

Capítulo 0

Porque aprender a programar?

A razão fundamental, que nos motivou a produzir este tutorial, é uma só: programar é divertido. É claro que nem todo mundo vai concordar com isso, assim como muita gente não acha graça nenhuma em jogar xadrez ou explorar cavernas. Ao contrário do xadrez e da espeleologia, a programação também é uma habilidade que tem forte demanda no mercado de trabalho.

Mas a dura realidade é que somente com centenas ou milhares de horas de experiência programando é que você estará apto a incluir essa disciplina em seu currículo e se dar bem em uma entrevista de emprego. Portanto nosso objetivo aqui é bem mais modesto do que transformar você em um programador profissional. O que estamos te oferecendo é uma introdução suave a esse tópico fascinante. Ao final, esperamos que você descubra se tem a vocação e a motivação necessárias para criar softwares, como hobby ou profissão.

Antes de começar, apenas um aviso: o prazer de construir um programa pode causar dependência psicológica. Não é apenas por dinheiro que programadores do mundo inteiro varam noites escrevendo código.

Material necessário

Para acompanhar esse tutorial você precisará de um computador qualquer onde tenha instalado um interpretador da linguagem Python. Em quase todas as versões modernas de Linux e Mac OS X o interpretador Python já vem instalado (experimente abrir um console e digitar "python"). Na página <http://www.python.org/download/> você encontra links para baixar o interpretador adequado para o seu computador. No caso do Windows, o instalador .msi da versão atual do Python tem cerca de 11 MB.

Porquê Python

Toda programação de computadores é feita através de uma ou mais linguagens de programação, portanto não é possível aprender a programar sem aprender ao menos uma linguagem de programação. Nossa meta não é mostrar como se programa em uma linguagem específica, mas sim como se programa de uma forma geral. Ou seja, a linguagem para nós será um veículo, e não o destino. Mesmo assim, pensamos bastante antes de escolher uma linguagem para este tutorial, e optamos por Python.

Centenas de linguagens já foram criadas desde que o computador eletrônico foi inventado nos anos 40. Algumas já são línguas mortas. Outras, como C++ e Java, são peças fundamentais no desenvolvimento da economia digital. No entanto, a complexidade dessas duas linguagens nos motivou a descartá-las, e focalizar o universo das chamadas linguagens de "*scripting*", que são mais simples e se prestam a um estilo de programação exploratória, mais sintonizado com um tutorial como esse.

As três linguagens de *scripting* mais populares atualmente são !PHP, JavaScript e !VBScript. Todas são utilizadas na construção de *web-sites* dinâmicos, mas praticamente não têm aplicação fora desse domínio, e por isso foram descartadas. É que, embora seja nosso objetivo abordar também esse tópico, achamos que é complexo demais para começar, especialmente devido à dificuldade de se diagnosticar erros de programação em páginas dinâmicas.

Escolhemos Python porque é uma linguagem muito versátil, usada não só no desenvolvimento Web

mas em muitos outros tipos de aplicação. Python roda nos servidores de mega-sites como Google e [YouTube](#), nos clusters de computação gráfica da Industrial Light & Magic, em laboratórios da NASA e da farmacêutica [AstraZeneca](#), e em games como Civilization IV e EVE-Online. O nome "Python" é uma homenagem ao grupo humorístico inglês Monty Python, adorado por *geeks* de todo o mundo, mas pela pequena amostra de usuários citados, não é uma linguagem de brinquedo.

Apesar de sua sintaxe simples e clara, Python oferece os seguintes recursos disponíveis também em linguagens mais complicadas como Java e C++:

- programação orientada a objetos (incluindo herança múltipla, conceito apenas parcialmente presente em Java)
- exceções, um moderno mecanismo para o tratamento de erros
- módulos, uma forma inteligente de acessar e organizar código a ser reutilizado
- coleta de lixo automática, sistema que elimina os erros causados pelo acúmulo de dados inúteis na memória do computador (característica presente também em Java, mas não em C++)
- recursos avançados de manipulação de textos, listas e outras estruturas de dados
- possibilidade de executar o mesmo programa sem modificações em várias plataformas de *hardware* e sistemas operacionais (difícil de se conseguir em C++)

Em resumo, Python nos oferece uma sintaxe tão simples quanto PHP ou VBScript, mas é mais versátil do que elas. E permite explorar vários recursos de Java e C++ de uma forma mais acessível. Por esses motivos acreditamos que seja a melhor escolha para quem quer começar a programar hoje.

Capítulo 1

Abrindo e fechando o interpretador

A melhor forma de aprender a programar é usando um interpretador em modo interativo. Dessa forma você pode digitar comandos linha por linha e observar a cada passo como o computador interpreta e executa esses comandos. Para fazer isso em Python, há duas maneiras:

1-executar o interpretador em modo texto (chamado "Python (command line)" no Windows, ou simplesmente `python` no Linux) 2-usar o IDLE, que é um ambiente baseado em janelas.

Se você usa Windows, escolha o IDLE para começar a acompanhar esse tutorial. O IDLE também está disponível para a plataforma Linux (algumas distribuições colocam o IDLE em um pacote separado do pacote do Python).

Seja qual for o interpretador que você escolheu, ao executá-lo você verá uma mensagem com informações de *copyright* mais ou menos como essa:

```
Python 2.5.1 (r251:54863, Oct 5 2007, 13:50:07)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

O símbolo ">>>" exibido pelo interpretador é o que os americanos chamam de "*prompt*", que alguns traduzem por "aviso", mas nós vamos chamar de "deixa" (em teatro, o termo "*prompt*" é a deixa que indica ao ator a hora de dizer ou fazer algo; em computação, o *prompt* informa o usuário que o sistema está pronto para receber um novo comando).

Para sair do interpretador você pode fechar a janela do IDLE, ou teclar [CTRL]+[D] (no IDLE ou no interpretador em UNIX) ou [CTRL]+[Z] e então [ENTER] (no interpretador DOS).

Uma calculadora melhor

Vamos então aproveitar a deixa e escrever algo. Experimente escrever uma expressão aritmética bem simples, como $2+2$:

```
>>> 2+2
4
>>>
```

A resposta é reconfortante: para Python, $2+2$ é igual a 4. Você pode experimentar outras expressões mais complexas, mas é bom saber que os quatro operadores básicos em Python (e em quase todas as linguagens modernas) são esses:

- + adição
- - subtração
- * multiplicação
- / divisão

Em Python, assim como na linguagem C, os números inteiros têm um tratamento especial. Isso fica evidente quando fazemos uma divisão:

```
>>> 7/2
3
>>>
```

Em vez de 3,5, o resultado foi 3. Isso acontece sempre que todos os números de uma expressão são inteiros. Neste caso, Python imagina que se deseja um resultado inteiro também (esse comportamento estranho às vezes é conveniente em programação).

Se você quiser operar com números decimais, deve usar o ponto e não a vírgula como separador decimal:

```
>>> 7.0/2
3.5
>>> 7/2.0
3.5
>>> 7/2.
3.5
>>>
```

Note que basta digitar um ponto após o número. O computador não consegue lidar com números do conjunto dos reais, mas apenas com uma aproximação chamada "número de ponto-flutuante" (porque o ponto decimal pode aparecer em qualquer posição do número). Ao lidar com ponto-flutuante, às vezes vemos resultados estranhos:

```
>>> 2.4 * 2
4.7999999999999998
>>>
```

O resultado não deveria ser 4.8? Deveria, mas antes de ficar revoltado note que a diferença foi muito pequena. Acontece que o sistema de "ponto-flutuante" padrão IEEE-754 usado em quase todos os computadores atuais tem uma precisão limitada, e Python não esconde este fato de você, programador. O problema não está na conta, mas na própria representação interna do valor 2.4 :

```
>>> 2.4
2.3999999999999999
```

Para exibir valores de ponto-flutuante para um usuário sem assustá-lo, use o comando `print`:

```
>>> print 2.4 * 2
4.8
>>>
```

Você pode digitar espaços entre os números e operadores para fazer uma expressão longa ficar mais legível. Veja esse exemplo:

```
>>> 1 + 2 * 3
7
>>>
```

Note que o interpretador Python é mais esperto que uma calculadora comum. Ele sabe que a multiplicação deve ser efetuada antes da adição. Se você teclar a mesma expressão em uma calculadora qualquer obterá o resultado 9, que é incorreto. Em Python, se você realmente deseja efetuar a soma antes da multiplicação, precisa usar parênteses:

```
>>> (1 + 2) * 3
9
>>>
```

Ao contrário do que você aprendeu na escola, aqui os símbolos `[]` e `{ }` não servem para agrupar expressões dentro de outras expressões. Apenas parênteses são usados:

```
>>> ( 9 - ( 1 + 2 ) ) / 3.0
2.0
>>> ( 9 - 1 + 2 ) / 3.0
3.33333333333
>>>
```

DICA: Se você escrever algo que o interpretador não reconhece, verá na tela uma mensagem de erro. Não crie o mau hábito de ignorar essas mensagens, mesmo que elas pareçam difíceis de entender num primeiro momento. A única vantagem de cometer erros é aprender com eles, e se a preguiça o impedir de ler as mensagens, seu aprendizado será bem mais lento.

Veja aqui como decifrar as mensagens de erro do Python.

Como ler uma mensagem de erro

A dura realidade é que um programador profissional passa boa parte de sua vida caçando erros, e por isso é fundamental saber extrair o máximo de informações das mensagens resultantes.

A essa altura você talvez já tenha provocado um erro para ver o que acontece. Vamos fazer isso agora, e aprender a ler as mensagens resultantes. Pode parecer perda de tempo, mas é importantíssimo saber interpretar as mensagens de erro porque a melhor forma de aprender a programar é experimentando, e ao experimentar você certamente vai provocar muitos erros.

Como exemplo, vamos digitar uma expressão aritmética sem sentido:

```
>>> 7 + / 2
      File "<stdin>", line 1
        7 + / 2
           ^
SyntaxError: invalid syntax
>>>
```

O interpretador indica o local de erro em vermelho no IDLE, ou com o sinal `^` no console. Nos dois casos a última linha contém as informações mais importantes: `SyntaxError: invalid`

syntax. A primeira parte, **SyntaxError** é o tipo do erro, e após o sinal de ":" vem a descrição: erro de sintaxe inválida.

No console a primeira linha da mensagem de erro indica em a linha do seu código onde ocorreu o problema. No modo interativo essa informação pouco útil, mas quando fizermos programas extensos será muito bom saber exatamente em qual linha está a falha. Agora vamos provocar um outro tipo de erro:

```
>>> 1.5/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division
>>>
```

Novamente, a parte mais importante é a última linha, que nesse caso é bem fácil de entender: **ZeroDivisionError: float division**, ou "erro de divisão por zero em divisão de ponto-flutuante".

Conversor de dólares

Digamos que você tem uma loja de discos importados, e precisa constantemente converter dólares em reais. O valor do dólar para venda em 20/05/1999 é de 1.686. Para converter US\$9,95 e US\$11,95 em reais você pode digitar:

```
>>> 9.95 * 1.686
16.775699999999997
>>> 11.95 * 1.686
20.147699999999997
>>>
```

Mas há uma forma melhor: em vez de digitar o valor 1.686 o tempo todo, você pode armazenar esse valor na memória do computador, assim:

```
>>> d = 1.686
>>>
```

Note que o interpretador não respondeu nada (a menos que você tenha cometido um erro), mas ele guardou o número em uma posição de sua memória, e associou o símbolo "d" a essa posição. Agora, fica mais confortável converter dólares em reais:

```
>>> 9.85 * d
16.607099999999999
>>> 11.95 * d
20.147699999999997
>>> 5 * d, 7 * d, 9 * d
(8.429999999999997, 11.802, 15.173999999999999)
>>>
```

No último caso, convertemos de uma vez só os valores 5, 7 e 9 em dólares. Para um resultado mais apresentável, use o comando **print**:

```
>>> print 5 * d, 7 * d, 9 * d
8.43 11.802 15.174
>>>
```

E se a cotação do dólar mudou para 1.61? Basta armazenar o novo número e refazer os cálculos:

```
>>> d = 1.61
>>> print 5 * d, 7 * d, 9 * d
8.05 11.27 14.49
>>>
```

Você precisa digitar a linha mais longa de novo. No IDLE, clique sobre a linha que digitamos no exemplo anterior e tecle [ENTER]. A linha será reproduzida na última deixo, e bastará um novo [ENTER] para processá-la. No console, teclando a seta para cima você acessa o histórico de comandos.

Tabela de preços em dólares e reais

Agora vamos mostrar como o interpretador Python é muito mais poderoso que uma calculadora. Imagine que em sua loja de discos importados você tem um balcão de ofertas com discos de \$4 até \$9. Se quisesse fazer uma tabela de preços em reais você poderia digitar:

```
>>> print 4*d, 5*d, 6*d, 7*d, 9*d
6.44 8.05 9.66 11.27 14.49
>>>
```

Mas isso é um tanto chato e repetitivo. Em programação, sempre que você fizer algo repetitivo é porque não encontrou ainda a melhor solução. Lidar com séries de números é uma atividade comum, e Python pode ajudar muito nesses casos. Digite o seguinte:

```
>>> lista = [5,6,7,8,9]
>>>
```

Aqui nós criamos uma lista de preços na memória do computador e associamos o nome "lista" a esses dados. Em seguida, digite o seguinte (você terá que teclar [ENTER] duas vezes ao final dessa linha; depois saberá porque).

```
>>> for p in lista: print p * d

8.05
9.66
11.27
12.88
14.49
>>>
```

Aqui nós instruímos o interpretador a fazer os seguintes passos:

- para cada item sucessivo da lista:

- - associe o nome p ao item da vez - exiba o valor de p * d

Agora digamos que você tem discos com valores de 4 a 15 dólares. Você poderia digitar a lista de novo, mas a coisa começa a ficar repetitiva novamente. Há uma forma melhor. A linguagem Python possui uma palavra chamada "range" que serve para gerar faixas de números. Vamos usar essa palavra. Digite:

```
>>> range
<built-in function range>
>>>
```

Quando você digita o nome de uma função sem fornecer dados, Python limita-se a dizer a que se refere o nome. Nesse caso: "built-in function range", ou função embutida range. Isso quer dizer que a palavra range é o nome de uma função, um tipo de comando que produz resultados a partir de dados fornecidos. E trata-se ainda de uma função embutida, ou seja, incluída no próprio interpretador (a maioria das funções da linguagem Python não são embutidas, mas fazem parte de módulos que o programador precisa chamar explicitamente; isso será explicado depois).

Acabamos de dizer que uma função "produz resultados a partir de dados fornecidos", então vamos

fornecer algum dado para ver que resultados a função range produz. Digite "range(5)" e veja o que acontece:

```
>>> range(5)
[0, 1, 2, 3, 4]
>>>
```

Quando apenas um dado N é fornecido, range gera uma lista de N números, de zero até N-1. É um comportamento um pouco estranho, mas útil em programação (o primeiro item de uma série, na maioria das linguagens, é o item número zero; isso será discutido mais profundamente quando aprendermos mais sobre listas).

Agora digamos que eu queira uma sequência a partir de 2, e não zero. Digite:

```
>>> range(2,5)
[2, 3, 4]
>>>
```

Agora para obter a lista de valores de discos podemos digitar:

```
>>> range(4,16)
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>>
```

E usando o comando for, calcular de uma só vez todos os valores convertidos:

```
>>> for p in range(4,16): print p * d

6.44
8.05
9.66
11.27
12.88
14.49
16.1
17.71
19.32
20.93
22.54
24.15
>>>
```

Mas o ideal mesmo era ter os valores em dólares e reais lado a lado. Isso é fácil:

```
>>> for p in range(4,16): print p, p * d
...
4 6.44
5 8.05
6 9.66
7 11.27
8 12.88
9 14.49
10 16.1
11 17.71
12 19.32
13 20.93
14 22.54
15 24.15
>>>
```

Resumindo o que foi feito até aqui, com apenas duas linhas de código em Python, você pode gerar

tabelas de conversão de qualquer tamanho. Experimente:

```
>>> d = 1.686
>>> for p in range(50,150): print p, p * d
```

Parabéns, você acaba de construir seu primeiro programa!

Capítulo 2

Introdução

No final do capítulo anterior digitamos o seguinte programa diretamente no interpretador Python:

```
>>> d = 1.686
>>> for p in range(50,150): print p, p * d
```

O resultado desta sequência de comandos é uma longa lista de números em duas colunas. Sabemos que a primeira coluna da esquerda contém preços em dólar e a outra, em reais. Mas nada na listagem indica isto. Observe esse trecho:

```
95 160.17
96 161.856
97 163.542
98 165.228
99 166.914
100 168.6
101 170.286
102 171.972
103 173.658
104 175.344
105 177.03
```

Aqui podemos observar outras deficiências: as colunas não estão corretamente alinhadas, e os valores em reais aparecem com uma, duas ou três casas decimais. Como se trata de uma tabela de preços, os valores em ambas colunas deveriam ter sempre duas casas decimais. Vamos fazer algumas melhorias em nosso programa gerador de tabelas de preços.

Quatro tipos de dados

Para evitar aquele degrau na segunda coluna entre o 99 e o 100, precisamos fazer um pequeno desvio para começar a aprender a lidar com textos, além de números. Digite **eu** = seguido do seu nome entre aspas:

```
>>> eu = 'Fulano'
```

Você tem que digitar as aspas para evitar um erro. As aspas podem ser 'simples' ou "duplas". Python guardará uma cópia do seu nome na memória do computador, e associará o identificador **eu** a esse dado. Agora basta digitar eu para ver o seu nome.

