 Processamento de um arquivo

Podemos também processar as linhas de um arquivo de entrada com se fossem comandos. Vejamos um exemplo na listagem 9.6, na qual as linhas cujo primeiro caractere é igual a ; serão ignoradas; as com > serão impressas alinhadas à direita; as com <, alinhadas à esquerda; e as com * serão centralizadas.

Crie um arquivo com as seguintes linhas e salve-o como `entrada.txt`:

```
;Esta linha não deve ser impressa.
>Esta linha deve ser impressa alinhada a direita
*Esta linha deve ser centralizada
<Uma linha normal
  Outra linha normal
```

► Listagem 9.6 – Processamento de um arquivo

```
LARGURA=79
entrada=open("entrada.txt")
for linha in entrada.readlines():
    if linha[0]==";":
        continue
    elif linha[0]==>:
        print(linha[1:].rjust(LARGURA))
    elif linha[0]==*:
        print(linha[1:].center(LARGURA))
    else:
        print(linha)
entrada.close()
```

Exercício 9.6 Modifique o programa da listagem 9.6 para imprimir 40 vezes o símbolo de = se este for o primeiro caractere da linha. Adicione também a opção para parar de imprimir até que se pressione a tecla Enter cada vez que uma linha iniciar com . como primeiro caractere.

Exercício 9.7 Crie um programa que leia um arquivo-texto e gere um arquivo de saída paginado. Cada linha não deve conter mais de 76 caracteres. Cada página terá no máximo 60 linhas. Adicione na última linha de cada página o número da página atual e o nome do arquivo original.

Exercício 9.8 Modifique o programa anterior para também receber o número de caracteres por linha e o número de páginas por folha pela linha de comando.

Exercício 9.9 Crie um programa que receba uma lista de nomes de arquivo e os imprima, um por um.

Exercício 9.10 Crie um programa que receba uma lista de nomes de arquivo e que gere apenas um grande arquivo de saída.

Exercício 9.11 Crie um programa que leia um arquivo e crie um dicionário onde cada chave é uma palavra e cada valor é o número de ocorrências no arquivo.

Exercício 9.12 Modifique o programa anterior para também registrar a linha e a coluna de cada ocorrência da palavra no arquivo. Para isso, utilize listas nos valores de cada palavra, guardando a linha e a coluna de cada ocorrência.

Exercício 9.13 Crie um programa que imprima as linhas de um arquivo. Esse programa deve receber três parâmetros pela linha de comando: o nome do arquivo, a linha inicial e a última linha a imprimir.

Exercício 9.14 Crie um programa que leia um arquivo-texto e elimine os espaços repetidos entre as palavras e no fim das linhas. O arquivo de saída também não deve ter mais de uma linha em branco repetida.

Exercício 9.15 Altere o programa da listagem 7.5, o jogo da forca. Utilize um arquivo em que uma palavra seja gravada a cada linha. Use um editor de textos para gerar o arquivo. Ao iniciar o programa, utilize esse arquivo para carregar a lista de palavras. Experimente também perguntar o nome do jogador e gerar um arquivo com o número de acertos dos cinco melhores.

Usando arquivos, podemos gravar dados de forma a reutilizá-los nos programas. Até agora, tudo que inserimos ou digitamos nos programas era perdido no fim da execução. Com arquivos, podemos registrar essa informação e reutilizá-la. Arquivos podem ser utilizados para fornecer uma grande quantidade de dados aos programas. Vejamos um exemplo no qual gravaremos nomes e telefones em um arquivo-texto. Utilizaremos um menu pra deixar o usuário decidir quando ler o arquivo e quando gravá-lo.

Execute o programa da listagem 9.7 e analise-o cuidadosamente.

► Listagem 9.7 – Controle de uma agenda de telefones

```

agenda = []
def pede_nome():
    return(input("Nome: "))
def pede_telefone():
    return(input("Telefone: "))
def mostra_dados(nome, telefone):
    print("Nome: %s Telefone: %s" % (nome, telefone))
def pede_nome_arquivo():
    return(input("Nome do arquivo: "))
def pesquisa(nome):
    mnome = nome.lower()
    for p, e in enumerate(agenda):
        if e[0].lower() == mnome:
            return p
    return None
def novo():
    global agenda
    nome = pede_nome()
    telefone = pede_telefone()
    agenda.append([nome, telefone])
def apaga():
    global agenda
    nome = pede_nome()
    p = pesquisa(nome)
    if p!=None:
        del agenda[p]
    else:
        print("Nome não encontrado.")
def altera():
    p = pesquisa(pede_nome())
    if p!=None:
        nome = agenda[p][0]
        telefone = agenda[p][1]
        print("Encontrado:")

