Capítulo 7: Entrada e saída

Existem várias maneiras de apresentar a saída de um programa; os dados podem ser exibidos ou impressos em forma legível para seres humanos, ou escritos em arquivos para uso posterior. Este capítulo apresenta algumas possibilidades.

Refinando a Formatação de Saída

Até agora vimos duas maneiras de exibir valores no console interativo: escrevendo expressões e usando o comando `print`. Em programas, apenas o `print` gera saída. (Uma outra maneira é utilizar o método `write` de objetos arquivo; a saída padrão pode ser referenciada como `sys.stdout`. Veja a Referência da Biblioteca Python para mais informações sobre isto.)

Frequentemente é desejável mais controle sobre a formatação de saída do que simplesmente exibir valores separados por espaços. Existem duas formas de formatar a saída. A primeira é manipular strings através de fatiamento (slicing) e concatenação. Os tipos string têm métodos úteis para criar strings com tamanhos determinados, usando caracteres de preenchimento; eles serão apresentados a seguir. A segunda forma é usar o método `str.format`.

O módulo `string` tem uma classe `string.Template` que oferece uma outra maneira de inserir valores em strings.

Permanece a questão: como converter valores para strings? Felizmente, Python possui duas maneiras de converter qualquer valor para uma string: as funções `repr` e `str`.

A função `str` serve para produzir representações de valores que sejam legíveis para as pessoas, enquanto `repr` é para gerar representações que o interpretador Python consegue ler (caso não exista uma forma de representar o valor, a representação devolvida por `repr` produz um `SyntaxError` [N.d.T. `repr` procura gerar representações fiéis; quando isso é inviável, é melhor encontrar um erro do que obter um objeto diferente do original]). Para objetos que não têm uma representação adequada para consumo humano, `str` devolve o mesmo valor que `repr`. Muitos valores, tal como inteiros e estruturas como listas e dicionários, têm a mesma representação usando ambas funções. Strings e números de ponto flutuante, em particular, têm duas representações distintas.

Alguns exemplos:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1.0/7.0)
'0.142857142857'
>>> repr(1.0/7.0)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
```

```
>>> print s
The value of x is 32.5, and y is 40000...
>>> # O repr() de uma string acresenta aspas e contrabarras:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # O argumento de repr() pode ser qualquer objeto Python:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

A seguir, duas maneiras de se escrever uma tabela de quadrados e cubos::

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Note a vírgula final na linha anterior
...     print repr(x*x*x).rjust(4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1,11):
...     print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Note que um espaço foi inserido entre as colunas no primeiro exemplo. É assim que o comando print `print` funciona: ele sempre insere espaço entre seus argumentos.)

Esse exemplo demonstra o método `str.rjust` de objetos string, que alinha uma string à direita juntando espaços adicionais à esquerda. Existem métodos análogas `str.ljust` e `str.center`. Esses métodos não exibem nada na tela, apenas devolvem uma nova string formatada. Se a entrada extrapolar o comprimento especificado, a string original é devolvida sem modificação; isso pode estragar o alinhamento das colunas, mas é melhor do que a alternativa, que seria apresentar um valor mentiroso. (Se for realmente desejável truncar o valor, pode-se usar fatiamento, por exemplo: `x.ljust(n)[:n]`.)

Existe ainda o método `str.zfill` que preenche uma string numérica com zeros à esquerda. Ele sabe lidar com sinais positivos e negativos:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Um uso básico do método `str.format` é assim:

```
>>> print 'Somos os {} que dizem "{}!"'.format('cavaleiros', 'Ni')
Somos os cavaleiros que dizem "Ni!"
```

As chaves e seus conteúdos (chamados de campos de formatação) são substituídos pelos objetos passados para o método `str.format`, respeitando a ordem dos argumentos.

