Capítulo 7

Arquivos

7.1 Persistência

Até agora, aprendemos como escrever programas e comunicar nossas intenções para a *Unidade Central de Processamento* usando a execução condicional, funções e iterações. Aprendemos como criar e usar estruturas de dados na *Memória Principal*. A CPU e a memória são onde o nosso software funciona e é executado, é onde todo o "pensamento" acontece.

Mas se você se lembrar de nossas discussões de arquitetura de hardware, uma vez que a energia é desligada, qualquer coisa armazenada na CPU ou memória principal é apagada. Então, até agora, nossos programas foram apenas exercícios divertidos para aprender Python.

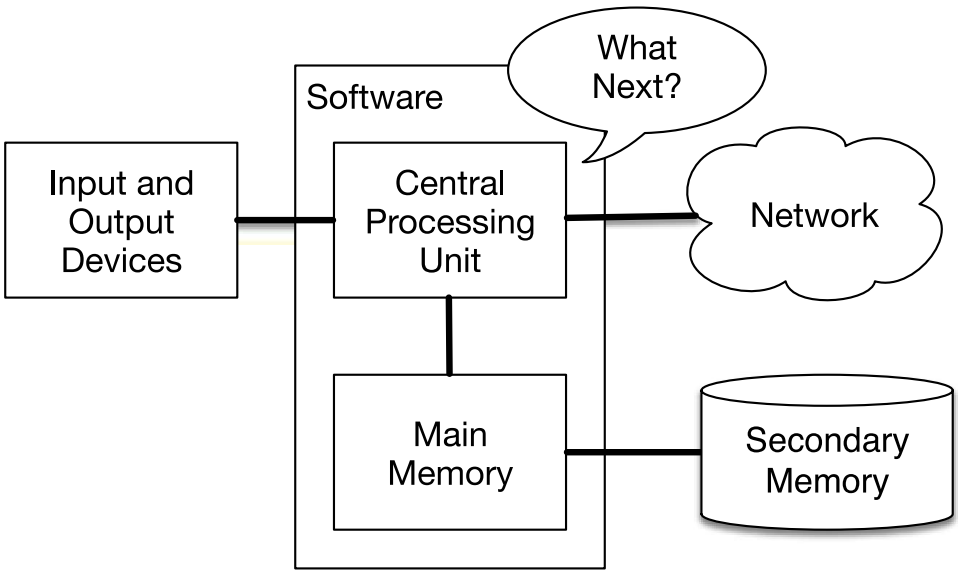

Figure 7.1: Secondary Memory

Neste capítulo, começamos a trabalhar com *Memória Secundária* (ou arquivos). Esta não é apagada quando a energia é desligada. Ou, no caso de uma unidade flash USB, os dados que escrevemos de nossos programas podem ser removidos do sistema e transportados para outro sistema.

Nós nos concentraremos primeiramente na leitura e escrita de arquivos de texto, como aqueles que criamos em um editor de texto. Mais tarde, veremos como trabalhar com arquivos de banco de dados que são arquivos binários, especificamente projetados para serem lidos e gravados através de software de banco de dados.