```

```

        mostra_dados(nome, telefone)
        nome = pede_nome()
        telefone = pede_telefone()
        agenda[p]=[nome, telefone]
    else:
        print("Nome não encontrado.")
def lista():
    print("\nAgenda\n-----")
    for e in agenda:
        mostra_dados(e[0], e[1])
    print("-----\n")
def lê():
    global agenda
    nome_arquivo = pede_nome_arquivo()
    arquivo = open(nome_arquivo, "r", encoding="utf-8")
    agenda = []
    for l in arquivo.readlines():
        nome, telefone = l.strip().split("#")
        agenda.append([nome, telefone])
    arquivo.close()
def grava():
    nome_arquivo = pede_nome_arquivo()
    arquivo = open(nome_arquivo, "w", encoding="utf-8")
    for e in agenda:
        arquivo.write("%s#%s\n" % (e[0], e[1]))
    arquivo.close()
def valida_faixa_inteiro(pergunta, inicio, fim):
    while True:
        try:
            valor = int(input(pergunta))
            if inicio <=valor <=fim:
                return(valor)
            except ValueError:
                print("Valor inválido, favor digitar entre %d e %d" % (inicio, fim))

```

```

def menu():
    print("""
1 - Novo
2 - Altera
3 - Apaga
4 - Lista
5 - Grava
6 - Lê
0 - Sai
""")

    return valida_faixa_inteiro("Escolha uma opção: ",0,6)

while True:
    opção = menu()

    if opção == 0:
        break
    elif opção == 1:
        novo()
    elif opção == 2:
        altera()
    elif opção == 3:
        apaga()
    elif opção == 4:
        lista()
    elif opção == 5:
        grava()
    elif opção == 6:
        lê()

```

Os exercícios a seguir modificam o programa da listagem 97.

Exercício 9.16 Explique como os campos nome e telefone são armazenados no arquivo de saída.

Exercício 9.17 Altere o programa para exibir o tamanho da agenda no menu principal.

Exercício 9.18 O que acontece se nome ou telefone contiverem o caractere usado como separador em seus conteúdos? Explique o problema e proponha uma solução.

Exercício 9.19 Altere a função lista para que exiba também a posição de cada elemento.

Exercício 9.20 Adicione a opção de ordenar a lista por nome no menu principal.

Exercício 9.21 Nas funções de altera e apaga, peça que o usuário confirme a alteração e exclusão do nome antes de realizar a operação em si.

Exercício 9.22 Ao ler ou gravar uma nova lista, verifique se a agenda atual já foi gravada. Você pode usar uma variável para controlar quando a lista foi alterada (novo, altera, apaga) e reinicializar esse valor quando ela for lida ou gravada.

Exercício 9.23 Altere o programa para ler a última agenda lida ou gravada ao inicializar. Dica: utilize outro arquivo para armazenar o nome.

Exercício 9.24 O que acontece com a agenda se ocorrer um erro de leitura ou gravação? Explique.

Exercício 9.25 Altere as funções `pede_nome` e `pede_telefone` de forma a receberem um parâmetro opcional. Caso esse parâmetro seja passado, utilize-o como retorno caso a entrada de dados seja vazia.

Exercício 9.26 Altere o programa de forma a verificar a repetição de nomes. Gere uma mensagem de erro caso duas entradas na agenda tenham o mesmo nome.

Exercício 9.27 Modifique o programa para também controlar a data de aniversário e o e-mail de cada pessoa.

Exercício 9.28 Modifique o programa de forma a poder registrar vários telefones para a mesma pessoa. Permita também cadastrar o tipo de telefone: celular, fixo, residência, trabalho ou fax.

9.5 Geração de HTML

Páginas web nada mais são que a combinação de textos e imagens, interligadas por links. Hoje essa definição vai além com a crescente popularidade de vídeo e as chamadas Aplicações de Internet Ricas (*Rich Internet Applications*), normalmente escritas em Javascript. Como aqui tratamos apenas de introduzir conceitos de programação, vejamos como utilizar o que já sabemos sobre arquivos para gerar *home pages* ou páginas web simples.