```
>>> eu
'Fulano'
>>>
```

Antes havíamos criado a variável **d** referindo-se à cotação do dólar, e no capítulo anterior também criamos uma variável chamada **lista**, contendo uma lista de valores. Agora criamos a variável **eu** para se referir ao seu nome. Estes são exemplos de três tipos de dados que Python é capaz de

processar: número de ponto flutuante, lista de valores, e texto.

Você pode saber o tipo de uma variável ou estrutura de dados usando a função `type`. Veja estes exemplos:

```
>>> eu = 'Luciano'
>>> d = 1.902
>>> type(eu)
<type 'str'>
>>> type(d)
<type 'float'>
>>>
```

Python acaba de nos dizer que a variável `eu` refere-se a um objeto do tipo `'str'`, uma abreviatura de "string" (basicamente o computador encara um texto como uma cadeia de caracteres). E a variável `d` aponta para um objeto do tipo `'float'`, ou "número de ponto-flutuante", como já vimos antes.

Vejamos mais alguns tipos de dados:

```
>>> type(1)
<type 'int'>
>>> type(1.)
<type 'float'>
>>> type([1,2,3])
<type 'list'>
>>>
```

Observe que o número 1 não é `'float'`, mas `'int'`. Já o número 1. ("um" seguido de um ponto decimal) é considerado um `'float'`. Como já dissemos no primeiro capítulo, inteiros e *floats* têm tratamento diferente em Python. Uma divisão de inteiros (como 7/2), sempre fornece um resultado inteiro (3, nesse exemplo). O próximo dado testado é uma lista, `[1, 2, 3]`, que Python chama de `'list'`.

Agora, experimente fazer esses dois testes:

```
>>> type(range)
<type 'builtin_function_or_method'>
>>> type(range(3))
<type 'list'>
>>>
```

Ao perguntarmos qual é o tipo associado ao nome `range`, Python responde: `'builtin_function_or_method'`. Também já vimos isso no capítulo anterior: o nome `range` refere-se a uma função embutida no próprio interpretador. No teste seguinte, fornecemos um argumento para a função `range`, e assim produzimos um resultado (neste caso, a lista `[0, 1, 2]`, que foi criada na memória do seu computador, mas não foi exibida). É sobre este resultado que a função `type` foi aplicada, retornando a informação de que se trata de um dado do tipo `'list'`. Ou seja, `range` é uma expressão do tipo `builtin_function_or_method`, mas `range(3)` é uma expressão do tipo `'list'`. Faz sentido? Se não faz, escreva reclamando!

Cada tipo de dados suporta operações diferentes. Faça algumas experiências e analise os resultados:

```
>>> n1 = 10
>>> n2 = 20
>>> n1 + n2
30
>>> n1 = 'abacate'
>>> n2 = 'banana'
>>> n1 + n2
'abacatebanana'
```

```
>>> n2 + n1
'bananaabacate'
>>>
```

Por exemplo, o operador `+` realiza uma soma quando aplicado a dados numéricos, mas quando aplicado a dados do tipo *string*, o sinal `+` faz uma operação de concatenação (junção de duas seqüências). Agora experimente isto:

```
>>> x = 3.
>>> x * 5
15.0
>>> 'x' * 5
'xxxxx'
>>>
```

Note que `x` e `'x'` são coisas totalmente diferentes. `x` é o nome de uma variável que neste momento se refere ao valor `3.` (um *float*). O resultado de `x * 5` é `15.0` (outro *float*, como era de se esperar). Já `'x'` é uma *string* com um caractere. Quando o sinal `*` é aplicado entre uma *string* e um número inteiro, Python realiza uma operação de repetição. Como você pode notar, os operadores `+` e `*` fazem coisas diferentes dependendo dos tipos de dados fornecidos na expressão.

É um prazer trabalhar com Python porque se trata de uma linguagem muito coerente. Observe:

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> '12' * 3
'121212'
>>> [1,2] * 3
[1, 2, 1, 2, 1, 2]
>>>
```

No primeiro exemplo, vemos o operador `+` concatenando duas listas. Os outros dois exemplos mostram a operação de repetição. Note que `'12'` não é um número, mas uma *string* composta pelos caracteres `'1'` e `'2'`. Para Python, *strings* e listas têm muito em comum: ambas são seqüências de itens. Enquanto *strings* são seqüências de caracteres, listas são seqüências de itens quaisquer. Nos dois casos, concatenação e repetição funcionam de forma logicamente idêntica.

Enfeitando a tabela

Agora que sabemos sobre alguns tipos de dados, e que os operadores funcionam de forma diferente conforme os dados da expressão, estamos prontos para aperfeiçoar nosso gerador de tabelas usando o poderoso operador `'%'`, que em Python não tem nada a ver com porcentagens. Para ver como ele funciona, vamos criar uma *string* como esta:

```
>>> msg = 'um dólar vale %f real.'
>>>
```

Agora vamos ver o que acontece quando chamamos a variável `msg`:

```
>>> msg
'um d\363lar vale %f real.'
>>>
```

Python representa variáveis *string* dessa forma: entre aspas simples, e trocando os acentos por códigos especiais (estamos falando do código ASCII em notação octal, algo que explicaremos depois). Se você quiser exibir o conteúdo de `msg` de forma mais apresentável, use o comando `print`:

```
>>> print msg
um dólar vale %f real.
>>>
```

OK, é hora de explicar porque colocamos esse estranho %f dentro da mensagem. Trata-se de um marcador de posição para sinalizar onde Python deverá inserir um número quando quisermos imprimir a mensagem com o valor da cotação. Experimente digitar o seguinte:

```
>>> d = 1.902
>>> print msg % d
um dólar vale 1.902000 real.
>>>
```

Veja o que aconteceu: Python substituiu a marca %f pelo valor da variável d. É assim que funciona: a partir de uma *string* com marcas de posição e um ou mais valores, o operador % produz uma nova *string* com os valores inseridos nas respectivas posições. Veja agora um exemplo com dois valores:

```
>>> msg2 = 'Um dólar vale %f real e um real vale %f dólar.'
>>> print msg2 % (d, 1/d)
Um dólar vale 1.902000 real e um real vale 0.525762 dólar.
>>>
```

Note que os valores d e 1/d estão entre parênteses. Isso é obrigatório quando queremos passar mais de um valor para o operador % (uma sequência de valores entre parênteses é um "tuplo", um tipo especial de lista que explicaremos em um outro capítulo).

O símbolo %f serve para informar a Python que o valor a ser inserido naquela posição é um *float*. Se você quiser limitar o número de casas após o ponto decimal, basta usar um formato como esse:

```
>>> d = 1.685
>>> '%.2f' % d
'1.69'
>>>
```

Após o marcador %, a indicação .2 determina que devem aparecer duas casas decimais após o ponto. Note que o resultado é arredondado: 1.685 virou 1.69. Vamos usar esse recurso na nossa tabela:

```
>>> for p in range(4,16): print 'US$ %.2f = R$ %.2f' % (p,p*d)
```

```
US$ 4.00 = R$ 6.74
US$ 5.00 = R$ 8.43
US$ 6.00 = R$ 10.12
US$ 7.00 = R$ 11.80
US$ 8.00 = R$ 13.49
US$ 9.00 = R$ 15.17
US$ 10.00 = R$ 16.86
US$ 11.00 = R$ 18.55
US$ 12.00 = R$ 20.23
US$ 13.00 = R$ 21.92
US$ 14.00 = R$ 23.60
US$ 15.00 = R$ 25.29
>>>
```

Está quase linda. Faltava só consertar o degrau que acontece entre a linha do 9 e do 10. No marcador de posição você também pode colocar um número à esquerda do ponto para definir a largura total

do espaço que será reservado. Na faixa de preços de 4 a 15, os maiores valores tem cinco caracteres de comprimento (incluindo o ponto decimal), por isso vamos usar '%5.2f'. Agora podemos fazer uma versão bem melhor da tabela:

```
>>> for p in range(4,16): print 'US$ %5.2f = R$ %5.2f' % (p,p*d)
```

```
US$  4.00 = R$  6.74
US$  5.00 = R$  8.43
US$  6.00 = R$ 10.12
US$  7.00 = R$ 11.80
US$  8.00 = R$ 13.49
US$  9.00 = R$ 15.17
US$ 10.00 = R$ 16.86
US$ 11.00 = R$ 18.55
US$ 12.00 = R$ 20.23
US$ 13.00 = R$ 21.92
US$ 14.00 = R$ 23.60
US$ 15.00 = R$ 25.29
>>>
```

Entendendo melhor o for

Como você percebeu, no comando `for` tudo aquilo que aparece após os sinais ":" é repetido várias vezes, uma vez para cada item da lista de valores indicada após a palavra `in`. Mas os comandos a serem repetidos podem ser vários, e na maioria das vezes não são escritos na mesma linha que o `for`, como temos feito, mas sim em linhas subsequentes.

O comando `for` é algo que chamamos de "estrutura de controle", que serve para determinar a forma de execução de um comando ou de uma sequência de comandos, às vezes chamada de um "bloco". Em outras linguagens, os blocos são delimitados por marcadores especiais. Java, Perl e C++ usam os sinais { e } para este fim. Pascal e Delphi usam as palavras **BEGIN** e **END**. Além desses marcadores exigidos pelas linguagens, os programadores usam também o recurso da endentação, ou seja, o recuo em relação à margem esquerda, para tornar mais fácil a visualização da estrutura do programa. Veja este exemplo em Perl:

```
for ($i = 0; $i < 5; $i++) {
    $v = $i * 3;
    print "$v\n";
}
```

 Atenção: isto é Perl, e não Python.

Aqui, os comandos `$v = $i * 3;` e `print "$v\n";` formam o bloco que está sobre o controle do comando `for`, ou seja, os dois comandos serão executados repetidamente. O programa equivalente em Python é escrito assim:

```
>>> for i in range(5):
...     v = i * 3
...     print v
```

Em nossa opinião, o código em Python é bem mais legível. Para sinalizar quais comandos fazem parte do bloco que está sob o controle do `for`, apenas a endentação é utilizada. Se você está usando o IDLE, esse recuo acontece automaticamente quando uma linha de comando termina com o sinal ':', que em Python sempre indica o início de um bloco. No interpretador Python invocado a partir da linha de comando no DOS ou em UNIX, a endentação não é automática. Você precisa digitar ao menos um espaço em branco para evitar uma mensagem de erro como essa:

```
>>> for i in range(5):
...     print i
      File "", line 2
        print i
        ^
```

SyntaxError: invalid syntax

Note que o interpretador está reclamando de sintaxe inválida, e apontando (^) para a primeira palavra do bloco que deveria estar recuado. Veja a mesma coisa, com a segunda linha recuada com a tecla [TAB]:

```
>>> for i in range(5):
...     print i
...
0
1
2
3
4
>>>
```

Já deve ter ficado claro porque era preciso teclar [ENTER] duas vezes depois do `for` nos exemplos anteriores: é que, no modo interativo, o interpretador Python espera uma linha em branco para sinalizar o final de uma série de comandos que formam um bloco dentro de uma estrutura de controle.

Agora que entendemos o conceito de bloco, podemos enfeitar ainda mais a nossa tabela colocando um segundo comando `print` dentro do nosso `for`:

Veja este exemplo:

```
>>> for p in range(9,13):
...     print 'US$ %5.2f = R$ %5.2f' % (p, p * d)
...     print '-' * 20
...
US$  9.00 = R$ 15.17
-----
US$ 10.00 = R$ 16.85
-----
US$ 11.00 = R$ 18.54
-----
US$ 12.00 = R$ 20.22
-----
>>>
```

A outra face do %

Antes de encerrar este capítulo, vale a pena contar que, assim como o `+` e o `*`, o operador `%` também tem dupla personalidade. Quando aplicado sobre dois números, que podem ser inteiros ou *floats*, o `%` retorna o resto da divisão inteira do primeiro pelo segundo. Veja só:

```
>>> 6 % 3
0
>>> 7 % 3
1
>>> 8 % 3
2
>>> 9 % 3
0
>>>
```

Explicando: $6 / 3$ dá 2, e o resto é 0; a divisão inteira de $7 / 3$ também dá 2, mas o resto é 1. Esta operação é chamada de "modulo" em inglês. Sua principal utilidade é determinar se um número é múltiplo de outro. Nos exemplos acima, o resultado de $6 \% 3$ e $9 \% 3$ é zero, porque 6 e 9 são múltiplos de 3.

No próximo capítulo vamos começar a elaborar programas mais extensos. O modo interativo, que temos usado até agora, vai continuar sendo útil para testarmos novas idéias e observar o comportamento de funções e módulos do Python rapidamente. Mas, a partir da próxima sessão, vamos começar a gravar nossos programas para uso posterior, em vez de digitá-los diretamente no interpretador. E vamos também descobrir como solicitar informações do usuário, de forma que os programas possam ser utilizados por pessoas que não sabem programar e preferem ficar longe de um interpretador interativo.

Capítulo 3

Introdução

Depois de dois capítulos bem básicos, é hora de engatar uma segunda e começar a criar programas mais dignos desse nome. Não vamos desprezar o que fizemos até aqui, digitando diretamente na deixa do interpretador Python. Ao contrário: tenha sempre em mente que você pode usá-lo para esclarecer aquela dúvida rápida ou mesmo localizar um bug escondido em um sistema complexo. A maioria das outras linguagens não oferece um ambiente para execução imediata de comandos como o Python. Uma exceção famosa é a linguagem Logo, onde o interpretador interativo serve justamente para facilitar a aprendizagem da programação através da exploração e experimentos.

Mas a partir de agora vamos atacar programas mais extensos, e não vamos querer digitá-los linha por linha no modo interativo. Em vez disso, vamos escrever os comandos em um editor de textos, salvar o arquivo, e mandar o interpretador Python ler o programa salvo.

Rodando programas no IDLE

A versão Windows do Python traz o IDLE, um interpretador interativo em modo gráfico que já apresentamos no primeiro capítulo. Se você não usa essa versão do Python, vá direto para a próxima seção: Testando no sistema. O IDLE inclui um editor de programas simplório, mas útil para quem está aprendendo a linguagem. O editor do IDLE exibe com cores diferentes as palavras da linguagem, de acordo com sua função sintática (lembra da aula de português onde o verbo era verde, o sujeito vermelho etc?). Para abrir o editor, rode o IDLE e acione o comando **File > New window**. A janela que se abrirá, com o título "untitled", é um editor. Experimente digitar um programinha como esse:

Esconder número das linhas

```
1 for i in range(100):  
2     print 'Fulano ',  
3 print 'e seus Sicranos'
```

Note que o editor pinta algumas palavras de laranja. São as chamadas palavras-chave, peças tão importantes em Python como os verbos em português. Outras cores indicam funções e variáveis. E os textos entre aspas aparecem em verde: dessa forma, fica difícil esquecer de fechar aspas. Outra coisa que acontece magicamente é a endentação. O editor "sabe" que após os ":" do comando `for`, deve vir um bloco endentado. Para encerrar o bloco endentado, você pode teclar **[ENTER]** duas vezes para pular uma linha, como ocorre também na deixa do interpretador, ou então teclar

[BackSpace] para apagar de uma vez só os quatro espaços à esquerda da linha.