7.2 Abrindo um arquivo

Quando queremos ler ou escrever um arquivo (digamos no seu disco rígido), primeiro devemos *abrir* ele. Fazendo isso há uma comunicação com seu sistema operacional, que sabe onde os dados para cada arquivo são armazenados. Quando se abre um arquivo, você está pedindo para o sistema operacional achá-lo por nome e ter certeza que ele existe. Nesse exemplo, nós abrimos o arquivo *mbox.txt*, o qual deveria estar armazenado na mesma pasta que você está quando inicia o Python. Você pode baixar este arquivo em [www.py4e.com/code3/mbox.txt]

```
>>> Arquivo = open('mbox.txt.')
>>> print(Arquivo)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

Se `open`for bem-sucedido, o sistema operacional retornará um *identificador de arquivo*. Este não é a informação real contida no arquivo, mas em vez disso, é um "identificador" que podemos usar para ler os dados. Você recebe um identificador se o arquivo solicitado existe e você tem as permissões apropriadas para lê-lo.

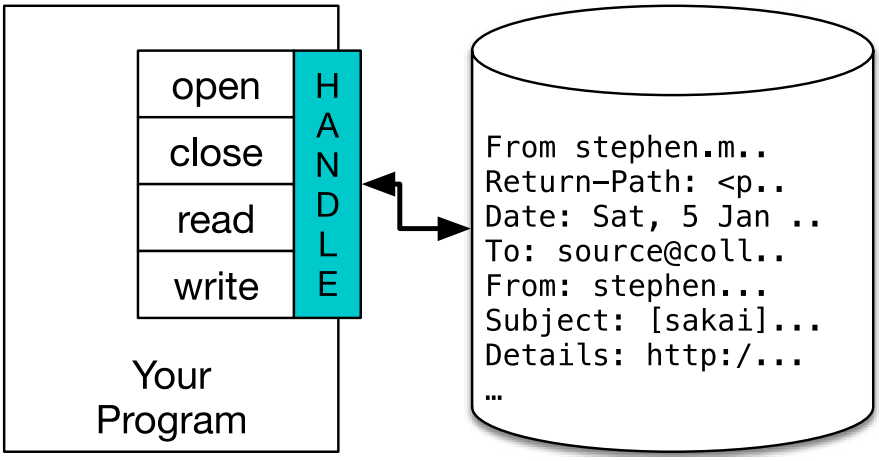

Figure 7.2: A File Handle

Se o arquivo não existir, `open` falhará com um Traceback e você não receberá um identificador para acessar o conteúdo dele:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

Mais tarde, usaremos `try` e `except` para lidar mais graciosamente com a situação em que tentamos abrir um arquivo que não existe.

7.3 Arquivos de texto e linhas

Um arquivo de texto pode ser pensado como uma sequência de linhas, assim como uma string em python pode ser pensada como uma sequência de caracteres. Por exemplo, essa é uma amostra de um arquivo de texto que grava a atividade de emails de vários inivíduos em um projeto *open source* em desenvolvimento.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

Todo o arquivo de interações de email está disponível a partir de

www.py4e.com/code3/mbox.txt

e uma versão reduzida do arquivo está disponível a partir de

www.py4e.com/code3/mbox-short.txt

Eles estão em um formato padrão para um arquivo que contém várias mensagens de email. As linhas que começam com "From" separam as mensagens e as linhas que começam com "From:" fazem parte delas. Para mais informações sobre o formato mbox, consulte https://en.wikipedia.org/wiki/Mbox.

Para quebrar o arquivo em linhas, há um caractere especial que representa o "fim da linha" chamado de * newLine *

Em Python, representamos o caractere * newLine * como `\n` em *strings* constantes. Mesmo que pareça ser dois caracteres, é na verdade um só. Quando olhamos para a variável inserindo "algo" no intérprete, ele nos mostra o `\n` na *string*, mas quando usamos 'print' para mostrá-la, nós vemos a *string* quebrada em duas linhas pelo *newline*.

```
>>> algo = 'Olá\nmundo!'
>>> algo
'Olá\nmundo!'
>>> print(algo)
Olá
mundo!
>>> algo = 'X\nY'
>>> print(algo)
X
Y
>>> len(algo)
3
```

Você também pode ver que o comprimento da *string* `X\nY` é * três * porque o *newline* é um único caractere.

Então, quando olhamos para as linhas em um arquivo, precisamos * imaginar * que há um caractere especial invisível chamado *newline* no final de cada linha que marca o fim desta.

Portanto, este caractere separa os outros caracteres no arquivo em linhas.

7.4 Lendo arquivos

Uma vez que o *identificador de arquivo* não contém os dados deste, é bem fácil construir um laço `for` para ler e contar cada linha do arquivo:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    count = count + 1
print('Line Count:', count)

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

Podemos usar o identificador de arquivo como a sequencia no nosso loop. Ele simplesmente conta o número de linhas no arquivo e mostra elas. Uma tradução grosseira para o loop `for` em português é: "para cada linha no arquivo representada pelo identificador, adicione um na variável `contagem`."

A razão para a função `open` não ler todo o arquivo é que este pode ser bem grande, com muitos gigabytes de dados. A sentença `open` toma a mesma quantidade de tmepo independente do tamanho do arquivo. O loop na verdade que faz os dados serem lidos.

Quando o arquivo é lido com o `for` dessa maneira, Python tem cuidado de separar os dados em linhas diferente utilizando o *newline*. Ele lê cada linha até o fim e inclui o newline quando alcança o último caractere na variável `linha` para cada iteração do loop `for`.

Uma vez que o laço lê os dados uma linha por vez, ele pode eficientemente ler e contar as linhas em arquivos muito grandes sem esgotar a memória principal para armazenar os dados. O programa acima conta as linhas em um arquivo de qualquer tamanho usando bem pouca memória já que cada linha é lida, contada e descartada

Se você sabe que o tamanho do arquivo é relativamente pequeno comparado ao da sua memória principal, então pode ler o arquivo todo em uma *string* usando o método `read` do identificador de arquivo.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

Nesse exemplo, o conteúdo inteiro (todos os 94626 caracteres) do arquivo *mbox-short.txt* é lido e armazenado diretamente na variável `inp`. Estamos usando o fatiamento de *strings* para mostrar os primeiros 20 caracteres da *string* armazenada em `inp`.

Quando o arquivo é lido dessa forma, todos os caracteres, incluindo todos os caracteres de linhas e *newline* são uma grande *string* na variável `inp`. É uma boa ideia armazenar a saída de `read` como uma variável porque cada chamada de `read` cansa o recurso.