Toda página web é escrita em uma linguagem de marcação chamada HTML (*Hypertext Mark-up Language* – Linguagem de Marcação de Hipertexto). Primeiro, precisamos entender o que são marcações. Como o formato HTML é definido apenas com texto simples, ou seja, sem caracteres especiais de controle, ele utiliza marcações, que são sequências especiais de texto delimitado pelos caracteres de menor (<) e maior (>). Essas sequências são chamadas de *tags* e podem iniciar ou finalizar um elemento. O elemento de mais alto nível de um documento HTML é chamado de <html>. Escreveremos nossas páginas web entre as tags <html> e </html>, onde a primeira marca o início do documento; e a segunda, seu fim. Diremos que <html> é a *tag* que inicia o elemento *html*, e que a tag </html> é a *tag* que o finaliza. Vamos seguir os conselhos definidos na especificação do padrão HTML 5 e definir a página inicial como na listagem 9.7. Observe que à esquerda de cada linha apresentamos seu número. Você não deve digitar esse número em seu arquivo.

► Listagem 9.7 – Página web simples ola.html

```
1  <!DOCTYPE html>
2  <html lang="pt-BR">
3  <head>
4  <meta charset="utf-8">
```

```
5  <title>Título da Página</title>
6  </head>
7  <body>
8  Olá!
9  </body>
10 </html>
```

Você pode escrever essa página usando um editor de textos. Grave o arquivo com o nome de *ola.html*. Abra em seu browser (navegador) de internet preferido: Internet Explorer, FireFox, Chrome, Safari, Opera etc. A forma mais fácil de abrir um arquivo em disco no browser é selecionar o menu **Arquivo** e depois a opção **Abrir**. Escolha o arquivo *ola.html* e veja o que aparece na tela. Se os acentos não aparecerem corretamente, verifique se você gravou o arquivo como texto UTF-8, da mesma forma que fazemos com os programas.

Vamos analisar essa página linha por linha. A linha 1 faz parte do formato e deve sempre ser incluída: ela indica que o documento foi escrito no formato HTML.

A linha 2 contém a *tag html*, mas não apenas o nome da *tag* e o atributo *lang*, cujo valor é *pt-BR*. Observe que o início da *tag* é marcado com < e o fim com > e que *html lang="pt-BR"* foi escrito na mesma *tag*, ou seja, entre os símbolos de < e >. O atributo *lang* indica a língua utilizada ao escrever o documento: o valor *pt-BR* indica português do Brasil.

A linha 4 especifica que o arquivo utiliza o formato de codificação de caracteres UTF-8 e permite escrever em português sem problemas. No entanto, você deve garantir que realmente escreveu o arquivo usando UTF-8. No Windows, você pode usar o editor PS-Pad para gerar arquivos UTF-8; no Linux e no Mac OS X esse tipo de codificação é usado por padrão e não deve causar problemas. Um sinal de problema de codificação é se você não conseguir ler Olá na tela ou se o título da página não aparecer corretamente. Se alterar o arquivo HTML, você deve solicitar que o browser o recarregue (*reload*) para visualizar as mudanças.

A linha 5 traz o título da página. Observe que o título é simplesmente escrito entre <title> e </title>. Observe também que escrevemos *title* e *meta* dentro do elemento *head* que começa na linha 3 e termina na linha 6. O formato HTML especifica que elementos devem ser escritos e onde podemos escrevê-los. Para mais detalhes, você pode consultar a especificação do formato HTML 5 na internet. Por enquanto, considere que nosso primeiro arquivo *html* será utilizado como um modelo em nossos programas.

Entre as linhas 7 e 9, definimos o elemento `body`, que é o corpo da página web. A linha 8 contém apenas a mensagem Olá!. Finalmente, na linha 10, temos a tag que finaliza o elemento `html`, ou seja, `</html>`, marcando o fim do documento. Experimente alterar esse arquivo, escrevendo uma nova mensagem na linha 8. Experimente também digitar várias linhas de texto (você pode copiar e colar um texto de 10 ou 12 linhas). Não se esqueça de salvar o arquivo com as modificações e de recarregar a página no browser para visualizá-las.

Como páginas web são arquivos texto, podemos criá-las facilmente em Python. Veja o programa da listagem 9.8 que cria uma página HTML.