Uma vez digitado esse programinha você pode executá-lo de duas maneiras: diretamente de dentro do IDLE ou no console do sistema operacional. Para fazer o IDLE rodar o seu programa é só teclar [F5]. Se você ainda não salvou o código do seu programa, o IDLE vai exibir uma mensagem pedindo para que você o faça. Basta usar o comando File > Save, ou melhor ainda, [CTRL]+[S]. Se você não sabe onde salvar, sugiro que crie uma pasta chamada Curso dentro da pasta onde está o seu interpretador Python e salve ali (provavelmente a pasta ficará sendo C:\Python25\Curso, no caso do Python 2.5). Assim fica fácil encontrá-lo depois. Use o nome `egotrip.py`.

O programinha `egotrip.py` faz o nome do autor aparecer 100 vezes, seguindo do nome de sua banda. No tempo dos computadores de 8 bits, programinhas como esse eram invariavelmente os primeiros exercícios de qualquer estudante de programação. No IDLE, a "saída" do programa (aquilo que ele exibe ao rodar), aparece dentro de uma janela intitulada "Python Shell". Você pode fechar essa janela quando o programa parar.

Você talvez tenha notado que o programa é meio lento. Em meu velho *notebook* Pentium 133 o programa levava 10 segundos para escrever as 101 linhas. É muito. Mas a culpa não é do Python, e sim do IDLE, como veremos a seguir.

Navegando pela linha de comando

No Linux a linha de comando está em toda parte, mas no Windows fica um pouco escondida. Para encontrá-la, clique na barra de tarefas do Windows XP em **Iniciar > Executar...** Na janelinha que se abre, digite apenas `cmd`. Isso executa o interpretador de comandos do Windows, equivalente ao velho MS-DOS nos Windows mais antigos. Para quem nunca navegou pelo sistema via prompt, eis aqui o mínimo que você precisa saber. Veja o que aparece na janela do Prompt:

À esquerda do cursor, você tem a informação mais importante para se orientar: a letra do drive e o nome da pasta onde você se encontra. Se o seu Windows está com a configuração de fábrica, você estará em `C:\Windows`. O sinal `>` é apenas a deixa do sistema, equivalente ao `>>>` usado pelo Python para indicar que está pronto para receber um comando. Antes de mais nada, vamos acionar um programinha que nos poupará muita digitação posteriormente.

Agora, vamos ver o que existe na pasta onde estamos (a pasta Windows). Digite:

```
C:\Windows>dir [ENTER]
```

Você verá uma longa listagem de arquivos, com seus nomes abreviados, extensões, tamanhos, datas e nomes longos. Em meu *notebook* aparecem 236 arquivos na pasta Windows. Não estamos interessados neles agora, o objetivo era apenas mostrar que o comando `dir` produz uma listagem dos arquivos da pasta, ou diretório, atual.

Agora vamos navegar até o diretório onde foi gravado o programa `egotrip.py`. Digite:

```
C:\Windows>cd \ [ENTER]
```

Agora você está no chamado diretório raiz do seu disco. Digite `dir` e veja como a maioria dos itens dentro dessa pasta são outras pastas, como a própria pasta Windows. Agora vamos entrar na pasta do Python:

```
C:\>cd pythonXX [ENTER]
```

E, em seguida, na pasta Curso, que você deve ter criado quando salvou o arquivo `egotrip.py`.

```
C:\PythonXX>cd curso  
C:\PythonXX\Curso>dir
```

Você deverá ver uma listagem como essa:

```
0 volume da unidade C é XXX
0 número de série do volume é XXXX-XXXX
Pasta de C:\PythonXX\Curso
```

```
.          <DIR>          25/10/99  20:57 .
..         <DIR>          25/10/99  20:57 ..
EGOTRIP  PY          89  25/10/99  20:32 egotrip.py
          1 arquivo(s)          89 bytes
          2 pasta(s)       21.331.968 bytes disponíveis
```

```
C:\PythonXX\Curso>
```

Agora você está no ponto certo para digitar o comando que causará a execução do seu programa `egotrip.py`.

Testando no sistema

Meu ambiente favorito para rodar programas em Python é a própria linha de comando do sistema operacional. Não costumo usar o editor do IDLE, mas sim o **NotePad++**, um excelente editor de textos livre e gratuito para Windows. Seja qual for o editor que você usa, o importante é salvar o arquivo como texto puro, sem marcas de formatação. O Notepad é melhor que Word para esse fim, mas o [NotePad++](#) é muito melhor. No Linux, Gedit, Kate, Pico, Vi e Emacs são alguns editores de texto puro bastante comuns. Entre esses, prefiro Gedit e Kate, que têm interfaces mais modernas. Uma vez digitado e salvo o arquivo, você precisa executá-lo a partir da linha de comando do seu sistema.

Quem usa Linux ou já está habituado ao DOS, pode seguir até a próxima seção, *ASCII art*.

ASCII art

No Windows, para executar o programa, digite esse encantamento (supondo que você fez tudo conforme descrito na seção acima, ou fez tudo diferente mas sabia o que estava fazendo):

```
C:\PythonXX\Curso>..\python egotrip.py
```

Os sinais `..\` na frente do comando `python` servem para dizer ao DOS para executar um programa que está no diretório anterior no caminho atual. Assim, acionamos o programa `python.exe` que está na pasta `C:\PythonXX`.

No Linux, você precisará chegar até o diretório que contém o exemplo, e digitar:

```
$ python egotrip.py
```

Ou, se isso não funcionar, tente algo como segue (o comando exato vai depender da sua instalação):

```
$ /usr/local/bin/python egotrip.py
$ /usr/bin/python egotrip.py
```

Bom, deu trabalho mas chegamos. E como você deve ter notado, a execução do programinha foi bem mais veloz que no IDLE (em meu computador, menos de 1 segundo, em vez de 10).

Agora vamos fazer uma pequena mudança no programa `egotrip` que terá um grande efeito. Para fazer essa alteração, no Windows o modo mais rápido é segurar a tecla `[ALT]` e pressionar `[TAB]` até que o ícone do editor do IDLE identificado pelo nome do arquivo `egotrip.py` esteja selecionado. Então solte a tecla `[ALT]`, que o editor aparecerá sobrepondo-se às demais janelas.

Agora vamos modificar o programa egotrip. Ao final da segunda linha, digite uma vírgula. O seu programa deverá ficar assim:

Esconder número das linhas

```
1 for i in range(100):  
2     print 'Luciano ',  
3 print 'e seus Camargos'
```

Salve com [CTRL]+[S] e rode o programa novamente. Tecle [F5] para rodar no IDLE, ou siga esses passos para testar no DOS:

- [ALT]+[TAB] até voltar ao *prompt* do DOS
- [^] (seta para cima) para repetir o comando `..\python egotrip.py`
- [ENTER] para executar o comando.

10 entre 10 programadores que usam a plataforma Windows têm muita prática com a sequência [ALT]+[TAB], [^], [ENTER]. Logo, logo, em sua primeira sessão de caça a um bug, você terá oportunidade de praticar bastante.

Nesse caso, é interessante testar o programa tanto no IDLE quanto na linha de comando. Você verá que os resultados são bem diferentes. Experimente e tente explicar porquê.

Como exercício final, substitua o argumento 100 da função range pelo número 1000, e rode o programa novamente (não recomendo usar o [F5] do IDLE dessa vez; será bem demorado). Tente acrescentar ou retirar letras dos nomes. O efeito será diferente. Bem vindo ao mundo da expressão artística com caracteres de computador.

Seu primeiro programa interativo

Até agora, todos os programas que mostramos não são interativos, ou seja, uma vez rodando, eles não aceitam a entrada de dados de um usuário ou do sistema. Programas não interativos são usados em muitas situações comuns. O programa que emite os cheques da folha de pagamentos de uma grande empresa provavelmente não é interativo, mas recebe todos os dados necessários em um único lote, antes de sua execução. Mas os programas mais interessantes, como um processador de textos, um *game* ou o piloto automático de um avião são todos interativos. Esse é o tipo de programa que passaremos a desenvolver agora.

Nosso passeio pela ASCII *art* não teve apenas objetivos estéticos. Quisemos mostrar como rodar um programa em Python a partir da linha de comando porque, a partir de agora, vamos usar um comando da linguagem Python que não funciona na atual versão do IDLE. O comando chama-se "raw_input", e sua função é receber uma entrada de dados do usuário (input quer dizer entrada de dados; cuidado porque você deve ter sido condicionado a acreditar que "antes de P e B sempre vem a letra M", mas input é inglês, e se escreve com N mesmo; eu perdi uma hora com isso quando aprendia BASIC).

Vejam um primeiro exemplo. Observe que não estamos acentuando o texto no programa porque o DOS não reproduz corretamente os acentos do Windows, e precisamos do DOS para testar esse programa. Deve haver uma forma de convencer o DOS a exibir os acentos corretos do Windows, mas ainda não descobrimos como.

De qualquer forma, isso não quer dizer que não dá para fazer programas com acentuação correta em Python; quando aprendermos a criar softwares gráficos esse problema desaparecerá.

Digite o programinha abaixo, salve como `despdom1.py` e execute na linha de comando.

Esconder número das linhas

```
1 # despdom1.py - Calculadora de despesas domesticas
2
3 print 'Banco de despesas domesticas'
4 ana = raw_input('Quanto gastou Ana? ')
5 bia = raw_input('Quanto gastou Bia? ')
6 total = float(ana) + float(bia)
7 print 'Total de gastos = R$ %s.' % total
8 media = total/2
9 print 'Gastos por pessoa = R$ %s.' % media
```

Os números que aparecem à esquerda na listagem acima não fazem parte do programa e não devem ser digitados. Eles estão aí para facilitar a explicação que vem logo a seguir.

Antes de esmiuçar o programa, vale a pena executá-lo para ver o que acontece. Você será solicitado a digitar um valor para Ana e outro para Bia. Note que os valores deverão ser apenas números. Se quiser usar centavos, use o ponto decimal em vez de vírgula, como já vínhamos fazendo antes. E nada de \$ ou R\$. Vejamos um exemplo de execução:

```
C:\PythonXX\Curso>..\python despdom1.py
Banco de despesas domesticas
```

```
Quanto gastou Ana? 10
Quanto gastou Bia? 20
Total de gastos = 30.0
Gastos por pessoa = 15.0
```

```
C:\PythonXX\Curso>
```

Dissecando o código

Agora vamos acompanhar, linha por linha, como o interpretador executou o programa. Essa é a atividade mais importante para desenvolver você como programador ou programadora. Você precisa aprender a ler um programa e simular mentalmente que acontece dentro do computador. "Quando você aprender a se colocar no lugar do computador ao ler um programa, estará pronto, Gafanhoto".

- **Linha 1:** O sinal # indica comentário. Tudo o que aparece em uma linha a partir desse sinal é ignorado pelo interpretador Python. Neste caso, o comentário explica para nós, humanos, o propósito do programa. Note que o comentário não aparece para o usuário final quando o programa é executado. Comentários servem apenas para ser lidos por outros programadores.
- **Linha 3:** O velho comando `print` é usado para escrever o título "Banco de despesas domesticas" na tela do usuário.
- **Linha 4:** O comando `raw_input` exibe a pergunta "Quanto gastou Ana?", aguarda uma resposta e armazena na variável `ana`.
- **Linha 5:** O mesmo comando é usado para guardar os gastos de Bia na variável `bia`.
- **Linha 6:** Aqui é calculado o total. Note o uso da função *float*. Acontece que a função `raw_input` não retorna números, e sim *strings*. Como vimos no capítulo anterior, o operador "+" tem efeitos diferentes quando aplicado a *strings*; em vez de somar, ele concatena ou junta os textos. Nesse caso, se `ana` é "10" e `bia` é "20", `ana + bia` seria "1020". Para realizar a soma, precisamos antes transformar as *strings* em números, o que é feito pela funções *float* ou *int*. Nesse caso, usamos *float* porque não vamos nos limitar a aceitar números inteiros.

- **Linha 7:** O total é exibido, com o auxílio do operador `%` que insere o valor na posição assinalada pelos caracteres `%s` dentro da mensagem. O código `%s` faz com que Python transforme o número em *string*.
- **Linha 8:** Cálculo da média. Como ambos os valores são float, o resultado será preciso (se fossem inteiros, o resultado também seria forçado a ser inteiro, o que nesse caso levaria a erros do tipo).
- **Linha 9:** Mostramos a média, usando a mesma técnica da linha 7.

Experimente rodar o programa algumas vezes. Note que não é um programa muito robusto: se você não digitar coisa alguma e teclar [ENTER] após uma das perguntas, ou responder com letras em vez de números, o programa "quebra". No próximo capítulo aprenderemos a lidar com entradas inesperadas.

Um programa mais esperto

O programa acima é quase útil. Ele calcula a despesa total e a média, mas não responde à pergunta fundamental: quanto Ana tem que pagar a Bia, ou vice-versa? A aritmética envolvida é simples: se Ana gastou menos, ela precisa pagar a Bia um valor igual à diferença entre o que gastou e a média. Gostaríamos que nosso programa funcionasse assim:

Balanco de despesas domesticas

```
Quanto gastou Ana? 10
Quanto gastou Bia? 20
Total de gastos: R$ 30.0
Gastos por pessoa: R$ 15.0
Ana deve pagar: R$ 5.0
```

Utilize o comando **File > Save As...** para salvar o programa `despdom1.py` como `despdom2.py`. Agora vamos modificá-lo para fazer o que queremos. Abaixo, o programa final, e a seguir, a explicação de cada mudança que foi feita.

Esconder número das linhas

```
1 # despdom2.py - Calculadora de despesas domesticas - versao 2
2
3 print 'Balanco de despesas domesticas'
4 ana = float(raw_input('Quanto gastou Ana? '))
5 bia = float(raw_input('Quanto gastou Bia? '))
6 print
7 total = ana + bia
8 print 'Total de gastos: R$ %s' % total
9 media = total/2
10 print 'Gastos por pessoa: R$ %s' % media
11 if ana < media:
12     diferenca = media - ana
13     print 'Ana deve pagar: R$ %s' %diferenca
14 else:
15     diferenca = media - bia
16     print 'Bia deve pagar: R$ %s' % diferenca
```

- **Linha 1:** Acrescentamos "versao 2" ao comentário
- **Linhas 4 e 5:** Aqui fazemos a conversão dos resultados de `raw_input` para float imediatamente, de modo que os valores armazenados na variáveis `ana` e `bia` são números, e não *strings* como antes.

- **Linha 6:** Uma mudança cosmética apenas: acrescentamos uma linha com apenas um print, para deixar na tela uma linha em branco entre as perguntas e os resultados.
- **Linhas 7:** Agora podemos simplesmente somar os valores de ana e bia, que já foram convertidos para float nas linhas 4 e 5.
- **Linhas 8 a 10:** Exibimos o total e processamos a média, como antes.
- **Linha 11:** Apresentamos um novo comando de bloco, o comando if, que pode ser traduzido exatamente como "se". Essa linha diz, literalmente: "se ana < media:". Ou seja, se o valor de Ana for menor que o valor da média, execute o bloco endentado a seguir (linhas 12 e 13). Caso contrário, não execute essas linhas, e passe direto para a linha 14.
- **Linhas 12 e 13:** Calculamos e exibimos quanto Ana deve pagar.
- **Linha 14:** Aqui vemos outro comando de bloco, o else, que pode ser traduzido como "senão". O else só pode existir após um bloco iniciado por if. O bloco que segue o else só é executado quando a condição prevista no if não ocorre. Isso significa que, quando temos um bloco if e um bloco else, é garantido que apenas um dos dois será executado. Nesse caso, as linhas 15 e 16 só serão executadas se o valor de Ana não for menor que a média.
- **Linhas 15 e 16:** Calculamos e exibimos quanto Bia deve pagar.

Experimente um pouco com o programa `despdom2.py`. O que acontece quando os gastos de Ana e Bia são iguais? Tente responder essa pergunta sem rodar o programa. A chave está na linha 11. Qual é a média quando os gastos são iguais? Tente simular mentalmente o comportamento do computador na execução passo a passo do programa. Dedique alguns minutos a esse desafio, e só então rode o programa com valores iguais para ver se acontece o que você imaginou.