```
>>> fhand = open('mbox-short.txt')
>>> print(len(fhand.read()))
94626
>>> print(len(fhand.read()))
0
```

Lembre que essa forma da função `open` deveria apenas ser usada se os dados do arquivo confortavelmente cabem na sua memória principal do computador. Se o arquivo é muito grande para a memória, você deveria escrever seu programa para ler ele em pedaços usando um `for` ou um `while`.

7.5 Searching through a file

Quando você está procurando por dados em um arquivo, é um padrão muito comum em percorrer um arquivo ignorando a maioria das linhas e apenas processando as que vão de encontro com uma condição particular. Podemos combinar o padrão de ler um arquivo com os métodos da *sting* para construir um simples mecanismo de busca.

Por exemplo, se queremos ler um arquivo e apenas mostrar as linhas que começam com o prefixo "From:", podemos usar o método *startswith* para selecionar apenas as linhas que possuem o prefixo desejado:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    if line.startswith('From:'):
        print(line)

# Code: http://www.py4e.com/code3/search1.py
```

Quando esse programa é executado, obtemos a seguinte saída:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

A saída parece ótima uma vez que as únicas linhas que estamos vendo são as que começam com "From:", mas por que vemos também essas linhas em branco extras? Isso é devido ao caractere *newline* invisível. Cada uma das linhas termina com um, então a sentença `print` mostra a string na variável *linha* que inclui um *newline* e então o `print` adiciona outro, resultando no efeito de duplo espaçamento que observamos.

Nós podíamos usar o fatiamento das linhas para mostrar todos menos o último caractere, mas uma abordagem mais simples é usar o método *rstrip* que

We could use line slicing to print all but the last character, but a simpler approach is to use the *rstrip* method which tira os espaços em branco do lado direito da *stirng*, como pode ser observado:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:'):
        print(line)

# Code: http://www.py4e.com/code3/search2.py
```

Quando esse programa executa, obtemos a seguinte saída:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

Uma vez que seus programas de processamento de arquivos ficam mais complicados, você deve querer estruturar seus laços de pesquisa usando `continue`. A ideia básica dos laços de pesquisa é procurar por linhas "interessantes" e efetivamente pular as "desinteressantes". E então, quando encontramos uma de nosso interesse, fazemos algo com aquela linha.

Podemos estruturar o laço seguindo o padrão de pular as linhas desinteressantes seguindo o modelo:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:'):
        continue
    # Process our 'interesting' line
    print(line)

# Code: http://www.py4e.com/code3/search3.py
```

A saída do programa é a mesma. Em português, as linhas não interessantes são aquelas que não começam com "From:", as quais são puladas utilizando o `continue`. Para as linhas "interessantes" (isto é, aquelas que começam com "From:") realizamos o processo nelas.

Nós podemos usar o método `find` da *string* para simular um editor de texto que encontra linhas onde a *string* procurada está em qualquer lugar da linha. Uma vez que o `find` procura por uma ocorrência de uma *string* dentro de outra e ou retorna a posição dela ou -1 se não for encontrada, podemos escrever o seguinte laço para mostrar linhas que contém a *string* "@uct.ac.za" (isto é, elas vêm da Universidade de Cape Town na África do Sul):

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1: continue
    print(line)

# Code: http://www.py4e.com/code3/search4.py
```

A qual produz a seguinte saída:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

Aqui nós também usamos a forma contraída da sentença `if` quando colocamos o `continue` na mesma linha do `if`. Essa forma funciona do mesmo jeito que se o `continue` estivesse na próxima linha e indentado.

7.6 Deixando o usuário escolher o nome do arquivo

Nós realmente não queremos ter quer editar nosso código Python toda vez que quisermos processar um arquivo diferente. Pode ser mais útil pedir ao usuário para inserir a *string* do nome do arquivo cada vez que o programa for executado, então, nosso programa poderá ser usado em arquivos diferentes sem mudar o código Python.

Isso é bastante simples de fazer, lendo o nome do arquivo do usuário usando `input` como à seguir:

```
fname = input('Enter the file name: ')
fhand = open(fname)
```

```
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: http://www.py4e.com/code3/search6.py
```

Nós lemos o nome do arquivo do usuário e o colocamos numa variável nomeada `fname`, abrimos esse arquivo e então podemos executar o programa repetidas vezes em arquivos diferentes.