► Listagem 9.8 – Criação de uma página inicial em Python

```
página=open("página.html","w", encoding="utf-8")
página.write("<!DOCTYPE html>\n")
página.write("<html lang=\"pt-BR\">\n")
página.write("<head>\n")
página.write("<meta charset=\"utf-8\">\n")
página.write("<title>Título da Página</title>\n")
página.write("</head>\n")
página.write("<body>\n")
página.write("Olá!")
for i in range(10):
    página.write("<p>%d</p>\n" % i)
página.write("</body>\n")
página.write("</html>\n")
página.close()
```

Simplesmente escrevemos nossa página dentro de um programa. Observe que ao escrevermos aspas dentro de aspas, como na tag `html`, onde o valor de `lang` é `utf-8` e é escrito entre aspas, tivemos o cuidado de colocar uma barra antes das aspas, informando, dessa forma, que as aspas não fazem parte do programa e que não marcam o fim da string. Um detalhe muito importante é o parâmetro extra que passamos em `open`. O parâmetro `encoding="utf-8"` informa que queremos os arquivos com a codificação UTF-8. Essa codificação tem que ser a mesma declarada no arquivo `html`, caso contrário, teremos problemas com caracteres acentuados.

Execute o programa e abra o arquivo `página.html` em seu browser. Modifique o programa para gerar 100 parágrafos em vez de 10. Execute-o novamente e veja o resultado no browser (abrindo o arquivo novamente ou clicando em recarregar).

Em nossos primeiros testes, vimos que escrevendo várias linhas de texto no corpo da página não alteramos o formato de saída da página. Isso porque o formato HTML ignora espaços em branco repetidos e quebras de linha. Se quisermos mostrar o texto em vários parágrafos, devemos utilizar o elemento `p`. Assim, as tags `<p>` e `</p>` marcam o início e o fim de um parágrafo, como no programa da listagem 9.8.

Python oferece recursos mais interessantes para trabalhar com strings, como aspas triplas que permitem escrever longos textos mais facilmente. Elas funcionam como as aspas, mas permitem digitar a mensagem em várias linhas. Vejamos o programa da listagem 9.9.

► Listagem 9.9 – Uso de aspas triplas para escrever as strings

```
página=open("página.html","w", encoding="utf-8")
página.write("""
<!DOCTYPE html>
<html lang="pt-BR">
<head>
<meta charset="utf-8">
<title>Título da Página</title>
</head>
<body>
Olá!
""")
for i in range(10):
    página.write("<p>%d</p>\n" % i)
página.write("""
</body>
</html>
""")
```

página.close()

Outra vantagem é que não precisamos colocar uma barra antes das aspas, pois agora o fim das aspas também é triplo ("""), não sendo mais necessário diferenciá-lo. Experimente agora modificar o programa retirando os \n do final das linhas. Execute-o e veja o resultado no browser. Você deve perceber que mesmo sem utilizar quebras de linhas no arquivo, o resultado permaneceu o mesmo. Isso porque espaços adicionais (repetidos) e quebras de linha são ignorados (ou quase ignorados) em HTML. No entanto, observe que a página criada é mais difícil de ler para nós.

O formato HTML contém inúmeras tags que são continuamente revisadas e expandidas. Além das tags que já conhecemos, temos também as que marcam os cabeçalhos dos documentos: `h1`, `h2`, `h3`, `h4`, `h5` e `h6`. Os números de 1 a 6 são utilizados para indicar o nível da seção ou subseção do documento, como faríamos no Microsoft Word ou OpenOffice com estilos de cabeçalho.

Vejamos como gerar páginas web a partir de um dicionário. Vamos utilizar as chaves como título das seções, e o valor como conteúdo do parágrafo (Listagem 9.10).

► Listagem 9.10 – Geração de uma página web a partir de um dicionário

```
filmes={  
    "drama": ["Cidadão Kane", "O Poderoso Chefão"],  
    "comédia": ["Tempos Modernos", "American Pie", "Dr. Dolittle"],  
    "policial": ["Chuva Negra", "Desejo de Matar", "Difícil de Matar"],  
    "guerra": ["Rambo", "Platoon", "Tora! Tora! Tora!"]}  
  
pagina=open("filmes.html", "w", encoding="utf-8")  
pagina.write("")  
<!DOCTYPE html>  
<html lang="pt-BR">  
<head>  
<meta charset="utf-8">  
<title>Filmes</title>  
</head>  
<body>  
    """)  
    for c, v in filmes.items():  
        pagina.write("<h1>%s</h1>" % c)  
        for e in v:  
            pagina.write("<h2>%s</h2>" % e)  
pagina.write("")  
</body>  
</html>  
    """)  
pagina.close()
```

Exercício 9.29 Modifique o programa da listagem 9.10 para utilizar o elemento `p` em vez de `h2` nos filmes.