Tudo sobre o if

O comando if, que acabamos de conhecer através de um exemplo, é uma peça fundamental da linguagem Python, e de quase todas as linguagens de programação existentes. Sua função é descrita como "comando de execução condicional de bloco", ou seja, é um comando que determina a execução ou não de um bloco de comandos, de acordo com uma condição lógica. No exemplo, a condição lógica é "ana < media". O operador < serve para comparar dois números e determinar se o primeiro é menor que o segundo (ele também funciona com *strings*, mas aí a comparação segue uma regra parecida com a ordem usada dos dicionários). Os operadores de comparação de Python são os mesmos usados em Java e C++:

| Operador | Descrição | Exemplo |
|----------|------------------|---------|
| == | igual | a == b |
| != | diferente | a != b |
| < | menor que | a < b |
| > | maior que | a > b |
| >= | maior ou igual a | a >= b |
| <= | menor ou igual a | a <= b |

Para sentir o funcionamento desses operadores, abra o interpretador interativo do Python e digite esses testes (não vamos mostrar os resultados aqui; faça você mesmo).

```
>>> a = 1
>>> b = 2
>>> a == 1
>>> a == 2
>>> a == b
>>> 2 == b
>>> a != b
>>> a != 1
>>> a < b
>>> a >= b
```

As linhas 1 e 2 não produzem nenhum resultado, como já vimos antes. Elas apenas atribuem valor às variáveis `a` e `b`. A linha 3 parece um pouco com a linha 1, mas significa algo completamente diferente. Aqui não acontece nenhuma atribuição, apenas uma comparação, que vai gerar um resultado. Um erro bastante comum cometido por quem está aprendendo Python, C ou Java é usar `=` no lugar de `==` ao fazer uma comparação (em Basic, por exemplo, o `=` é usado nos dois casos). Após cada uma das linhas a partir da linha 3, o interpretador mostrará um número 1 ou 0, para indicar que a comparação é verdadeira (1) ou falsa (0).

Voltando ao comando `if`, não existe nenhuma lei que obrigue a presença de um operador de comparação na condição do `if`. A única coisa que interessa é que a expressão que estiver no lugar da condição será considerada falsa se for igual a 0 (zero), uma *string* vazia, uma lista vazia ou o valor especial `None`, sobre o qual voltaremos a falar depois. Qualquer valor que não seja um desses será considerado "verdadeiro", e provocará a execução do bloco subordinado ao `if`. É por isso que os operadores de comparação retornam 0 ou 1 para representar falso ou verdadeiro.

Não é obrigatória a presença de um bloco `else` após um `if`. Mas um `else` só pode existir após um `if`. E um `if` pode conter, no máximo, um `else`. Existe um terceiro comando de bloco relacionado a esses, chamado `elif`. Ele corresponde à combinação `else-if` existente em outras linguagens. Assim como o `if`, cada `elif` deve ser acompanhado de uma condição que determinará a execução do bloco subordinado. Como todo comando de bloco, a primeira linha do `elif` deve ser terminada por um sinal de `:`.

Um `if` pode ser seguido de qualquer quantidade de blocos `elif`, e se houver um bloco `else` ele deverá vir depois de todos os `elif`. Veja esse fragmento de código, parte de um jogo simples que criaremos no próximo capítulo:

Esconder número das linhas

```
1 if vf == 0:
2     print 'Alunissagem perfeita!'
3 elif vf <= 2:
4     print 'Alunissagem dentro do padrao.'
5 elif vf <= 10:
6     print 'Alunissagem com avarias leves.'
7 elif vf <= 20:
8     print 'Alunissagem com avarias severas.'
9 else:
10    print 'Modulo lunar destruido no impacto.'
```

Numa sequência de `if/elif/elif/.../else` é garantido que um, e apenas um dos blocos será executado. Fica como desafio para o leitor descobrir como usar o comando `elif` para corrigir o bug dos gastos

iguais, que aparece no programa `despdom2.py`.

Capítulo 4

Orçamentos, pousos lunares e tratamento de erros

O capítulo anterior terminou com uma questão no ar. Após estudarmos todas as formas de se usar o comando `if`, restou o desafio de usar um bloco `elif` para consertar um pequeno defeito no programa `despdom2.py`. O bug se manifesta quando os gastos de Ana e Bia são iguais. Nesse caso, o programa escreve na tela:

Bia deve pagar: R\$ 0.0

Em vez de fazer a Bia escrever um cheque de zero reais, o melhor seria tratar esse caso especial. Veja como fazê-lo, usando uma construção `if/elif/else` (listagem 1). Se você guardou o arquivo `despdom2.py` da lição anterior, terá muito pouco o que digitar. Abra-o e salve com o nome de `despdom3.py`. O código é idêntico à versão anterior até a linha 14. Ali, você faz a primeira alteração: o `else` é substituído por um `elif` que verifica se Bia gastou menos que a média. As linhas 15 e 16 continuam como antes, mas agora elas só serão executadas se `bia < media` for verdadeiro. As linhas 17 e 18 são novas, e servem para tratar o caso em que nem `ana < media` nem `bia < media`, ou seja, quando não há diferença a ser paga. Agora você pode testar o programa digitando valores diferentes e depois valores iguais para as despesas de Ana e Bia.

Esconder número das linhas

```
1 # despdom3.py - Calculadora de despesas domesticas - versao 3
2
3 print 'Balanco de despesas domesticas'
4 ana = float(raw_input('Quanto gastou Ana? '))
5 bia = float(raw_input('Quanto gastou Bia? '))
6 print
7 total = ana + bia
8 print 'Total de gastos: R$ %s' % total
9 media = total/2
10 print 'Gastos por pessoa: R$ %s' % media
11 if ana < media:
12     diferenca = media - ana
13     print 'Ana deve pagar: R$ %s' % diferenca
14 elif bia < media:
15     diferenca = media - bia
16     print 'Bia deve pagar: R$ %s' % diferenca
17 else:
18     print 'Ana e Bia gastaram a mesma Quantia.'
```

Somadora infinita

Logo adiante iremos reescrever o programinha acima para torná-lo mais flexível, permitindo digitar os nomes e os gastos de qualquer número de pessoas. Assim ele será útil para repartir as contas de uma viagem de férias ou daquela festa entre amigos. Para começar, vamos construir um programa um pouco mais simples, capaz de somar uma série de números (listagem 2).

Esconder número das linhas

```
1 # somadora1.py - somadora infinita - versao 1
2
3 print 'Digite os valores a somar seguidos de [ENTER].'
```

```

4 print 'Para encerrar digite zero: 0'
5 n = float(raw_input(':'))
6 total = n
7 while n != 0:
8     n = float(raw_input(':'))
9     total = total + n
10 print 'TOTAL: %s' % total

```

Vamos ver o que faz esse programa, linha por linha.

- **Linhas 3 e 4:** Exibimos as instruções de uso.
- **Linha 5:** Usamos o comando `raw_input()` para exibir o sinal ":" e ler o primeiro valor digitado pelo usuário, e a função `float` para transformar a *string* resultante em um número de ponto flutuante. O resultado é armazenado na variável `n`.
- **Linha 6:** A variável `total` servirá para guardar a soma acumulada. Para começar, colocamos nela o primeiro valor digitado.
- **Linha 7:** Aqui usamos um novo comando de bloco, o `while`. Essa linha pode ser traduzida assim: "enquanto `n` é diferente de zero...". Assim como o comando `for`, o `while` causa a execução repetida do bloco subordinado (linhas 8 e 9). Em um comando `while`, a repetição é condicionada a uma expressão lógica do mesmo tipo que usamos com o comando `if`. Nesse exemplo, a condição `n != 0` causará a repetição do bloco enquanto for verdadeiro que `n` é diferente de 0. No momento que `n` contiver o valor 0, a condição será falsa e a repetição deixará de ocorrer. O programa então seguirá para a linha 10.
- **Linha 10:** Mostramos o total acumulado. Fim do programa.

Mais sobre o while

Os comandos `while` e `for` são semelhantes por causarem a repetição de um bloco. Ambos são chamados, pelos computólogos, de comandos de iteração (iteração é sinônimo de repetição; não confunda com "interação", que é uma ação recíproca entre dois ou mais agentes).

A diferença é que no comando `for` a iteração serve para percorrer uma lista de itens, como fizemos anteriormente quando trabalhamos com tabelas de conversão. No caso do `for`, o número de repetições é sempre conhecido de antemão: o bloco será executado uma vez para cada item da lista. O comando `while` serve para todos os outros casos de iteração, quando o número de repetições é indefinido. Nossa somadora infinita é um exemplo típico: a iteração que solicita valores e os totaliza poderá ser repetida qualquer número de vezes, dependendo apenas da sua vontade.

Agora vamos analisar de perto duas circunstâncias especiais. Rode o programa e digite 0 (zero) como primeiro valor. Nas linhas 5 e 6 o programa armazenará o zero nas variáveis `n` e `total`. A seguir, na linha 7, o comando `while` verificará a condição `n != 0`. Nesse caso, a condição será falsa. Então o bloco subordinado ao `while` não será executado nenhuma vez, e o programa passará direto para a linha 10, mostrando o total.

Outro momento interessante ocorre quando o primeiro valor digitado não é zero, e a iteração é executada. Digamos que o usuário digitou [1][Enter], [2][Enter] e [0][Enter]. O zero digitado pelo usuário será lido e armazenado em `n` na linha 8, como já vimos. Na linha 9 o valor de `n` é somado ao total. Nessa iteração o valor de `n` é zero, portanto estamos somando zero ao total, uma operação inofensiva. Só após efetuar essa soma inútil, o programa retornará ao início do bloco e verificará que a condição do `while` não é mais verdadeira, pois agora nosso `n` é igual a zero. É importante perceber que, apesar de o valor de `n` passar a ser zero na linha 8, a execução continua até o fim do bloco, passando pela linha 9, para só então ocorrer o retorno ao início do bloco e a verificação da

condição de continuidade da repetição.

Quando estudamos as condições lógicas no final do capítulo anterior, aprendemos que Python considera o valor 0 (zero) como sinônimo de "falso", e valores não-zero como "verdadeiros". Programadores experientes em Python costumam tirar proveito desse fato para abreviar as condições que colocam em seus ifs e whiles. Em nosso programa `somadora1.py`, a linha 7:

```
while n != 0:
```

pode ser escrita de forma mais abreviada assim:

```
while n:
```

Faça essa alteração no programa e experimente. Você verá que nada mudou no seu funcionamento. Isso porque, quando `n` é diferente de zero, a condição "`n`" expressa em `while n:` é considerada verdadeira, e a iteração é executada. Quando `n` passa a ser zero, a condição é falsa, encerrando a iteração.

Loops (quase) infinitos

Outra forma de escrever a somadora, mais elegante em minha opinião, é a mostrada na listagem 3.

Esconder número das linhas

```
1 # somadora2.py - somadora infinita - versao 2
2
3 print 'Digite os valores a somar seguidos de [ENTER]. '
4 print 'Para encerrar digite zero: 0'
5 total = 0
6 while 1:
7     n = float(raw_input(':'))
8     if n == 0: break
9     total = total + n
10 print 'TOTAL: %s' % total
```

Aqui a lógica é um pouco diferente: na linha 6 o loop while tem como condição o número 1. Como o número 1 é constante, e é considerado "verdadeiro" pelo interpretador Python, o loop das linhas 6 a 9 seria repetido infinitas vezes, em tese. Na prática, a linha 8 verifica se o valor de `n` é zero. Em caso afirmativo, o comando "break" é acionado. Isso faz com que o loop while seja interrompido imediatamente, e a execução do programa passa diretamente para a próxima linha após o bloco (linha 10 em nosso exemplo).

Essa forma de codificar, usando loops infinitos com breaks, não está de acordo com a Programação Estruturada, a filosofia dominante entre os programadores nos anos 70. O problema é que não fica imediatamente aparente qual é a condição de terminação do loop e alguns professores de computação podem descontar pontos por isso. Mas em se tratando de um bloco de apenas três linhas, não acho que isso seja um grande problema. A vantagem é que agora a função de leitura de dados ocorre em apenas um lugar no programa (na linha 7) e não em dois, como na versão anterior (linhas 5 e 8 de `somadora1.py`). Isso simplificará nossa próxima alteração. Além disso, não acontece mais a totalização inútil da linha 9, somando zero ao total na saída, porque o comando break da linha 8 faz o programa passar direto para a linha 10.

Uma forma mais natural de codificar esse loop seria usar comandos com o do/while ou repeat/until existentes em linguagens como C/C++/Java e Pascal/Delphi; nessas estruturas de controle, o teste é feito no fim do loop, garantindo a execução do bloco ao menos uma vez. É o que precisamos fazer aqui, mas Python não possui um comando de loop especial para essa situação. Vejamos outro

exemplo.

Suponha que você queira, por algum motivo estranho, somar os números naturais (1, 2, 3 etc.) até obter um total maior ou igual a 100. Observe na listagem 4 como ficaria o loop central para fazer isso em Delphi, Java e Python.

Delphi

```
REPEAT
    n := n + 1;
    total := total + n;
UNTIL (total >= 100);
```

Java

```
do {
    n = n + 1;
    total = total + n;
} while (total < 100);
```

Python

Esconder número das linhas

```
1 while True:
2     n = n + 1
3     total = total + n
4     if total >= 100: break
```

Note que os três programas acima estão incompletos; reproduzimos apenas o loop principal. Generalizando, qualquer loop com teste no final pode ser codificado em Python usando-se uma combinação de while 1 e if/break, assim:

```
while True:
    comando1
    comando2
    # etc.
    if condicao_final: break
```

Um programa mais tolerante

Um defeito das nossas somadoras, e de todos os programas que fizemos até agora, é que eles não toleram falhas na digitação. Se você rodar o programa `somadora2.py` e digitar apenas [Enter] para encerrar, verá a seguinte mensagem na tela:

```
Traceback (innermost last):
  File 'somadora1.py', line 7, in ?
    n = float(raw_input())
ValueError: empty string for float()
```

A segunda linha dessa mensagem identifica o local do erro: linha 7 do arquivo (file) `somadora1.py`. Na terceira linha está reproduzida a linha do programa onde ocorreu o problema, e a mensagem final informa qual foi o erro. Podemos traduzí-la assim: "Erro de valor: *string* vazia para a função `float()`".

O problema é que, ao digitarmos [Enter] sem fornecer um número, a função `raw_input()`

retorna uma *string* vazia (nada mais justo, pois nada foi digitado). Em seguida, a função `float()` tenta transformar a *string* vazia em um ponto flutuante, mas não sabe como. É ela que dispara a mensagem de erro, fazendo com que o programa seja interrompido antes de mostrar o valor total da soma.

Efeito semelhante pode ser obtido se você digitar um texto qualquer em vez de um número. Experimente.

Nesse caso, a mensagem de erro final é: "[ValueError](#): invalid literal for *float()*: blah". Nesse caso, a reclamação é de "*invalid literal*", significando que o texto fornecido para a função `float()` não se parece com um número.

A melhor maneira de resolver esse problema envolve o uso de mais uma comando de bloco de Python: o conjunto `try/except` (tentar/exceto). Esse par de palavras-chave formam o mecanismo de "tratamento de exceções" de Python, algo que só se encontra em linguagens bastante modernas como Java e as versões mais recentes de C++. A idéia básica é simples: no caso da nossa somadora, vamos tentar (`try`) converter a *string* digitada em *float*; se isso não der certo, temos uma exceção, que deve ter tratamento especial. No nosso caso, vamos simplesmente acionar o comando `break` para interromper o *loop* e exibir a totalização.

Veja na listagem abaixo como fica a `somadora3.py`, agora com tratamento de exceções.

Esconder número das linhas

```
1 # somadora3.py - somadora infinita - versao 3
2
3 print 'Digite os valores a somar seguidos de [ENTER]. '
4 print 'Para encerrar apenas [ENTER]. '
5 total = 0
6 while 1:
7     try:
8         n = float(raw_input(':'))
9         total = total + n
10    except:
11        break
12 print 'TOTAL: %s' % total
```

Vamos comentar apenas as diferenças em relação à versão anterior:

- **Linha 4:** mudamos a mensagem para o usuário, já que agora basta um [Enter] para encerrar.
- **Linha 7:** início do bloco `try`: tentaremos executar as linhas 8 e 9. Qualquer erro que ocorrer aqui será tratado no bloco `except`.
- **Linha 8:** aqui é o local mais provável do erro, quando `float()` tenta converter o resultado de `raw_input()`.
- **Linha 9:** se ocorrer um erro na linha 8, a linha 9 não será executada porque, dentro do bloco `try` qualquer erro causa a transferência imediata da execução para o bloco `except` correspondente.
- **Linha 10:** início do bloco `except` associado ao bloco `try` da linha 7
- **Linha 11:** tratamento do erro: em caso de exceção, vamos simplesmente interromper o `loop` com um comando `break`.
- **Linha 12:** como esta linha vem logo após um `loop` infinito (`while 1`), a única forma de chegarmos aqui é através de um `break`. Ou seja, nesse caso o `loop` só termina em consequência de uma exceção.