```
python pesquisa6.py
Digite o nome do arquivo: mbox.txt
Havia 1797 linhas de conteúdo em mbox.txt

python pesquisa6.py
Digite o nome do arquivo: mbox-short.txt
Havia 27 linhas de conteúdo em mbox-short.txt
```

Antes de dar uma olhada na próxima seção, dê uma olhada no programa acima e pergunte a si mesmo: "O que poderia dar errado aqui?" ou "O que nosso amiguinho usuário poderia fazer que pararia desgraçadamente a execução do nosso programinha com um erro de *traceback* e faria com que não parecêssemos mais tão legais aos olhos do usuário?"

7.7 Usando `try`, `except`, e `open`

Eu te falei pra não espiar, essa é a sua última chance.

E se nosso usuário digitar algo que não seja um nome de arquivo?

```
python pesquisa6.py
Digite o nome do arquivo: missing.txt
Traceback (most recent call last):
  File "pesquisa6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'

python pesquisa6.py
Digite o nome do arquivo: na na boo boo
Traceback (most recent call last):
  File "pesquisa6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'na na boo boo'
```

Não ria. Os usuários eventualmente farão todas as coisas possíveis que podem fazer para quebrar seus programas, seja de propósito ou com intenção maliciosa. De fato, uma parte importante de qualquer equipe de desenvolvimento de software é uma pessoa ou grupo chamado Quality Assurance (ou QA, abreviado) cujo trabalho

é fazer as coisas mais loucas possíveis na tentativa de quebrar o software que o programador criou.

A equipe de QA é responsável por encontrar as falhas no programa antes de entregá-lo aos usuários finais que possivelmente estarão comprando o software ou pagando nosso salário para fazer o software. Então a equipe de QA é a melhor amiga do programador.

Então, agora que vemos a falha no programa, podemos corrigi-la usando a estrutura `try` / `except`. Precisamos assumir que a chamada do `open` pode falhar e adicionar o código de recuperação da seguinte forma:

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: http://www.py4e.com/code3/search7.py
```

A função `exit` termina o programa. É uma função sem retorno. Agora, quando nosso usuário (ou equipe de QA) digitar bobagens ou nomes de arquivos que não existem, nós os "pegamos" e recuperamos com elegância.

```
python pesquisa7.py
Digite o nome do arquivo: mbox.txt
Havia 1797 linhas de conteúdo em mbox.txt

python pesquisa7.py
Digite o nome do arquivo: na na boo boo
File cannot be opened: na na boo boo
```

Proteger a chamada do `open` é um bom exemplo do uso adequado do `try` e `except` em um programa Python. Usamos o termo "Pythonico" quando estamos fazendo algo do tipo "Python". Podemos dizer que o exemplo acima é a maneira Python de abrir um arquivo.

Uma vez que você se torne mais habilidoso em Python, você pode interagir com outros programadores para decidir qual das duas soluções equivalentes para um problema é "mais Pythonico". O objetivo de ser "mais Pythonico" capta a noção de que a programação é parte engenharia e parte arte. Nem sempre estamos interessados em fazer algo funcionar, queremos também que a nossa solução seja elegante e seja apreciada como elegante pelos nossos pares.

7.8 Escrevendo arquivos

Para escrever um arquivo você tem que abri-lo com o modo "w" como segundo parâmetro:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

Se o arquivo já existir, abri-lo no modo de gravação limpa os dados antigos e começa de novo, por isso tome cuidado! Se o arquivo não existir, um novo será criado.

O método `write` do objeto de manipulação de arquivos coloca dados no arquivo, retornando o número de caracteres gravados. O modo de gravação padrão é texto para escrever (e ler) strings.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

Novamente, o arquivo mantém o controle de onde está, portanto, se você chamar `write` novamente, ele adicionará os novos dados ao final. Devemos nos certificar de gerenciar as extremidades das linhas à medida que escrevemos no arquivo inserindo explicitamente o caractere de *newline* quando queremos terminar uma linha. A instrução `print` anexa automaticamente uma nova linha, mas o método `write` não adiciona a nova linha automaticamente.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
24
```

Quando terminar de escrever, feche o arquivo para certificar-se de que o último bit de dados esteja fisicamente gravado no disco, para que ele não seja perdido se faltar energia.