Exercício 9.30 Modifique o programa da listagem 9.10 para gerar uma lista html, usando os elementos `ul` e `li`. Todos os elementos da lista devem estar dentro do elemento `ul`, e cada item dentro de um elemento `li`. Exemplo:

```
<ul><li>Item1</li><li>Item2</li><li>Item3</li></ul>
```

Você pode ler mais sobre HTML e CSS (*Cascading Style-Sheets*) para gerar páginas profissionais em Python.

9.6 Arquivos e diretórios

Agora que já sabemos ler, criar e alterar arquivos, vamos aprender como listá-los, manipular diretórios, verificar o tamanho e a data de criação de arquivos em disco.

Para começar, precisamos que os programas saibam de onde estão sendo executados. Vamos utilizar a função `getcwd` do módulo `os` para obter esse valor (Listagem 9.11).

► Listagem 9.11 – Obtenção do diretório atual

```
>>> import os  
>>> os.getcwd()
```

Também podemos trocar de diretório em Python, mas, antes, precisamos criar alguns diretórios para teste. Na linha de comando, digite:

```
mkdir a  
mkdir b  
mkdir c
```

Se preferir, você pode criar as pastas `a`, `b` e `c` usando o Windows Explorer, o Finder ou outro utilitário de sua preferência. Agora, vejamos a listagem 9.12.

► Listagem 9.12 – Troca de diretório

```
import os  
os.chdir("a")
```

```
print(os.getcwd())
os.chdir("../")
print(os.getcwd())
os.chdir("b")
print(os.getcwd())
os.chdir("../c")
print(os.getcwd())
```

A função `chdir` muda o diretório atual, por isso a função `getcwd` apresenta valores diferentes. Você pode referenciar um arquivo apenas usando seu nome se ele estiver no diretório atual ou de trabalho. O que `chdir` faz é mudar o diretório de trabalho, permitindo que você acesse seus arquivos mais facilmente.

Quando passamos .. para `chdir` ou qualquer outra função que manipule arquivos e diretórios, estamos nos referindo ao diretório pai ou de nível superior. Por exemplo, considere o diretório z, que contém os diretórios h, i, j. Dentro do diretório (ou pasta) j, temos o diretório k (Figura 9.1). Quando um diretório está dentro de outro, dizemos que o diretório pai contém o diretório filho. No exemplo, k é filho de j, e j é pai de k. Temos também que h, i, j são todos filhos de z, ou seja, z é o pai de h, i, j.



Figura 9.1 – Estrutura de diretórios no Windows Explorer do Windows 8.1.

Esse tipo de endereçamento é sempre relativo ao diretório corrente, por isso é tão importante saber em que diretório estamos. Se o diretório corrente for k, .. refere-se a j. Mas se o diretório atual for i, .. refere-se a z. Podemos também combinar vários .. no mesmo endereço ou caminho (*path*). Por exemplo, se o diretório atual for k, ../../ será uma referência a z e assim por diante. Também podemos usar o diretório pai para navegar entre os diretórios. Por exemplo, para acessar h, estando em k, podemos escrever ../../h.

Podemos também criar diretórios nos programas utilizando a função `mkdir` (Listagem 9.13).

► Listagem 9.13 – Criação de diretórios

```
import os
os.mkdir("d")
os.mkdir("e")
os.mkdir("f")
print(os.getcwd())
os.chdir("d")
print(os.getcwd())
os.chdir("../e")
print(os.getcwd())
os.chdir("../")
print(os.getcwd())
os.chdir("f")
print(os.getcwd())
```

A função `mkdir` cria apenas um diretório de cada vez. Se precisar criar um diretório, sabendo ou não se os superiores foram criados, use a função `makedirs`, que cria todos os diretórios intermediários de uma só vez.

► Listagem 9.14 – Criação de intermediários de uma só vez

```
import os
os.makedirs("avô/pai/filho")
os.makedirs("avô/mãe/filha")
```

Abra o diretório atual usando o Windows Explorer ou Finder, no Mac OS X, para ver os diretórios criados (Figura 9.2).