Experimente o programa agora: ele ficou muito mais conveniente de usar. Para interromper a soma e obter o total, basta teclar [Enter] em uma linha em branco. Uma boa melhoria na "usabilidade" da somadora!

Como tratar um erro de verdade

A terceira versão da nossa somadora ainda não chegou lá: tratamos da mesma forma a situação em que usuário não digitou nada e aquela onde ele digitou algo que não é um número válido em Python. Pode ser que o usuário seja um datilógrafo à moda antiga, que digita L minúsculo no lugar do dígito 1. Ou ainda alguém que quer usar, com toda razão, a "," como separador decimal (Python só aceita números com ponto decimal). Para diferenciar um tipo de erro do outro, e saber quando o usuário apenas quer encerrar o programa, precisamos guardar a linha que ele digitou antes de tentar transformá-la em um número. Veja como na listagem abaixo:

Esconder número das linhas

```
1 # somadora4.py - somadora infinita - versao 4
2
3 print 'Digite os valores a somar seguidos de .'
4 print 'Para encerrar apenas .'
5 total = 0
6 while 1:
7     try:
8         linha = raw_input(':')
9         n = float(linha)
10        total = total + n
11    except:
12        if len(linha) == 0:
13            break
14        elif ',' in linha:
15            print 'Use o . (ponto) como separador decimal.'
16        else:
17            print 'Isso nao parece um numero valido.'
18 print 'TOTAL: %s' % total
```

Vamos analisar as novidades dessa versão:

- **Linha 8:** a nova variável `linha` armazena a linha digitada pelo usuário, para verificação posterior.
- **Linha 9:** a linha é convertida em número.
- **Linha 11:** início do bloco que tratará os erros, provavelmente ocorridos na linha 9.
- **Linha 12:** a função `len()` retorna o número de itens de uma sequência; nesse caso, o número de caracteres da *string* `linha`. Se o número é igual a zero, então a string está vazia.
- **Linha 13:** no caso da *string* vazia, executamos um `break` porque o usuário não quer mais digitar.
- **Linha 14:** o operador `in` (em) retorna verdadeiro se o item à esquerda for encontrado na sequência à direita; nesse caso verificamos se existe uma vírgula dentro da *string* `linha`.
- **Linha 15:** como encontramos uma vírgula, vamos supor que o usuário tentou digitar um número fracionário. Então vamos sugerir que ele use o ponto decimal. Nesse caso, não executamos o `break`. Nenhum outro comando no bloco `if/elif/else` será executado, e o loop recomeçará de novo a partir da linha 6.
- **Linhas 16 e 17:** aqui vamos tratar todos os demais casos, dizendo que o que foi digitado não

se parece com um número. Novamente, sem o **break**, o *loop* reiniciará, e logo o sinal '.' aparecerá na tela aguardando nova digitação.

Associação de nomes a valores

Voltemos ao problema do cálculo de despesas. Nossa meta é fazer um programa que seja capaz de calcular a partilha de gastos de qualquer grupo de pessoas, e não apenas de Ana e Bia. Para isso, vamos precisar associar o nome das pessoas aos seus respectivos gastos. A linguagem Python possui uma estrutura de dados ideal para essa aplicação. É o dicionário, conhecido pelos programadores Perl como *hash* ou associação. Como ocorre em Perl, em Python o dicionário serve para associar chaves a valores. O mais comum é que as chaves sejam *strings*, como no nosso caso, onde as chaves serão nomes de pessoas. Mas as chaves podem ser qualquer tipo de objeto.

Em Python o dicionário é bem mais poderoso que em Perl, pois seus valores podem conter qualquer tipo de objeto como listas e até mesmo outros dicionários. Para entender rapidamente o funcionamento de um dicionário, nada melhor que experimentar com o interpretador interativo IDLE. Faça os seguintes testes, que explicaremos a seguir, com a abaixo:

```
Python (#0, Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)] on win32
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> dic = {}
>>> dic['ze'] = 300
>>> dic['mauricio'] = 100
>>> dic['heloisa'] = 150
>>> dic['ze']
300
>>> dic
{'mauricio': 100, 'ze': 300, 'heloisa': 150}
>>> dic['ze'] = 200
>>> dic
{'mauricio': 100, 'ze': 200, 'heloisa': 150}
>>> dic.keys()
['mauricio', 'ze', 'heloisa']
>>> dic['paulo']
Traceback (innermost last):
  File '', line 1, in ?
    dic['paulo']
KeyError: paulo
>>> dic.has_key('heloisa')
True
>>> dic.has_key('paulo')
False
>>>
```

- **Linha 3:** antes de usar um dicionário, é preciso criá-lo. Nesse caso, criamos um dicionário vazio. As chaves {} são usadas para representar dicionários, como veremos novamente nas linhas 10 e 13.
- **Linhas 4, 5 e 6:** criamos três itens no dicionário, usando as chaves 'ze', 'mauricio' e 'heloisa' e os valores 300, 100 e 150, respectivamente.
- **Linhas 7 e 8:** aqui acessamos o valor associado à chave 'ze' e obtemos o número 300.
- **Linhas 9 e 10:** agora acessamos o dicionário como um todo, e obtemos uma listagem entre chaves, com os itens separados por vírgula. Cada par de chave e valor aparece separado pelo sinal '.'. Note que a ordem dos itens não tem lógica aparente. Python não garante a ordem dos itens de um dicionário.

- **Linha 11:** associamos um novo valor a uma chave existente. Num dicionário, todas as chaves são únicas. Não pode haver dois itens com a mesma chave 'ze'. Assim, essa operação muda o valor associado à esta chave.
- **Linhas 12 e 13:** exibimos de novo o dicionário inteiro. Note que o valor associado à chave 'ze' mudou.
- **Linha 14:** o método `keys()` retorna a lista de chaves do dicionário. Um método nada mais é que uma função associada a um objeto, que deve ser invocada usando a sintaxe `objeto.metodo()`. Em nosso exemplo temos `dic.keys()`.
- **Linha 15:** aqui aparece a lista de chaves. Note que a lista, como sempre, vem delimitada por colchetes. O resultado do método `keys()` é uma lista de chaves, e não um dicionário.
- **Linhas 16 a 20:** tentamos acessar o valor de uma chave inexistente. Python reclama com a mensagem '[KeyError](#): paulo', indicando que o dicionário não possui uma chave igual a 'paulo'.
- **Linhas 21 a 24:** para verificar se uma determinada chave existe, usamos o método `has_key()` (tem_chave). Os exemplos mostram que `has_key()` retorna 1 quando a chave existe, e zero quando ela não existe.

Resolvendo o Orçamento da República

Agora que conhecemos o funcionamento básico dos dicionários, podemos implementar o nosso aplicativo de acerto de contas, que pode ser muito útil por exemplo na administração de uma república de universitários. Antes de mais nada, vejamos como vai funcionar o programa:

```
C:\PythonXX\Curso>python desprep1.py
Balanco de despesas da Republica Recanto Suico
```

```
(deixe um nome em branco para encerrar)
```

```
Digite o nome da pessoa: Marcos
Quanto gastou Marcos? 10
Digite o nome da pessoa: Alexandre
Quanto gastou Alexandre? 500
Digite o nome da pessoa: Tyrone
Quanto gastou Tyrone? 250
Digite o nome da pessoa: Harley
Quanto gastou Harley? 124,67
Numero invalido.
Quanto gastou Harley? 124.67
Digite o nome da pessoa:
```

```
Numero de pessoas: 4
Total de gastos: R$ 884.67
Gastos por pessoa: R$ 221.17
```

```
Saldo de Marcos: -211.17
Saldo de Alexandre: 278.83
Saldo de Tyrone: 28.83
Saldo de Harley: -96.50
```

```
C:\PythonXX\Curso>
```

- **Linha 1:** invocação do programa a partir da linha de comando.
- **Linhas 2 e 4:** apresentação e instruções de uso.

- **Linha 6:** o programa pergunta o nome de uma pessoa.
- **Linha 7:** a seguir, solicita o valor dos gastos daquela pessoa.
- **Linhas 8 a 12:** o processo é repetido quantas vezes for necessário.
- **Linha 13:** o usuário digita um número com vírgula no lugar do ponto decimal.
- **Linha 14:** o programa informa que o número é "inválido".
- **Linha 15:** novamente o programa pede o valor gasto por Harley.
- **Linha 16:** o usuário não fornece outro nome, encerrando a digitação.
- **Linhas 18 a 20:** o número de pessoas, o total gasto e o gasto médio por pessoa são calculados.
- **Linhas 22 a 25:** para cada pessoa, o programa exibe seu saldo. Aqueles que têm saldo negativo têm valores a pagar; os que de saldo positivo têm valores a receber.

Agora, vamos à listagem do programa `desprep1.py`:

Esconder número das linhas

```

1 #desprep1.py - calculo de despesas da republica
2
3 print 'Balanço de despesas da Republica Recanto Suico'
4 print
5 print '(deixe um nome em branco para encerrar)'
6 print
7 total = 0
8 contas = {}
9 while 1:
10     pessoa = raw_input('Digite o nome da pessoa: ')
11     if not pessoa: break
12     while 1:
13         resp = raw_input('Quanto gastou %s? ' % pessoa)
14         try:
15             gasto = float(resp)
16             break
17         except:
18             print 'Numero invalido.'
19     contas[pessoa] = gasto
20     total = total + gasto
21
22 num_pessoas = len(contas)
23 print
24 print 'Numero de pessoas: %d' % num_pessoas
25 print 'Total de gastos: R$ %.2f' % total
26 media = total/num_pessoas
27 print 'Gastos por pessoa: R$ %.2f' % media
28 print
29 for nome in contas.keys():
30     saldo = contas[nome] - media
31     print 'Saldo de %s: %.2f' % (nome, saldo)

```

- **Linhas 3 a 5:** exibir identificação e instruções.
- **Linha 7:** a variável total é inicializada com o valor zero. Isso é necessário em função da linha 21.
- **Linha 8:** o dicionário de contas é criado, sem conteúdo. Ele armazenará as contas de cada pessoa.

- **Linha 9:** início do loop principal.
- **Linha 10:** solicitamos um nome e armazenamos na variável `nome`.
- **Linha 11:** se a variável `nome` estiver vazia, nenhum nome foi digitado. Então executamos um `break` para deixar o loop principal, já que o usuário não quer mais fornecer nomes.
- **Linha 12:** início do loop secundário, para digitação do valor numérico.
- **Linha 13:** solicitamos o valor gasto pela pessoa em questão.
- **Linha 14:** início do bloco `try`, onde tentaremos converter a string digitada em número.
- **Linha 15:** a conversão fatídica. Em caso de erro aqui, o programa saltará para o bloco `except`, na linha 17.
- **Linha 16:** esse `break` só será executado se não ocorrer erro na linha 15. Sua função é interromper o loop secundário quando obtivermos um valor numérico.
- **Linhas 17 e 18:** o bloco `except` simplesmente exibe na tela a mensagem "Numero invalido". Aqui se encerra o loop secundário, que repetirá novamente a partir da linha 12, solicitando outro valor.
- **Linha 19:** o gasto obtido é armazenado no dicionário, usando o nome da pessoa como chave.
- **Linha 20:** o total de gastos é atualizado. Aqui é o final do loop principal. Daqui o programa voltará para a linha 9, e pedirá os dados da próxima pessoa.
- **Linha 22:** a função `len()` é usada para contar o número de itens no dicionário.
- **Linhas 23 a 25:** são exibidos o número de pessoas e total gasto. A notação `%.2f` faz com que os gastos apareçam com duas casas decimais, pois trata-se de um valor em dinheiro.
- **Linhas 26 a 27:** o gasto por cabeça é calculado e mostrado, também com duas casas decimais.
- **Linha 29:** aqui começamos um loop `for` que será repetido para cada nome que constar na lista de chaves do dicionário. A lista de chaves é obtida através do método `keys()`. A variável `nome` apontará, sucessivamente, para cada nome encontrado nesta lista.
- **Linha 30:** o valor gasto por uma pessoa é obtido acessando o dicionário com a expressão `contas[nome]`. Subtraímos o gasto médio para obter o saldo daquela pessoa.
- **Linha 31:** exibimos o nome e o saldo da pessoa. Esta é a última linha do loop `for`, que percorrerá todas as chaves do dicionário.

Nossa primeira simulação

Agora já sabemos tudo o que precisávamos para implementar um jogo simples, como havíamos prometido no capítulo anterior. Trata-se de uma simulação de pouso lunar, em modo texto. Esse programinha é baseado em um jogo clássico escrito para calculadoras HP-25. Nossa versão é bem mais fácil de entender que o original para calculadora. Em vez de explicar linha por linha o funcionamento do programa, colocamos comentários abundantes na própria listagem, delimitados pelo sinal `#`. Lembre-se de que não é preciso digitar os comentários (e o programa inteiro pode ser simplesmente copiado aqui no site). Esse simulador de alunissagem é um game de recursos mínimos, mas ainda assim deve valer alguns minutos de diversão, especialmente se você curte a física newtoniana ensinada no colegial.

Esconder número das linhas

```
1 # 0 jogo da alunissagem
2 # lunar.py
3 # importar funcao sqrt do modulo math
4 from math import sqrt
5
6 x = 500.    # altitude em pes
7 v = -50.    # velocidade em pes/s
8 g = -5.    # aceleracao gravitacional lunar em pes/s/s
9 t = 1.    # tempo entre jogadas em segundos
10 comb = 120. # quantidade de combustível
11
12 print 'Simulacao de alunissagem'
13 print
14 print '(digite a quantidade de combustivel a queimar)'
15
16 fmt = 'Alt: %6.2f Vel: %6.2f Comb: %3d'
17 while x > 0:    # enquanto nao tocamos o solo
18     msg = fmt % (x, v, comb) # montar mensagem
19     if comb > 0:    # ainda temos combustivel?
20         # obter quantidade de combustivel a queimar
21         resp = raw_input(msg + ' Queima = ')
22         try:    # converter resposta em numero
23             queima = float(resp)
24         except: # a resposta nao era um numero
25             queima = 0
26         if queima > comb: # queimou mais do que tinha?
27             queima = comb # entao queima o que tem
28         comb = comb - queima    # subtrai queimado
29         a = g + queima    # acel = grav + queima
30     else:    # sem combustivel
31         print msg    # mensagem sem perguntar
32         a = g    # aceleracao = gravidade
33         x0 = x    # armazenar posicao inicial
34         v0 = v    # armazenar velocidade inicial
35         x = x0 + v0*t + a*t*t/2    # calc. nova posicao
36         v = v0 + a*t    # calc. nova vel.
37 # se o loop acabou, tocamos no solo (x <= 0)
38 vf = sqrt(v0*v0 + 2*-a*x0) # calcular vel. final
39 print '>>>CONTATO! Velocidade final: %6.2f' % (-vf)
40 # avaliar pouso de acordo com a velocidade final
41 if vf == 0:
42     msg = 'Alunissagem perfeita!'
43 elif vf <= 2:
44     msg = 'Alunissagem dentro do padrao.'
45 elif vf <= 10:
46     msg = 'Alunissagem com avarias leves.'
47 elif vf <= 20:
48     msg = 'Alunissagem com avarias severas.'
49 else:
50     msg = 'Modulo lunar destruido no impacto.'
51 print '>>>' + msg
```

Como jogar

Seu objetivo é desacelerar a nave, queimando combustível na dosagem certa ao longo da queda, para tocar o solo lunar com uma velocidade bem próxima de zero. Se você quiser, pode usar um diagrama como o mostrado abaixo (colocamos em nosso site um desses em branco, para você imprimir e usar). As unidades estão no sistema inglês, como no original. O mais importante é você

saber que cada 5 unidades de combustível queimadas anulam a aceleração da gravidade. Se queimar mais do que 5 unidades, você desacelera; menos do que 5, você ganha velocidade. Primeiro, pratique seus pousos preocupando-se apenas com a velocidade final. Depois você pode aumentar a dificuldade, estabelecendo um limite de tempo: por exemplo, o pouso tem que ocorrer em exatos 13 segundos. Uma última dica: cuidado para não queimar combustível cedo demais. Se você subir, vai acabar caindo de uma altura ainda maior! Boas alunissagens!