Um número na primeira posição dentro das chaves identifica o argumento pela sua posição na chamada do método (N.d.T. esse número era obrigatório na versão 2.6 do Python; tornou-se opcional na versão 2.7):

```
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

Se argumentos nomeados são passados para o método `str.format`, seus valores pode ser identificados pelo nome do argumento:

```
>>> print 'Este {alimento} é {adjetivo}.'.format(
...       alimento='spam', adjetivo='absolutamente horrível')
Este spam é absolutamente horrível.
```

Argumentos posicionais e nomeados podem ser combinados à vontade:

```
>>> print 'A história de {0}, {1}, e {outro}.'.format('Bill', 'Manfred',
...                                                  outro='Georg')
A história de Bill, Manfred, e Georg.
```

As marcações `'!s'` e `'!r'` podem ser usadas para forçar a conversão de valores aplicando respectivamente as funções `str` e `repr`:

```
>>> import math
>>> print 'O valor de PI é aproximadamente {}.'.format(math.pi)
O valor de PI é aproximadamente 3.14159265359.
>>> print 'O valor de PI é aproximadamente {!r}.'.format(math.pi)
O valor de PI é aproximadamente 3.141592653589793.
```

Após o identificador do campo, uma especificação de formato opcional pode ser colocada depois de `:` (dois pontos). O exemplo abaixo arredonda Pi até a terceira casa após o ponto decimal.

```
>>> import math
>>> print 'O valor de PI é aproximadamente {0:.3f}.'.format(math.pi)
O valor de PI é aproximadamente 3.142.
```

Colocar um inteiro *n* logo após o `:` fará o campo ocupar uma largura mínima de *n* caracteres. Isto é útil para organizar tabelas.

```
>>> tabela = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for nome, ramal in tabela.items():
...     print '{0:10} ==> {1:10d}'.format(nome, ramal)
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

Se você tem uma string de formatação muito longa que não deseja quebrar, pode ser bom referir-se aos valores a serem formatados por nome em vez de posição. Isto pode ser feito passando um dicionário usando colchetes `[]` para acessar as chaves:

```
>>> tabela = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print ('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...        'Dcab: {0[Dcab]:d}'.format(tabela))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Isto também pode ser feito passando o dicionário como argumentos nomeados, usando a notação `**`:

```
>>> tabela = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab:
{Dcab:d}'.format(**tabela)
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Isto é particularmente útil em conjunto com a função embutida `vars`, que devolve um dicionário contendo todas as variáveis locais.

Para uma visão completa da formatação de strings com `str.format`, veja a seção Format String Syntax na Referência da Biblioteca Python.

Formatação de strings com `%`

O operador `%` também pode ser usado para formatação de strings. Ele interpreta o operando da esquerda de forma semelhante à função `sprintf` da linguagem C, aplicando a formatação ao operando da direita, e devolvendo a string resultante. Por exemplo:

```
>>> import math
>>> print 'O valor de PI é aproximadamente %5.3f.' % math.pi
O valor de PI é aproximadamente 3.142.
```

Como o método `str.format` é bem novo (apareceu no Python 2.6), muito código Python ainda usa o operador `%`. Porém, como esta formatação antiga será um dia removida da linguagem, `str.format` deve ser usado.

Mais informações podem ser encontradas na seção String Formatting Operations da Referência da Biblioteca Python.