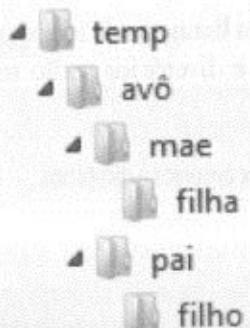


Figura 9.2 – Estrutura de diretórios criada em disco no Windows Explorer do Windows 8.

Você pode mudar o nome de um diretório ou arquivo, ou seja, renomeá-lo usando a função `rename` (Listagem 9.15).

► Listagem 9.15 – Alteração do nome de arquivos e diretórios

```
import os
os.mkdir("velho")
os.rename("velho","novo")
```

A função `rename` também pode ser utilizada para mover arquivos, bastando especificar o mesmo nome em outro diretório (Listagem 9.16).

► Listagem 9.16 – Alteração do nome de arquivos e diretórios

```
import os
os.makedirs("avô/pai/filho")
os.makedirs("avô/mãe/filha")
os.rename("avô/pai/filho","avô/mãe/filho")
```

Se você quiser apagar um diretório, utilize a função `rmdir`. Se quiser apagar um arquivo, use a função `remove` (Listagem 9.17).

► Listagem 9.17 – Exclusão de arquivos e diretórios

```
import os
# Cria um arquivo e o fecha imediatamente
open("morimbundo.txt","w").close()
os.mkdir("vago")
os.rmdir("vago")
os.remove("morimbundo.txt")
```

Podemos também solicitar uma listagem de todos os arquivos e diretórios usando a função `listdir`. Os arquivos e diretórios serão retornados como elementos da lista (Listagem 9.18).

► Listagem 9.18 – Listagem do nome de arquivos e diretórios

```
import os
print(os.listdir("."))
```

Onde `.` significa o diretório atual. A lista contém apenas o nome de cada arquivo. Vamos ver como obter o tamanho do arquivo e as datas de criação, acesso e modificação a seguir com o módulo `os.path`.

O módulo `os.path` traz várias outras funções que vamos utilizar para obter mais informações sobre os arquivos em disco. As duas primeiras são `isdir` e `isfile`, que retornam `True` se o nome passado for um diretório ou um arquivo respectivamente (Listagem 9.19).

► Listagem 9.19 – Verificação se é diretório ou arquivo

```
import os
import os.path
for a in os.listdir("."):
    if os.path.isdir(a):
        print("%s/" % a)
    elif os.path.isfile(a):
        print("%s" % a)
```

Execute o programa e veja que imprimimos apenas os nomes de diretórios e arquivos, sendo que adicionamos uma barra no final dos nomes de diretório.

Podemos também verificar se um diretório ou arquivo já existe com a função `exists` (Listagem 9.20).

► Listagem 9.20 – Verificação se um diretório ou arquivo já existe

```
import os.path
if os.path.exists("z"):
    print("O diretório z existe.")
else:
    print("O diretório z não existe.")
```

Exercício 9.31 Crie um programa que corrija o da listagem 9.20 de forma a verificar se `z` existe e é um diretório.

Exercício 9.32 Modifique o programa da listagem 9.20 de forma a receber o nome do arquivo ou diretório a verificar pela linha de comando. Imprima se existir e se for um arquivo ou um diretório.

Exercício 9.33 Crie um programa que gera uma página html com links para todos os arquivos jpg e png encontrados a partir de um diretório informado na linha de comando.

Temos também outras funções que retornam mais informações sobre arquivos e diretórios como seu tamanho e datas de modificação, criação e acesso (Listagem 9.21).

► Listagem 9.21 – Obtenção de mais informações sobre o arquivo

```
import os
import os.path
import time
import sys
nome = sys.argv[1]
print("Nome: %s" % nome)
print("Tamanho: %d" % os.path.getsize(nome))
print("Criado: %s" % time.ctime(os.path.getctime(nome)))
print("Modificado: %s" % time.ctime(os.path.getmtime(nome)))
print("Acessado: %s" % time.ctime(os.path.getatime(nome)))
```

Onde `getsize` retorna o tamanho do arquivo em bytes, `getctime` retorna a data e hora de criação, `getmtime` de modificação e `getatime` de acesso. Observe que chamamos `time.ctime` para transformar a data e hora retornadas por `getmtime`, `getatime` e `getctime` em string. Isso é necessário porque o valor retornado é expresso em segundos e precisa ser corretamente convertido para ser exibido.