Capítulo 5

O segredo dos objetos-função: saiba como criar seus próprios comandos

O simulador de alunissagem `lunar.py`, apresentado no último capítulo, tem 50 linhas de código. É um jogo bem simples, mas foi nosso exemplo mais extenso até o momento. Em termos de programação profissional, trata-se de um programa bem pequeno. No mundo real, softwares modestos têm milhares de linhas de código, e essa contagem chega aos milhões quando se fala de grandes aplicativos como o Microsoft Word ou o sistema operacional Linux. Ao se trabalhar com programas maiores, é fundamental poder dividir o trabalho em módulos, em vez de criar uma solução "monolítica" como a do nosso `lunar.py`, onde o programa inteiro está expresso em uma única sequência de comandos. A partir de agora, vamos ver porque e como modularizar nossos programas, utilizando os conceitos de função, objeto, classe, módulo e pacote.

Dividir para conquistar

Um programa modularizado facilita o planejamento, a distribuição de tarefas entre vários programadores, o controle de qualidade e a reutilização de soluções. Por exemplo, no capítulo anterior utilizamos várias vezes sequências de comandos para ler dados do usuário, parecidas com o fragmento do programa `desprepl.py` (mostrado na listagem abaixo).

O ideal seria reunir sequências como esta em um módulo que nós pudéssemos reutilizar facilmente em qualquer um de nossos programas, em vez de redigitar ou cortar e colar esse código sempre que precisarmos reutilizá-lo. Ao evitar a redigitação, não só economizamos tempo, mas ainda limitamos a propagação de "bugs", ou falhas de programação. Imagine se, após meses de programação, usando centenas de vezes o fragmento, descobrimos que ele contém um erro em sua lógica. Se o código foi copiado manualmente para cada programa onde foi utilizado, seremos obrigados a localizá-lo e corrigi-lo em centenas de arquivos diferentes. Por outro lado, se o fragmento foi devidamente empacotado em um módulo, a correção somente precisa ser feita em um arquivo.

Esconder número das linhas

```
1 while 1:
2     resp = raw_input('Quanto gastou %s? ' % pessoa)
3     try:
4         gasto = float(resp)
5         break
6     except:
7         print 'Numero invalido.'
```

Programação estruturada

A primeira grande onda a favor da modularização no desenvolvimento de software foi a chamada "programação estruturada". No início dos anos 70, essa expressão estava tão na moda quanto a "programação orientada a objetos" de hoje. Na realidade, a programação orientada a objetos, ou

OOP, pode ser entendida como uma radicalização da programação estruturada. A peça-chave da programação estruturada é o conceito de subprograma, um fragmento com começo, meio e fim, que desempenha um papel bem definido dentro de um programa maior. Na linguagem Python, um subprograma é definido através do comando de bloco `def`. Existem dois tipos de subprogramas: procedimentos e funções. Em Python, a única diferença entre eles é que as funções produzem valores, e os procedimentos não. Seguindo a tradição da linguagem C, os criadores do Python preferem falar apenas de funções, considerando os procedimentos apenas um tipo especial de função.

Vamos usar o IDLE para ver como se define uma função. Digite as duas linhas abaixo e tecle [Enter] duas vezes para concluir:

```
>>> def dobro(x):  
...     return x * 2
```

Aparentemente, nada acontece. Mas você acabou de definir uma função, chamada `dobro`, que está armazenada na memória do interpretador Python. Para ver sua função funcionar, basta invocá-la assim:

```
>>> dobro(3)  
6
```

Agora vamos aos detalhes da nossa definição de função. A primeira linha, `def dobro(x):`, traz duas informações importantes: o nome da função, `dobro`, e a presença de um argumento, `x`. O argumento é uma variável especial que é associada ao valor fornecido pelo usuário na invocação da função. Ao receber a instrução `dobro(3)`, Python associa `x` ao valor 3. A segunda linha da função, `return x * 2` pode ser lida da direita para a esquerda. Primeiro Python calcula a expressão `x * 2`. Em nosso exemplo, o `x` está associado ao valor 3, portanto o resultado é 6. O comando `return` sinaliza o fim da função, e faz com que o resultado seja passado para quem a invocou. No exemplo abaixo, a função é invocada no meio de uma expressão aritmética:

```
>>> y = dobro(7) + 1  
>>> y  
15  
>>>
```

É hora de quebrar algumas regras para ver o que acontece. Primeiro, experimente digitar isso:

```
>>> dobro()
```

O resultado será um "traceback" com a mensagem de erro "not enough arguments; expected 1, got 0", ou "argumentos insuficientes; 1 esperado, 0 recebido". Isso aconteceu porque nossa definição, `def dobro(x)`, obriga o usuário da função a fornecer um argumento. É possível criar uma função que não pede argumentos, como veremos depois.

Outro experimento interessante é digitar apenas o nome da função:

```
>>> dobro  
<function dobro at 82fa30>
```

Vale a pena parar e pensar no que acabou de acontecer.

Se você digita o nome da função sem parênteses, o interpretador não a executa, mas apenas informa a que se refere aquele nome. O que ocorre quando usamos o comando `def` é a criação, na memória, de um objeto do tipo "function", ou função. O nome fornecido após o comando `def` é associado ao objeto-função. Mas o objeto função existe independente do nome.

Funções como objetos

Acabamos de fazer uma afirmação importante, que vale a pena repetir: Python permite criar funções que são tratadas da mesma forma que outros objetos da linguagem, como números e listas. Para entender as implicações disso, é bom reforçar o nosso entendimento de como Python lida com os objetos que criamos. Para tanto, vamos deixar as funções um pouco de lado e voltar a brincar com listas:

```
>>> l = [10,20,30,40]
```

Acabamos de criar uma lista "l" com quatro elementos. Essa é a forma sucinta de dizer o que ocorreu. Uma descrição bem melhor é a seguinte: criamos uma lista com quatro elementos e associamos a variável "l" a esta lista. A letra "l" é apenas uma etiqueta que identifica a lista; é importante notar que a lista existe mesmo antes de receber uma etiqueta.

Comprove:

```
>>> m = l
>>> m
[10, 20, 30, 40]
>>>
```

Agora associamos m a l, ou melhor, à lista associada a l. Nosso objeto-lista agora tem duas etiquetas. Podemos usar qualquer uma delas para nos referirmos a ele, tanto que, ao digitarmos m, o interpretador mostra a mesma lista. Podemos também acessar e modificar um item específico da lista:

```
>>> m[2]
30
>>> m[2] = 55
>>> m
[10, 20, 55, 40]
>>>
```

Agora digite l e veja o resultado:

```
>>> l
[10, 20, 55, 40]
>>>
```

O que aconteceu com o l? Absolutamente nada! Ele continua sendo uma mera etiqueta colada em nosso objeto-lista. Mudamos a lista através da etiqueta m, mas tanto m quanto l referem-se à mesma lista, como você acabou de comprovar.

O mesmo ocorre com funções. Ao interpretar o código `def dobro(x): return x * 2`, Python cria um objeto-função e o associa à etiqueta dobro. Nada impede que você associe outras etiquetas ao mesmo objeto, assim:

```
>>> f = dobro
>>> f
<function dobro at 82fa30>
```

Note que o nome f agora está associado ao mesmo objeto-função que antes chamamos de dobro.

O novo nome também pode ser usado para invocar a função:

```
>>> f(19)
38
>>> y = f(17) + 2
>>> y
```

Ao tratar funções como objetos, Python deixa para trás linguagens mais tradicionais como C++ e Java, e se junta a uma classe de linguagens utilizadas em trabalhos avançados de Ciência da Computação: linguagens de programação funcional. A mais famosa delas, Lisp, tem sido ferramenta fundamental na pesquisa de Inteligência Artificial há várias décadas. Um dialeto simplificado de Lisp, chamado Scheme, é usado nos cursos introdutórios de computação do MIT (Massachusetts Institute of Technology), um dos mais importantes centros de pesquisa em informática do planeta. Como você vê, estudando Python estamos em ótima companhia.

Vejamos na prática uma vantagem de tratarmos funções como objetos. Python possui uma função poderosa chamada `map`. Vamos usá-la agora:

```
>>> map(dobro, m)
[20, 40, 110, 80]
>>>
```

Invocamos a função `map` com dois argumentos. O primeiro é a nossa função `dobro`, e o segundo é a lista `m`, `[10, 20, 55, 40]`. A função `map` aplica o objeto-função a cada item do segundo argumento. O resultado é a criação de um novo objeto-lista, sem modificar o original.

Veja este outro exemplo:

```
>>> map(str, m)
['10', '20', '55', '40']
>>>
```

Neste caso, usamos a função embutida (ou pré-definida) `str` para converter cada um dos itens numéricos em uma string.

Argumentos default

Como já dissemos, uma função não precisa retornar um valor. Veja este exemplo:

```
>>> def bom_dia():
...     print 'Bom dia, humanóide!'
```

Isso é o que chamamos de procedimento: uma função que faz alguma coisa (neste caso, imprime uma mensagem), mas não retorna um valor. Você pode invocá-lo assim:

```
>>> bom_dia()
Bom dia, humanóide!
>>>
```

É inútil usar esse procedimento em uma expressão:

```
>>> x = bom_dia()
Bom dia, humanóide!
>>> x
>>> x == None
1
>>>
```

Nossa função `bom_dia` dispensa argumentos, já que em sua definição não colocamos nada entre os parênteses. Para sermos mais simpáticos com nossos usuários, poderíamos modificá-la para aceitar um nome, desta maneira:

```
>>> def bom_dia(nome = 'humanóide'):
```

```
...     print 'Bom dia, %s!' % nome
```

Note que, neste caso, associamos um valor ao argumento nome. É o chamado valor "default", que será usado caso o argumento não seja fornecido.

Veja como:

```
>>> bom_dia('Martinha')
Bom dia, Martinha!
>>> bom_dia()
Bom dia, humanóide!
>>>
```

A idéia de argumento default é outro ponto forte da linguagem Python, oferecendo grande flexibilidade na definição de funções.

Usando módulos

Uma vez entendido o básico de funções, podemos passar para os módulos, que são coleções de funções. Antes de criarmos nossos próprios módulos, é bom aprender a usar módulos prontos, para não ficarmos "reinventado a roda". Assim como qualquer boa linguagem moderna, Python possui uma coleção de módulos com milhares de funções testadas e prontas para uso em diferentes tipos de aplicações. O Python inclui mais de 140 módulos, sem contar com a extensão gráfica Tk. E muitos outros podem ser encontrados a partir do site Python.org, quase todos livres e gratuitos.

Que tipo de coisa pode ser encontrada nessa vasta biblioteca? Eis alguns exemplos de módulos, apenas para dar uma idéia:

- **cgi**: programação de páginas dinâmicas para a Web
- **ftplib**: montagem de scripts para interação com servidores FTP
- **gzip**: leitura e escrita de arquivos comprimidos
- **math**: funções matemáticas (trigonometria, logaritmos etc.)
- **re**: buscas de texto avançadas com expressões regulares (como na linguagem Perl)
- **string**: operações com strings, incluindo conversões de listas
- **time**: hora atual e conversão de formatos de data
- **xmllib**: interpretação de arquivos em formato XML

Como primeiro exemplo de como se usa um módulo, vamos recorrer ao módulo `calendar`, um conjunto de funções de alto nível (ou seja, fáceis de usar) para gerar calendários. Voltando ao seu interpretador Python, digite o seguinte:

```
>>> import calendar
```

O comando `import` geralmente não produz um resultado visível. Ele localiza o módulo mencionado, carrega para a memória suas funções e executa os comandos de inicialização do módulo, se existirem. Em nosso caso, as funções do arquivo `calendar.py` acabaram de ser lidas para a memória. Para usá-las, você digita o nome do módulo e o nome da função separados por um `"."`:

```
>>> calendar.prmonth(2000,3)
      March 2000
Mo Tu We Th Fr Sa Su
    1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
```

```
20 21 22 23 24 25 26
27 28 29 30 31
>>>
```

Fornecendo o ano e o mês, você recebe o calendário do mês prontinho. Existe também uma função para gerar um calendário anual. Experimente:

```
>>> calendar.prcal(2000)
```

Devido a limitações das bibliotecas-padrão da linguagem C que são a base do Python, o módulo `calendar` não chega a ser um "calendário perpétuo". Ele só trabalha com datas de janeiro de 1970 a janeiro de 2038. Para os curiosos, a explicação é que, internamente, as funções de C armazenam datas contando o número de segundos transcorridos desde 1/1/1970. Exatamente sete segundos após 1:14 da madrugada do dia 19/01/2038, esse número excederá o limite de um número inteiro de 32 bits. É mais um bug do novo milênio...

Agora, vamos supor que você deseja exibir o calendário mensal de uma outra maneira, por exemplo, separando os dias por tabs, para facilitar a exportação para um programa de editoração eletrônica. Ou ainda, podemos querer gerar um calendário em HTML. Nesses dois casos, o resultado da função `prmonth()` não é muito útil. A função `monthcalendar()` nos dá mais liberdade. Veja como ela funciona:

```
>>> calendar.monthcalendar(2000,3)
[[0, 0, 1, 2, 3, 4, 5], [6, 7, 8, 9,
10, 11, 12], [13, 14, 15, 16, 17, 18,
19], [20, 21, 22, 23, 24, 25, 26], [27,
28, 29, 30, 31, 0, 0]]
>>>
```

O resultado é uma lista de listas. Cada uma das cinco listas de dentro representa uma semana com seus respectivos dias. Zeros aparecem nos dias que ficam fora do mês.

Agora vamos começar a destrinchar o resultado da função `monthcalendar`. Antes de mais nada, já que vamos usar muitas vezes essa função, podemos economizar alguma digitação se usarmos uma outra forma do comando `import`:

```
>>> from calendar import monthcalendar
```

Agora não precisaremos mais usar o prefixo `calendar`, podendo chamar a função `monthcalendar()` diretamente por seu nome; assim:

```
>>> for semana in monthcalendar(2000,3):
...     print semana
...
[0, 0, 1, 2, 3, 4, 5]
[6, 7, 8, 9, 10, 11, 12]
[13, 14, 15, 16, 17, 18, 19]
[20, 21, 22, 23, 24, 25, 26]
[27, 28, 29, 30, 31, 0, 0]
>>>
```

Através do comando `for`, listamos separadamente cada semana. Para trabalhar com cada dia individualmente, podemos criar outro comando `for` para percorrer cada uma das semanas. O resultado você pode ver na listagem 2.

A cada ciclo do primeiro `for`, a variável `semana` representa uma lista de sete dias. No segundo `for`, cada ciclo escreve na tela um dia. Para que todos os dias da semana apareçam na mesma linha, usamos um truque do comando `print`: a vírgula no final de `print '%s\t' % dia`, faz com que o Python

não inicie uma nova linha. Note pela indentação que o último comando print está dentro do primeiro for, e não dentro do segundo. Isso significa que esse print será executado uma apenas vez para cada semana.