```
>>> fout.close()
```

Poderíamos fechar os arquivos que abrimos para leitura também, mas podemos ser um pouco desleixados se abrirmos apenas alguns arquivos, pois o Python garante que todos os arquivos abertos sejam fechados quando o programa terminar. Quando estamos escrevendo arquivos, queremos fechar explicitamente os arquivos para não deixar nada ao acaso.

7.9 Debugging

Quando você está lendo ou escrevendo arquivos, pode ser que encontre problemas com espaços em branco. Esses erros podem ser difíceis de tratar porque espaços, tabulações e *newlines* são normalmente invisíveis:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

A função interna `repr` pode ajudar. Ela pega qualquer parâmetro como argumento e retorna uma representação em string desse objeto. Para strings, caracteres de espaço em branco são representados como sequências de barra invertida:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

Isso pode ajudar no tratamento de erros.

Um outro problema que você deve encontrar é que diferentes sistemas usam caracteres diferentes par indicar o fim de uma linha. Alguns usam o *newline*, representado por `\n`. Outros usam o caractere de retorno, o `\r`. ALguns usam ambos. Se você mover arquivos de um sistema para outro, essas inconsistências podem causam problemas.

Para a maioria dos sistemas, existem aplicações que convertem de um formato para o outro. Você pode achar elas (e ler mais sobre esse problema) em wikipedia.org/wiki/Newline. Ou, é claro, você pode escrever uma por conta própria.

7.10 Glossário

arquivo de texto Uma sequência de caracteres colocada em um armazenamento permanente, como um disco rígido

capturar Para prevenir que uma exceção encerre o programa. É usado com as sentenças `try`e `except`

Controle de Qualidade Uma pessoa ou time focado em assegurar a qualidade geral de um produto de software. O CA é frequentemente envolvido em testes de um produto e na identificação de problemas antes que o produto seja distribuído.

newline Um caractere especial usado em arquivos e *strings* para indicar o fim de uma linha.

Pythonico Uma técnica que funciona de forma elegante em Python. "Usar try e except é a forma *Pythonica* de contornar arquivos inexistentes"

7.11 Exercícios

Exercício 1: Escreva um programa que leia um arquivo e mostre o conteúdo deste (linha por linha), completamente em caixa alta. A execução do programa deverá ser a seguinte:

```
python shout.py
Digite o nome de um arquivo: mbox-short.txt
```

```
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN  5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
     BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
     SAT, 05 JAN 2008 09:14:16 -0500
```

Você pode baixar o arquivo em: www.py4e.com/code3/mbox-short.txt

Exercício 2: Escreva um programa que solicite um arquivo e então leia ele e procure por linhas da forma:

```
X-DSPAM-Confidence: 0.8475
```

Quando encontrar uma linha que inicie com "X-DSPAM-Confidence:" separe a linha do texto para extrair o número de ponto flutuante que ela contém. Conte essas linhas e então compute o total de valores de Confiança de Spam nelas. Quando chegar no fim do arquivo, mostre a média de Confiança de Spam.

```
Digite o nome de um arquivo: mbox.txt
Média de confiança de spam: 0.894128046745

Digite o nome de um arquivo: mbox-short.txt
Média de confiança de spam: 0.750718518519
```

Teste seu programa com os arquivos *mbox.txt* e *mbox-short.txt*

Exercício 2: Às vezes, quando os programadores estão entediados ou querem um pouco de diversão, eles adicionam um Easter Egg inofensivo em seus programas. Modifique o programa que solicita um arquivo ao usuário para que ele mostre uma mensagem engraçada quando o usuário deigitar no nome do arquivo "na na boo boo". O programa deve se comportar normalmente para todos os outros arquivos que existem e que não existem. Aqui está uma amostra da execução desse programa:

```
python egg.py
Digite o nome de um arquivo: mbox.txt
Há 1797 linhas de assunto em mbox.txt

python egg.py
Digite o nome de um arquivo: missing.tyxt
Arquivo não pôde ser aberto: missing.tyxt

python egg.py
Digite o nome de um arquivo: na na boo boo
NA NA BOO BOO PRA VOCÊ TAMBÉM!
```

Nós não estamos encorajando você a pôr *Easter Eggs* nos seus programas, isso é apenas um exercício.