9.7 Um pouco sobre o tempo

O módulo `time` traz várias funções para manipular o tempo. Uma delas foi apresentada na seção anterior: `time.ctime`, que converte um valor em segundos após 01/01/1970 em string. Temos também a função `gmtime`, que retorna uma tupla com componentes do tempo separados em elementos. Vejamos alguns exemplos no interpretador (Listagem 9.22).

► Listagem 9.22 – Obtenção das horas em Python

```
>>> import time
>>> agora=time.time()
>>> agora
1277310220.906508
>>> time.ctime(agora)
'Wed Jun 23 18:23:40 2010'
>>> agora2=time.localtime()
>>> agora2
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=23, tm_hour=18, tm_min=23,
tm_sec=40, tm_wday=2, tm_yday=174, tm_isdst=1)
>>> time.gmtime(agora)
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=23, tm_hour=16, tm_min=23,
tm_sec=40, tm_wday=2, tm_yday=174, tm_isdst=0)
```

A função `time.time` retorna a hora atual em segundos, usando o horário de Greenwich ou UTC (Tempo Universal Coordenado). Na listagem 9.22, atribuímos seu resultado à variável `agora` e convertemos em string usando `time.ctime`. Se você deseja trabalhar com a hora em seu fuso horário, utilize `time.localtime`, como mostrado com a variável `agora2`. Observe que `time.localtime` retornou uma tupla e que `time.time` retorna apenas um número. Podemos utilizar a função `gmtime` para converter a variável `agora` em uma tupla de nove elementos, como a retornada por `time.localtime` (Tabela 9.2).

Tabela 9.2 – Elementos da tupla de tempo retornada por `gmtime`

Posição	Nome	Descrição
0	<code>tm_year</code>	ano
1	<code>tm_mon</code>	mês
2	<code>tm_mday</code>	dia
3	<code>tm_hour</code>	hora
4	<code>tm_min</code>	minutos
5	<code>tm_sec</code>	segundos
6	<code>tm_wday</code>	dia da semana entre 0 e 6, onde segunda-feira é 0
7	<code>tm_yday</code>	dia do ano, varia de 1 a 366.
8	<code>tm_isdst</code>	horário de verão, onde 1 indica estar no horário de verão.

Podemos também acessar esses dados por nome (Listagem 9.23).

Listagem 9.23 – Obtenção de data e hora por nome

```
import time
agora=time.localtime()
print("Ano: %d" % agora.tm_year)
print("Mês: %d" % agora.tm_mon)
print("Dia: %d" % agora.tm_mday)
print("Hora: %d" % agora.tm_hour)
print("Minuto: %d" % agora.tm_min)
print("Segundo: %d" % agora.tm_sec)
print("Dia da semana: %d" % agora.tm_wday)
print("Dia no ano: %d" % agora.tm_yday)
print("Horário de verão: %d" % agora.tm_isdst)
```

A função `time.strftime` permite a formatação do tempo em string. Você pode passar o formato desejado para a string, seguindo os códigos de formatação da tabela 9.3.

Tabela 9.3 – Códigos de formatação de `strftime`

Código	Descrição
%a	dia da semana abreviado
%A	nome do dia da semana
%b	nome do mês abreviado
%B	nome do mês completo
%c	data e hora conforme configuração regional
%d	dia do mês (01-31)
%H	hora no formato 24 h (00-23)
%I	hora no formato 12 h
%j	dia do ano 001-366
%m	mês (01-12)
%M	minutos (00-59)
%p	AM ou PM
%S	segundos (00-61)
%U	número da semana (00-53), onde a semana 1 começa após o primeiro domingo.
%w	dia da semana (0-6) onde 0 é o domingo

Código	Descrição
%W	número da semana (00-53), onde a semana 1 começa após a primeira segunda-feira
%x	representação regional da data
%X	representação regional da hora
%y	ano (00-99)
%Y	ano com 4 dígitos
%Z	nome do fuso horário
%	símbolo de %

Se precisar converter uma tupla em segundos, utilize a função `timetuple` do módulo `calendar`. Se precisar trabalhar com data e hora em seus programas, consulte a documentação do Python sobre os módulos `time`, `datetime`, `calendar` e `locale`.

Exercício 9.34 Altere o programa da listagem 7.45, o jogo da forca. Dessa vez, utilize as funções de tempo para cronometrar a duração das partidas.