Em programação, sempre há uma outra forma de obter algum resultado. Neste caso, não resistimos à tentação de mostrar um outro jeito de gerar a mesma listagem. O módulo string contém uma função, join, que serve para transformar listas em strings, concatenando (juntando) os elementos da lista com algum elemento separador. Para usar esta função, precisamos primeiro importá-la:

```
>>> from string import join
```

Para testá-la, experimente digitar algo assim:

```
>>> join(['1','2','3'])
'1 2 3'
>>> join(['1','2','3'], ' + ')
'1 + 2 + 3'
>>>
```

Note que o segundo argumento define a string que será usada como separador. No primeiro exemplo, omitimos o separador e Python usou o argumento default, um espaço. Agora vamos pegar uma semana do mês para fazer mais algumas experiências:

```
>>> s = monthcalendar(2000,3)[0]
>>> s
[0, 0, 1, 2, 3, 4, 5]
>>>
```

Aqui usamos o mecanismo de indexação de Python para obter apenas uma semana do mês. Chamamos a função monthcalendar(2000,3) com um índice, [0]. Lembre-se que monthcalendar retorna uma lista de listas. O índice [0] refere-se ao primeiro elemento da lista, ou seja a lista dos dias da primeira semana de março de 2000. Para exibir os dias dessa semana separados por tabs, usamos a função join com o caractere de tabulação, representado por '\t', assim:

```
>>> join(s, '\t')
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: first argument must be sequence of strings
>>>
```

Oops, Python reclamou: "Erro de tipo: o primeiro argumento tem que ser uma sequência de strings". Precisamos transformar a lista s, que contém números, em uma lista de strings. Felizmente, acabamos de descobrir como fazer isso usando a função map, no início deste capítulo:

```
>>> map(str, s)
['0', '0', '1', '2', '3', '4', '5']
```

Agora podemos executar o join:

```
>>> join(map(str,s), '\t')
'0\0110\0111\0112\0113\0114\0115'
```

O resultado ficou pouco apresentável, porque Python exibe o caractere "tab" através de seu código em numeração octal, \011. Mas isso não ocorre se usamos o comando print:

```
>>> print join(map(str,s), '\t')
0    1    2    3    4    5
>>>
```

Agora podemos fazer em duas linhas o que fizemos em quatro linhas na listagem abaixo:

```
>>> for semana in monthcalendar(2000,3):
...     for dia in semana:
...         print '%s\t' % dia,
...         print
...
0  0  1  2  3  4  5
6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31 0  0
>>>
```

Veja:

```
>>> for semana in monthcalendar(2000,3):
...     print join( map(str,semana), '\t')
...
0  0  1  2  3  4  5
6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31 0  0
>>>
```

Agora que aprendemos o básico sobre funções e sabemos como importar módulos, estamos prontos para criar nossas próprias "bibliotecas de código". Hoje vimos como definir e importar funções. Em seguida, aprenderemos como organizá-las em módulos e usá-las no contexto de programas maiores, aplicando primeiro conceitos da programação estruturada, e depois, da orientação a objetos. Mas isso terá que ficar para o mês que vem.

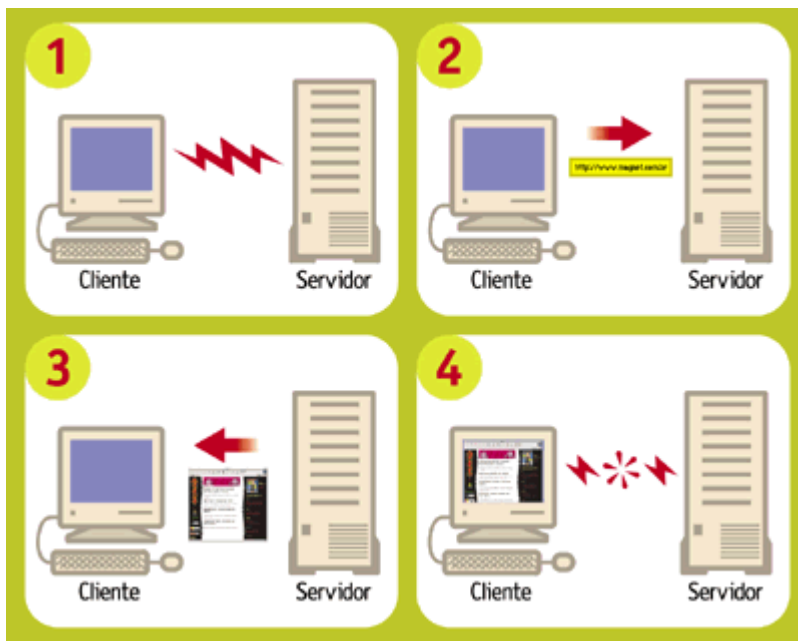
Capítulo 6

Perca o medo do servidor Apache e do protocolo CGI

É hora de colocar em prática os nossos conhecimentos de funções e módulos para criar nosso primeiro programa para a Internet: o Calendário Dinâmico. Uma vez instalado em um servidor web, ele exibirá o calendário do mês atual com o dia de hoje assinalado. Ao final desse capítulo você terá construído seus primeiros programas CGI em linguagem Python. Mas para chegar lá, é preciso entender o funcionamento de um CGI, e conhecer como se dá a operação básica de um servidor HTTP. Quem é quem do HTTP

A Web é construída a partir de duas tecnologias fundamentais: a linguagem HTML e o protocolo HTTP. HTML, ou Hypertext Markup Language é a codificação usada para criar as páginas da Web. Este não é o assunto deste curso, mas vamos usar um pouco de HTML nos exemplos desse capítulo.

O segundo pilar da Web é o HTTP, ou Hypertext Transport Protocol - protocolo de transporte de hipertexto. Esse é o conjunto de comandos e regras que define como deve ser a comunicação entre um browser (como o Internet Explorer ou o Mozilla) e um servidor HTTP (como o Apache ou o Internet Information Server). A expressão "servidor HTTP" pode significar duas coisas: o software que serve páginas via HTTP, ou o computador onde esse software está instalado. No mundo Unix, softwares servidores são chamados de "daemons", e a sigla HTTPd descreve um "HTTP daemon" genérico. Essa é a sigla que vamos usar para diferenciar o software do hardware.



A relação entre um browser e um HTTPd é descrita pelos computólogos como "cliente-servidor". Isso significa que a interação entre esses dois softwares sempre parte do browser, que é o cliente. O servidor não tem nenhuma iniciativa, limitando-se a responder aos comandos enviados pelo cliente.

Quando você digita uma URL como <http://www.tom-b.com/nomono/index.html>, o seu browser localiza e conecta-se ao servidor www.tom-b.com e envia-lhe o comando GET /nomono/index.html. O servidor então lê o arquivo index.html da pasta nomono, transmite seu conteúdo para o cliente e encerra a conexão. Esses são os passos básicos de qualquer interação de um browser com um HTTPd: conexão, solicitação, resposta e desconexão.

Páginas dinâmicas

No exemplo que acabamos de ver, `index.html` é o que chamamos de uma página estática. A resposta do servidor consiste apenas em enviar uma cópia do documento para o cliente. Sites que incluem transações (como lojas virtuais), interatividade (como chats), ou atualizações muito frequentes (como este) utilizam páginas dinâmicas. Neste caso, ao receber a URL <http://www.magnet.com.br/index.html>, nosso servidor HTTPd Apache passa a solicitação para o aplicativo Zope, instalado no servidor. O Zope monta imediatamente a página `index.html` listando as notícias mais recentes de nosso banco de dados, a hora atual e outros elementos. A página recém montada então é passada para o Apache, que finalmente a transmite para o seu navegador.

O Zope é apenas uma das tecnologias de páginas dinâmicas que existem hoje. O ASP da Microsoft, o ColdFusion da Macromedia e o software livre PHP são outros sistemas dinâmicos de montagem de páginas. Mas o mecanismo mais antigo, e também o mais simples de entender e de configurar, é o velho e bom CGI - ou Common Gateway Interface, um protocolo básico para interação entre um HTTPd e um programa gerador de páginas dinâmicas. É com ele que nos vamos trabalhar a partir de agora.

Configurando o seu HTTPd

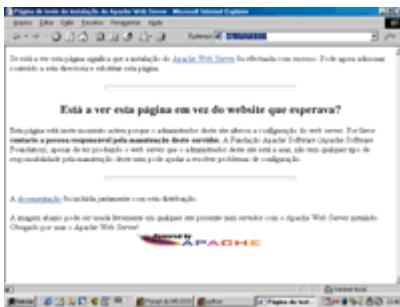
Para desenvolver aplicativos CGI é importante ter um bom ambiente de testes. O ideal é ter acesso a um HTTPd só para você na fase de desenvolvimento, para maior agilidade na depuração, e não correr o risco de comprometer o funcionamento de um servidor público com bugs nos seus CGIs

em construção.

Pode ser que o seu micro já possua um servidor HTTP. A Microsoft inclui o Personal Web Server ou o IIS em diferentes versões do Windows. Você pode tentar usar um desses HTTPd para fazer os exemplos, mas sugerimos fortemente que você vá até o <http://www.apache.org> baixe o Apache, que não custa nada (é open source), roda em qualquer plataforma Win32 ou Unix, é fácil de instalar e é tão robusto e versátil que é o HTTPd mais usado em todo mundo, além de ser o favorito disparado entre os melhores e maiores sites da Web. Vale a pena conhecê-lo, e o download tem apenas 3 MB.

No Windows, o Apache vem com um instalador bem amigável. Nossa única recomendação é instalar diretamente em um diretório como c:\apache e não no famoso c:\Arquivos de Programas. Isso porque os espaços em nomes de diretórios às vezes causam problemas na execução de programas originários do Unix como Python e o próprio Apache.

Uma vez terminada a instalação, você deve rodar o servidor, acionando o programa "Start Apache" que foi instalado em Iniciar > Programas > Apache Web Server. Isso faz abrir uma janela DOS com uma mensagem como "Apache/1.3.9 (Win32) running...". Não feche esta janela, pois isso encerrará a execução do servidor. Agora você pode ver se está tudo certo digitando essa URL mágica em seu browser: <http://127.0.0.1/>. Se a instalação foi bem sucedida, você verá uma página com o texto: "It Worked! The Apache Web Server is Installed on this Web Site!" (Funcionou! O servidor Apache está instalado neste Web Site!). (Figura 1).



Vale a pena saber que o endereço 127.0.0.1 tem um significado especial. Os criadores da Internet reservaram esse número IP para o "loopback", ou seja, testes de conexão de uma máquina com ela mesma. Em outras palavras, o endereço 127.0.0.1 sempre se refere à máquina onde você está, que é conhecida também pelo nome "localhost". Se o seu micro estiver bem configurado, a URL <http://localhost/> deve ter o mesmo efeito. Caso contrário, utilize o número IP e vamos em frente.

Seu primeiro CGI

Chegamos ao grande momento. Seguindo a tradição milenar, vamos começar fazendo um CGI em Python que produz uma página com as palavras "Olá, Mundo!". O programa completo você vê na listagem abaixo:

Esconder número das linhas

```
1 #!/python/python
2
3 print 'Content-type: text/html'
4 print
5 print '<HTML><BODY>'
6 print '<H1>Olá, Mundo!</H1>'
7 print '</BODY></HTML>'
```

Antes de digitar este exemplo, é bom destacar dois aspectos essenciais. Primeiramente, o comentário da linha 1 é importante. O CGI não vai funcionar sem ele. Ao executar um script CGI, o

Apache procura na primeira linha um comentário especial marcado pelos caracteres '#!'. Os sinais '#!' devem estar encostados na margem esquerda da página, e o restante dessa linha deve conter o caminho até o programa executável do interpretador que rodará o script. Note que o caminho pode ser especificado usando a barra '/' do Unix, em vez da contra-barra '\' preferida pelo Windows.

Em meu computador, o Python está instalado em uma pasta chamada "python" localizada no mesmo drive onde está o Apache (D:, no meu caso). Se o seu Python está instalado em outro lugar, você precisará alterar a linha 1. Em caso de dificuldades, nossa sugestão é que você desinstale o interpretador e o reinstale em uma pasta "python" diretamente na raiz do mesmo disco onde você acabou de instalar o Apache.

Outro detalhe importante são os print das linhas 3 e 4. Sua função não é meramente decorativa. O comando print 'Content-type: text/html' produz a parte obrigatória do cabeçalho da página, exigida pelo protocolo CGI. Este cabeçalho define o tipo do documento a ser transmitido de acordo com um esquema de classificação chamado "MIME". O "text/html" é o "MIME type" padrão de documentos HTML. Um texto ASCII puro teria o MIME type "text/plain" e um arquivo de foto JPEG seria "image/jpeg". O print da linha 4 gera uma linha em branco, que marca o fim do cabeçalho. Se uma dessas duas linhas não for digitada corretamente, o CGI não funcionará.

O restante da listagem apenas produz o código HTML de uma página muito simples. Os comandos marcados pelos sinais < e > são os chamados tags, ou marcações, da linguagem HTML. A marcação <H1>Manchete</H1>, por exemplo, define que uma manchete de nível 1, que será exibida pelo navegador como um texto em letras grandes.

Instalar e testar o CGI

O programinha de exemplo da seção anterior deverá ser salvo com o nome `ola.py` no diretório `cgi-bin` dentro da pasta do Apache. Este diretório é criado automaticamente pelo instalador do Apache no Windows, mas deve estar vazio inicialmente. Coloque o `ola.py` ali dentro e faça o grande teste: digite `http:// 127.0.0.1/cgi-bin/ola.py` em seu browser. Das duas, uma: ou você viu a página "Olá, Mundo!" e está feliz com seu primeiro CGI, ou ficou deprimido por encontrar uma mensagem de "Internal Server Error". Neste caso, saiba que você está em boa companhia: todo programador de CGI já se deparou com esta mensagem. Os que dizem que nunca a viram estão mentindo. Mesmo que seu script tenha funcionado, é proveitoso entender as causas de um "Internal Server Error" para saber como depurá-lo.

O "Internal Server Error" ocorre quando o script CGI não gera um cabeçalho mínimo, formado por uma linha de Content-type e uma linha em branco. É o que o nosso `ola.py` deveria fazer nas linhas 3 e 4. Vejamos passo a passo como diagnosticar a causa do problema.

1) Verifique se o script pode ser executado a partir do prompt do DOS.

Abra uma janela do DOS e digite:

```
X:\> c: (ou d:)
```

O passo acima não é necessário se você já está no disco certo.

```
X:\> cd \apache\cgi-bin
```

```
X:\> python ola.py
```

Neste momento, três coisas podem acontecer:

1. o script funciona perfeitamente, exibindo o cabeçalho, uma linha em branco, e o HTML da página - pule para o passo 2.
2. o DOS responde "Comando ou nome de arquivo inválido" - leia o passo 3;

3. o interpretador Python exibe um traceback e uma mensagem de erro qualquer - vá até o passo 4;

2) Se o script funciona a partir do prompt mas não através do Apache, existem quatro causas possíveis:

1. O comentário da linha 1 está incorreto. Lembre-se que é ele que informa ao Apache qual interpretador invocar para rodar o script. Se o seu interpretador Python (python.exe) foi instalado na pasta d:\python, então a linha 1 do seu script deve ser assim: `#! d:\python\python.exe` (Na verdade, a extensão .exe é dispensável, e se o Apache e o Python estão no mesmo disco, você pode usar a notação mais elegante do Unix e escrever apenas `#!/python/python`)
2. O script não foi colocado na pasta cgi-bin. Se o seu Apache foi instalado em c:\apache, o programa ola.py tem que estar exatamente neste local: c:\apache\cgi-bin\ola.py. Se este foi o problema, corrija e volte ao passo 1.
3. Apache pode não estar configurado para executar programas na pasta cgi-bin. Isso não deve acontecer em uma instalação nova do Apache, mas se você está usando um HTTPd que foi instalado por outra pessoa, ela pode ter mudado esta configuração. Neste caso, peça ajuda ao responsável pela instalação.
4. No Linux, ou em qualquer Unix, o Apache deverá ter permissão de executar o script ola.py. Para tanto, você pode precisar usar o comando `chmod` para setar o bit de execução de ola.py. Em Linux, o comando abaixo deve dar conta do recado:

```
$ chmod a+x ola.py
```

Se você usa outro Unix, experimente:

```
$ chmod 755 ola.py
```

Uma vez marcado como executável o script poderá ser invocado diretamente pelo nome, sem necessidade de mencionar o interpretador, assim:

```
$ ./ola.py
```

Se este teste funcionou, tente acionar o script novamente pelo browser, porque um shell do Unix também utiliza o comentário `#!` da linha 1 para localizar o interpretador. Se isto não deu certo, volte ao item 2a acima. Se o teste funcionou mas o programa exibe um traceback, vá até o passo 4.