Leitura e escrita de arquivos

A função `open` devolve um objeto arquivo, e é frequentemente usada com dois argumentos: `open(nome_do_arquivo, modo)`.

```
>>> f = open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

O primeiro argumento é uma string contendo o nome do arquivo. O segundo argumento é outra string contendo alguns caracteres que descrevem o modo como o arquivo será usado. O parâmetro `mode` pode ser `'r'` quando o arquivo será apenas lido, `'w'` para escrever (se o arquivo já existir seu conteúdo prévio será apagado), e `'a'` para abrir o arquivo para adição; qualquer escrita será adicionada ao final do arquivo. A opção `'r+'` abre o arquivo tanto para leitura como para escrita. O parâmetro `mode` é opcional, em caso de omissão será assumido `'r'`.

No Windows, `'b'` adicionado a string de modo indica que o arquivo será aberto em modo binário. Sendo assim, existem os modos compostos : 'rb', 'wb', e 'r+b'. O Windows faz distinção entre arquivos texto e binários: os caracteres terminadores de linha em arquivos texto são alterados ao ler e escrever. Essa mudança automática é útil em arquivos de texto ASCII, mas corrompe arquivos binários como 'JPEG' ou 'EXE'. Seja cuidadoso e use sempre o modo binário ao manipular tais arquivos. No Unix, não faz diferença colocar um `'b'` no modo, então você pode usar isto sempre que quiser lidar com arquivos binários de forma independente da plataforma.

N.d.T. Para ler arquivos de texto contendo acentuação e outros caracteres não-ASCII, a melhor prática atualmente é usar a `codecs.open`, do módulo `codecs`, em vez da função embutida `open`. O motivo é que `codecs.open` permite especificar a codificação logo ao abrir o arquivo. Desta forma, a leitura do arquivo sempre devolverá objetos `unicode`, independente da codificação interna do mesmo. E ao escrever em um arquivo aberto via `codecs.open`, basta enviar sempre strings `unicode`, pois a conversão para o encoding do arquivo será feita automaticamente.

Métodos de objetos arquivo

Para simplificar, o resto dos exemplos nesta seção assumem que um objeto arquivo chamado `f` já foi criado.

Para ler o conteúdo de um arquivo, invoque `f.read(size)`, que lê um punhado de dados devolvendo-os como uma string de bytes `str`. O argumento numérico *size* é opcional. Quando *size* é omitido ou negativo, todo o conteúdo do arquivo é lido e devolvido; se o arquivo é duas vezes maior que memória da máquina, o problema é seu. Caso contrário, no máximo *size* bytes serão lidos e devolvidos. Se o fim do arquivo for atingido, `f.read()` devolve uma string vazia (`""`).

```
>>> f.read()
'Texto completo do arquivo.\n'
>>> f.read()
''
```

O método `f.readline()` lê uma única linha do arquivo; o caractere de quebra de linha (`'\n'`) é mantido ao final da string, só não ocorrendo na última linha do arquivo, se ela não termina com uma quebra de linha. Isso elimina a ambiguidade do valor devolvido; se `f.readline()` devolver uma string vazia, então é certo que arquivo acabou. Linhas em branco são representadas por um `'\n'` -- uma string contendo apenas o terminador de linha.

```
>>> f.readline()
'Primeira linha do arquivo.\n'
>>> f.readline()
'Segunda linha do arquivo.\n'
>>> f.readline()
''
```

O método `f.readlines()` devolve uma lista contendo todas as linhas do arquivo. Se for fornecido o parâmetro opcional *sizehint*, será lida a quantidade especificada de bytes e mais o suficiente para completar uma linha. Frequentemente, isso é usado para ler arquivos muito grandes por linhas, sem ter que ler todo o arquivo para a memória de uma só vez. Apenas linhas completas serão devolvidas.

```
>>> f.readlines()
['Primeira linha do arquivo.\n', 'Segunda linha do arquivo.\n']
```

Uma maneira alternativa de ler linhas do arquivo é iterar diretamente pelo objeto arquivo. É eficiente, rápido e resulta em código mais simples:

```
>>> for line in f:
        print line,

Primeira linha do arquivo.
Segunda linha do arquivo.
```

Essa alternativa é mais simples, mas não oferece tanto controle. Como as duas maneiras gerenciam o buffer do arquivo de modo diferente, elas não devem ser misturadas.

O método `f.write(string)` escreve o conteúdo da string de bytes para o arquivo, devolvendo `None`.

```
>>> f.write('Isto é um teste.\n')
```

N.d.T. Neste exemplo, a quantidade de bytes que será escrita no arquivo vai depender do encoding usado no console do Python. Por exemplo, no encoding UTF-8, a string acima tem 18 bytes, incluindo a quebra de linha, porque são necessários dois bytes para representar o caractere `'é'`. Mas no encoding CP1252 (comum em Windows no Brasil), a mesma string tem 17 bytes. O método `f.write` apenas escreve bytes; o que eles representam você decide.

Ao escrever algo que não seja uma string de bytes, é necessário converter antes:

```
>>> valor = ('a resposta', 42)
>>> s = str(valor)
>>> f.write(s)
```

N.d.T. Em particular, se você abriu um arquivo `f` com a função embutida `open`, e deseja escrever uma string Unicode `x` usando `f.write`, deverá usar o método `unicode.encode()` explicitamente para converter `x` do tipo `unicode` para uma string de bytes `str`, deste modo: `f.write(x.encode('utf-8'))`. Por outro lado, se abriu um arquivo `f2` com `codecs.open`, pode usar `f2.write(x)` diretamente, pois a conversão de `x` -- de `unicode` para o encoding do arquivo -- será feita automaticamente.

O método `f.tell()` devolve um inteiro `long` que indica a posição atual de leitura ou escrita no arquivo, medida em bytes desde o início do arquivo. Para mudar a posição utilize

`f.seek(offset, de_onde)`. A nova posição é computada pela soma do deslocamento *offset* a um ponto de referência especificado pelo argumento *de_onde*. Se o valor de *de_onde* é 0, a referência é o início do arquivo, 1 refere-se à posição atual, e 2 refere-se ao fim do arquivo. Este argumento pode ser omitido; o valor default é 0.

```
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Vai para o sexto byte do arquivo
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Vai para terceiro byte antes do fim
>>> f.read(1)
'd'
```

Quando acabar de utilizar o arquivo, invoque `f.close()` para fechá-lo e liberar recursos do sistema (buffers, descritores de arquivo etc.). Qualquer tentativa de acesso ao arquivo depois dele ter sido fechado resultará em falha.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

É uma boa prática usar o comando `with` ao lidar com objetos arquivo. Isto tem a vantagem de garantir que o arquivo seja fechado quando a execução sair do bloco dentro do `with`, mesmo que uma exceção tenha sido levantada. É também muito mais sucinto do que escrever os blocos `try-finally` necessários para garantir que isso aconteça.

```
>>> with open('/tmp/workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

Objetos arquivo têm métodos adicionais, como `~file.isatty` e `~file.truncate` que são usados com menos frequência; consulte a Referência da Biblioteca Python para mais informações.

O módulo `pickle`

Strings podem ser facilmente escritas e lidas de um arquivo. Números exigem um pouco mais de esforço, uma vez que o método `read` só devolve strings, obrigando o uso de uma função como `int` para produzir o número 123 a partir da string `'123'` . Entretanto, quando estruturas de dados mais complexas (listas, dicionários, instâncias de classe,etc) estão envolvidas, o processo se torna bem mais complicado.

Para não obrigar os usuários a escrever e depurar constantemente código para salvar estruturas de dados, Python oferece o módulo padrão `pickle`. Este é um módulo incrível que permite converter praticamente qualquer objeto Python (até mesmo certas formas de código!) para uma string de bytes. Este processo é denominado pickling (N.d.T. literalmente, "colocar em conserva", como picles de pepinos em conserva). E unpickling é o processo reverso: reconstruir o objeto a partir de sua representação como string de bytes. Enquanto estiver representado como uma string,

o objeto pode ser facilmente armazenado em um arquivo ou banco de dados, ou transferido pela rede para uma outra máquina.

Se você possui um objeto qualquer `x`, e um objeto arquivo `f` que foi aberto para escrita, a maneira mais simples de utilizar este módulo é:

```
pickle.dump(x, f)
```

Para reconstruir o objeto `x`, sendo que `f` agora é um arquivo aberto para leitura:

```
x = pickle.load(f)
```

(Existem outras variações desse processo, úteis quando se precisa aplicar sobre muitos objetos ou o destino da representação string não é um arquivo; consulte a documentação do módulo `pickle` na Referência da Biblioteca Python.)

O módulo `pickle` é a forma padrão da fazer objetos Python que possam ser compartilhados entre diferentes programas Python, ou pelo mesmo programa em diferentes sessões de execução; o termo técnico para isso é **objeto persistente**. Justamente porque o módulo `pickle` é amplamente utilizado, vários autores que escrevem extensões para Python tomam o cuidado de garantir que novos tipos de dados, como matrizes numéricas, sejam compatíveis com esse processo.