9.8 Uso de caminhos

Uma tarefa comum quando se trabalha com arquivos é manipular caminhos. Como essa tarefa depende do sistema operacional – e cada sistema tem suas próprias características, como “/” no Linux e no Mac OS X para separar o nome dos diretórios e “\” no Windows –, a biblioteca padrão do Python traz algumas funções interessantes. Aqui veremos as mais importantes: a documentação do Python traz a lista completa. Vejamos alguns exemplos dessas funções quando chamadas no interpretador Python na listagem 9.24.

► Listagem 9.24 – Uso de caminhos

```
>>> import os.path
>>> caminho="i/j/k"
>>> os.path.abspath(caminho)
'C:\\Python31\\i\\j\\k'
>>> os.path.basename(caminho)
'k'
>>> os.path.dirname(caminho)
'i/j'
```

```
>>> os.path.split(caminho)
('1/j', 'k')
>>> os.path.splitext("arquivo.txt")
('arquivo', '.txt')
>>> os.path.splitdrive("c:/Windows")
('c:', '/Windows')
```

A função `abspath` retorna o caminho absoluto do path passado como parâmetro. Se o caminho não começar com /, o diretório atual é acrescentado, retornando o caminho completo a partir da raiz. No caso do Windows, incluindo também a letra do disco (*drive*), no caso C:.

Observe que o caminho “j/k” é apenas uma string. As funções de `os.path` não verificam se esse caminho realmente existe. Na realidade, `os.path`, na maioria das vezes, oferece apenas funções inteligentes para manipulação de caminhos como strings.

A função `basename` retorna apenas a última parte do caminho, no exemplo, k. Já a função `dirname` retorna o caminho à esquerda da última barra. Mais uma vez, essas funções não verificam se k é um arquivo ou diretório. Considere que `basename` retorna a parte do caminho à direita da última barra, e que `dirname` retorna o caminho à esquerda. Você até pode combinar o resultado dessas duas funções com a função `split`, que retorna uma tupla onde os elementos são iguais aos resultados de `dirname` e `basename`.

No Windows, você pode também usar a função `splitdrive` para separar a letra do drive do caminho em si. A função retorna uma tupla, onde a letra do drive é o primeiro elemento; e o restante do caminho, o segundo.

A função `join` junta os componentes de um caminho, separando-os com barras, se necessário. Veja o exemplo da listagem 9.25. No Windows, a função verifica se o nome termina com “:” e, nesse caso, não insere uma barra, permitindo a criação de um caminho relativo. Podemos combinar o resultado dessa função com `abspath` e obter um caminho a partir da raiz. Veja que a manipulação da letra do drive é feita automaticamente.

► Listagem 9.25 – Combinação dos componentes de um caminho

```
>>> import os.path
>>> os.path.join("c:", "dados", "programas")
'C:\dados\programas'
>>> os.path.abspath(os.path.join("c:", "dados", "programas"))
'C:\Python31\dados\programas'
```

9.9 Visita a todos os subdiretórios recursivamente

A função `os.walk` facilita a navegação em uma árvore de diretórios. Imagine que deseje percorrer todos os diretórios a partir de um diretório inicial, retornando o nome do diretório sendo visitado (*raiz*), os diretórios encontrados dentro do diretório sendo visitado (*diretórios*) e uma lista de seus arquivos (*arquivos*). Observe e execute o programa da listagem 9.26. Você deve executá-lo passando o diretório inicial a visitar na linha de comando.

► Listagem 9.26 – Árvore de diretórios sendo percorrida

```
import os
import sys
for raiz, diretorios, arquivos in os.walk(sys.argv[1]):
    print("\nCaminho:", raiz)
    for d in diretorios:
        print("  %s/" % d)
    for f in arquivos:
        print("  %s" % f)
    print("%d diretório(s), %d arquivo(s)" % (len(diretorios), len(arquivos)))
```

A grande vantagem da função `os.walk` é que ela visita automaticamente todos os subdiretórios dentro do diretório passado como parâmetro, fazendo-o repetidamente até navegar a árvore de diretórios completa.

Combinando a função `os.walk` às funções de manipulação de diretórios e arquivos que já conhecemos, você pode escrever programas para manipular árvores de diretórios completas.

Exercício 9.35 Utilizando a função `os.walk`, crie uma página HTML com o nome e tamanho de cada arquivo de um diretório passado e de seus subdiretórios.

Exercício 9.36 Utilizando a função `os.walk`, crie um programa que calcule o espaço ocupado por cada diretório e subdiretório, gerando uma página html com os resultados.