3) Verifique se o interpretador Python (arquivo python.exe no DOS) está instalado corretamente e em local acessível. Se ele foi instalado em uma pasta chamada c:\python, o seguinte comando deve acionar o seu CGI:

```
X:\> c:\python\python ola.py
```

O que fazer então:

1. se agora o script funcionou perfeitamente, exibindo o cabeçalho, uma linha em branco, e o HTML da página, pule para o passo 2.
 2. se você continua vendo "Comando ou nome de arquivo inválido" ou outra mensagem do sistema operacional, verifique novamente o local exato da instalação do seu Python e se necessário, reinstale. Feito isso, volte para o passo 1.
- 4) Se ao rodar o script a partir do prompt você está vendo um traceback do interpretador Python, o problema está mesmo dentro do seu programa. Quando ocorre um erro de sintaxe (**SyntaxError**) o interpretador apenas leu, mas não chegou a executar nenhuma linha do seu script. Assim, o famoso cabeçalho "Content-type: ..." e a linha em branco não são enviados para o servidor, e o

traceback que o ajudaria a detectar o problema não chega ao seu browser, mas vai para um arquivo onde o Apache registra mensagens de erro. Este arquivo chama-se error.log e fica em /apache/logs/. Você pode inspecioná-lo com qualquer editor de texto. Outras vezes, o erro pode acontecer durante a execução e após o envio do cabeçalho. Neste caso, o traceback é perdido para sempre. É por isso que programadores de CGI experientes procuram testar exaustivamente seus scripts a partir da linha de comando antes de tentar acioná-lo pelo browser. Há também alguns truques que podem ser usados durante a depuração de um CGI para que as mensagens de erro sejam transmitidas para o browser. Em seguida veremos como.

Afinal, um CGI dinâmico

Nosso primeiro exemplo de CGI foi bolado para ser simples, mas é também um tanto tolo. Não gera nenhuma informação variável; o mesmo efeito poderia ser obtido com uma página estática. A página dinâmica mais simples que conseguimos imaginar é uma que mostre a hora certa (de acordo com o relógio do servidor). Para fazer um CGI assim, é bom conhecermos duas funções do módulo time. Vamos ver o que elas fazem acionando o interpretador Python. Primeiro, temos que importar as duas funções de dentro do módulo:

```
>>> from time import time, localtime
```

Podemos invocar diretamente a função time():

```
>>> time()
953500536.8
```

Que número é esse? Como explicamos no final do capítulo passado, o Python, assim como muitos programas originários da plataforma Unix, marca o tempo contando o número de segundos desde 1 de janeiro de 1970. Isto quer dizer que haviam se passado 953 milhões, 500 mil e 536 segundos e 8 décimos desde 1/1/1970 quando eu digitei time() no IDLE do meu computador. Isto é muito interessante, mas como transformar segundos transcorridos na hora atual? É para isso que serve a função localtime():

```
>>> t = time()
>>> localtime(t)
(2000, 3, 19, 18, 33, 19, 6, 79, 0)
>>>
```

Agora nós associamos os segundos transcorridos à variável t, e em seguida usamos a função localtime para transformar os segundos em uma sequência de 9 números que fornecem os seguintes dados:

```
localtime(t)[0:3] -> 2000, 3, 19 (ano, mês e dia)
localtime(t)[3:6] -> 18, 33, 19 (hora, minutos e segundos)
localtime(t)[6] -> 6 (dia da semana; 0 = segunda-feira; 6 = domingo)
localtime(t)[7] -> 79 (número do dia no ano; de 1 a 366 em anos bissextos)
localtime(t)[8] -> 0 (indicador de horário de verão; 0 = não; 1 = sim)
```

Esta função se chama localtime porque além de converter de segundos transcorridos para data e hora, ela o faz levando em conta o fuso horário configurado no sistema operacional, fornecendo portanto a hora local. Para saber a hora no meridiano de Greenwich, ou UTC no jargão moderno, usaríamos a função gmtime():

```
>>> from time import gmtime
>>> gmtime(t)
(2000, 3, 19, 21, 33, 19, 6, 79, 0)
>>>
```

Agora vamos combinar as novas funções para montar um CGI, hora.py, que mostre a hora local do servidor:

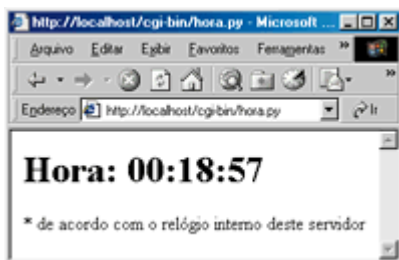
[Esconder número das linhas](#)

```
1 #!/python/python
2 # hora.py - CGI que exhibe a hora local do servidor
3
4 from time import time, localtime
5
6 print 'Content-type: text/html'
7 print
8
9 h, m, s = localtime(time())[3:6]
10 print '<HTML><BODY>'
11 print '<H1>Hora: %02d:%02d:%02d</H1>' % (h, m, s)
12 print '<P>* de acordo com o relógio interno deste servidor</P>'
13 print '</BODY></HTML>'
```

Uma vez salvo no diretório cgi-bin, este script poderá ser acessado pela URL `http://127.0.0.1/cgi-bin/hora.py`. A página gerada conterá a hora, minutos e segundos do instante em que ela foi invocada. Qual o defeito do nosso relógio em CGI? Experimente e você verá.

Um relógio que se atualiza

É um pouco frustrante acessar uma página que mostra a hora certa, com precisão de segundos, mas fica parada no tempo (Figura 2). Para atualizar os segundos, você tem que acionar o comando de "reload" do seu browser (Exibir ; Atualizar ou [F5] no Internet Explorer; View ; Reload ou [Control][R] no Navigator). Nossa página parece um relógio quebrado, que só mostra a hora certa quanto chacoalhado.



O ideal seria que o servidor atualizasse a página que está no seu browser a cada segundo. Infelizmente, isso é impossível. Como já dissemos, o protocolo HTTP é do tipo cliente-servidor, e isto quer dizer que a iniciativa de toda interação fica do lado do cliente. Não há como o servidor por conta própria enviar uma nova página sem que ela seja antes solicitada pelo navegador. Esta é uma limitação importante do protocolo HTTP que você precisa ter em mente ao bolar seus programas CGI.

Os browsers modernos suportam uma solução parcial para este problema. Eles reconhecem um cabeçalho especial chamado Refresh, cuja presença em um documento serve para instruir o browser a solicitar novamente a página após algum tempo. O argumento do Refresh é um número de segundos que o navegador deve esperar para pedir a atualização. Logo veremos como isso funciona na prática.

Para usar o Refresh basta acrescentar uma linha ao cabeçalho da resposta gerado pelo nosso CGI hora.py. A nova versão, `hora2.py` ficará assim:

[Esconder número das linhas](#)

```

1 #!/python/python
2 # hora2.py - CGI que exhibe continuamente hora local do servidor
3
4 from time import time, localtime
5
6 print 'Content-type: text/html'
7 print 'Refresh: 0.6'
8 print
9
10 h, m, s = localtime(time())[3:6]
11 print '<HTML><BODY>'
12 print '<H1>Hora: %02d:%02d:%02d</H1>' % (h, m, s)
13 print '<P>* de acordo com o relógio interno deste servidor</P>'
14 print '</BODY></HTML>'

```

A única novidade é a linha 7, onde acrescentamos "Refresh: 0.6" ao cabeçalho. Em vez de mandar o browser atualizar a página a cada 1 segundo, após alguns testes decidimos fazê-lo a cada 6 décimos de segundo. Fizemos assim porque quanto experimentamos com "Refresh: 1" a contagem freqüentemente pulava um segundo, por exemplo de 10:30:20 direto para 10:30:22. Isso não quer dizer que o relógio adiantava, porque a cada nova solicitação a hora certa estava sendo consultada pelo nosso CGI; mas como o tempo de espera somado ao tempo de solicitação e resposta era maior que 1 segundo, a exibição da hora sofria alguns sobressaltos.

Fazendo o refresh a cada 6 décimos, muitas vezes estamos atualizando a página duas vezes no mesmo segundo, o que é um desperdício de processamento. Mas pelo menos nos livramos da enervante perturbação na contagem. É claro que se o servidor estiver sobrecarregado, ele pode levar mais de um segundo para responder. Nesse caso, de nada adiantará se browser fizer novas solicitações a cada 0.6 segundo.

Calendário Dinâmico

Agora vamos juntar o que já sabemos sobre CGI com o módulo calendar que vimos no capítulo anterior para fazer um protótipo rápido do nosso Calendário Dinâmico. As passagens mais interessantes da listagem abaixo são comentados a seguir:

Esconder número das linhas

```

1 #!/python/python
2 # calendin.py - calendário dinâmico - protótipo 1
3
4 print 'Content-type: text/html\n'
5
6 try:
7     from time import time, localtime
8     from calendar import monthcalendar
9     from string import join
10
11     ano, mes = localtime(time())[:2]
12
13     print '<HTML><TITLE>Calendário Dinâmico</TITLE>'
14     print '<BODY>'
15
16     print '<H1>Calendário do mês %02d/%04d</H1>' % (mes, ano)
17     print '<PRE>'
18     for semana in monthcalendar(ano, mes):
19         print join( map(str, semana), '\t' )
20     print '</PRE>'
21
22 except:

```

```

23     import sys
24     sys.stderr = sys.stdout
25     from traceback import print_exc
26     print '<HR><H3>Erro no CGI:</H3><PRE>'
27     print_exc()
28     print '</PRE>'
29
30 print '</BODY></HTML>'

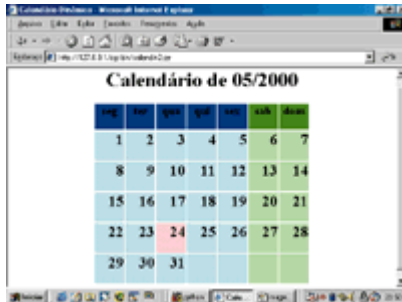
```

- **Linha 4:** logo de saída produzimos o cabeçalho mínimo necessário. A linha em branco, em vez de ser gerada por um segundo comando print, está incluída no final própria string do cabeçalho (o '\n' representa uma quebra de linha, e o próprio print produz outra quebra; assim obtemos a linha em branco para encerrar o cabeçalho).
- **Linha 6:** para facilitar a depuração, colocamos praticamente o CGI inteiro dentro de um bloco try/except. Qualquer falha na execução deste bloco será tratada a partir da linha 23. Com isso, esse script só deverá gerar um "Internal Server Error" se houver um erro de sintaxe, justamente o tipo de falha mais fácil de localizar rodando o script a partir de uma linha de comando. Desta forma a depuração fica bem mais simples.
- **Linhas 7 a 9:** importamos várias funções, todas velhas conhecidas.
- **Linha 11:** extraímos o ano e o mês do resultado de localtime(time()).
- **Linha 13:** iniciamos a produção do HTML, agora colocando um título na página (que aparecerá na barra de título da janela do browser).
- **Linhas 17 a 20:** para simplificar a formatação do calendário, colocamos seu conteúdo entre um par de tags <PRE></PRE>. O tag <PRE> faz com que o browser respeite as quebras de linha até o tag </PRE>. Normalmente, o navegador ignora tabulações e quebras de linha, tratando tudo como simples espaços, mas isso estragaria nosso calendário, pois queremos mostrar uma semana por linha. O código das linhas 18 e 19 foi roubado sem alterações da listagem 3 do capítulo anterior.
- **Linha 22:** abrimos um bloco except para tratar qualquer erro que tenha ocorrido até aqui. Abrir um bloco except sem qualificar o tipo de exceção que será tratado é normalmente uma má idéia, porque pode mascarar muitos bugs. Nesse caso, o except "pega-tudo" está justificado porque em seguida exibiremos o traceback completo, revelando qualquer bug que tentar se esconder.
- **Linhas 23 e 24:** importamos o módulo sys, para podermos manipular os objetos stdout e stderr. Esses são os chamados "dispositivos lógicos" para onde toda a saída de dados do Python é direcionada. Mensagens geradas pelo comando print são enviadas para stdout, que normalmente está associado à tela do computador ou terminal. Durante a execução de um CGI, o stdout é redirecionado para o HTTPd, que vai enviar para o cliente tudo o que passar por este dispositivo. Mensagens de erro e tracebacks do Python, no entanto, vão para stderr. Se o script é invocado pela linha de comando, o stderr também está associado à tela, e assim podemos ver os tracebacks. Mas ao executar um CGI, o HTTPd simplesmente ignora o dispositivo stderr após o envio do cabeçalho, ocasionando a perda dos tracebacks. Na linha 24 associamos sys.stderr ao objeto sys.stdout. Desta maneira as mensagens de erro passam a ser enviadas para o browser através do HTTPd, como ocorre com os textos gerados por print.
- **Linha 25:** importamos uma função do módulo traceback para uso na linha 27.
- **Linha 26:** geramos tags para uma linha horizontal (<HR>) e o título 'Erro no CGI:'; abrimos um tag <PRE> para manter a formatação original das linhas do traceback.

- **Linha 27:** usamos a função `print_exc()` do módulo `traceback` para gerar o texto de uma descrição de erro.
- **Linha 30:** encerramos o programa gerando os tags que marcam o fim de uma página HTML.

=== Protótipo melhorado

Agora que colocamos o calendário básico para funcionar, está na hora de melhorar sua apresentação. Vamos deixar de lado o recurso preguiçoso do tag `<PRE>` e colocar os dias do mês dentro de uma tabela construída em HTML (Figura 3). Aproveitando outros recursos daquela linguagem, vamos também colorir os finais de semana e assinalar o dia de hoje. Você encontra o programa `calendin2.py` na listagem abaixo.



| Seg | Ter | Qua | Qui | Sex | Sab | Dom |
|-----|-----|-----|-----|-----|-----|-----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| 29 | 30 | 31 | | | | |

Esconder número das linhas

```

1 #!/python/python
2 # calendin2.py - calendário dinâmico - protótipo 2
3
4 print 'Content-type: text/html\n'
5
6 try:
7     from time import time, localtime
8     from calendar import monthcalendar
9     from string import join
10
11     ano, mes, hoje = localtime(time())[:3]
12
13     print '<HTML><TITLE>Calendário Dinâmico</TITLE>'
14     print '<BODY>'
15     print '<CENTER>'
16     print '<H1>Calendário de %02d/%04d</H1>' % (mes, ano)
17     print '<TABLE>'
18     print '<TR>'
19     for dia_sem in ['seg', 'ter', 'qua', 'qui', 'sex', 'sab', 'dom']:
20         if dia_sem in ['sab', 'dom']: bgcolor = 'green'
21         else: bgcolor = 'blue'
22         print '<TH WIDTH="45" BGCOLOR="%s">' % bgcolor
23         print '<H3>%s</H3></TH>' % dia_sem
24     print '</TR>'
25     for semana in monthcalendar(ano, mes):
26         print '<TR>'
27         num_dia_sem = 0
28         for dia in semana:
29             if dia == hoje:
30                 bgcolor = 'pink'
31             elif num_dia_sem >= 5:
32                 bgcolor = 'lightgreen'
33             else:
34                 bgcolor = 'lightblue'
35             print '<TD ALIGN="RIGHT" BGCOLOR="%s">' % bgcolor

```

```

36         if dia != 0:
37             print '<H2>%2d</H2>' % dia
38             print '</TD>'
39             num_dia_sem = num_dia_sem + 1
40         print '</TR>'
41     print '</TABLE></CENTER>'
42
43 except:
44     import sys
45     from traceback import print_exc
46     sys.stderr = sys.stdout
47     print '<HR><H3>Erro no CGI:</H3><PRE>'
48     print_exc()
49     print '</PRE>'
50
51 print '</BODY></HTML>'

```

Principais novidades em relação à versão anterior:

- **Linha 11:** além do ano e do mês, guardamos o dia de hoje, para poder assinalá-lo no calendário.
- **Linha 15:** vamos centralizar tudo na página.
- **Linha 17:** abrimos o tag <TABLE> que conterá o calendário propriamente dito. Esta tabela só será fechada pelo tag </TABLE> na linha 41 do programa.
- **Linha 18:** iniciamos a primeira linha da tabela (<TR> = Table Row ou linha da tabela).
- **Linha 19:** vamos percorrer os nomes dos dias da semana para construir o cabeçalho da tabela.
- **Linhas 20 e 21:** sábados e domingos terão fundos verdes (green); demais dias terão fundos azuis (blue).
- **Linha 22:** tabelas em HTML são divididas em "células". Há dois tags para definir células, o <TH> (table head = cabeça) e o <TD> (table data = dados). Os dias da semana ficarão dentro de células TH com largura de 45 pixels e a cor de fundo definida acima. A largura dessas células determinará a largura das colunas.
- **Linha 23:** o nome de cada dia da semana é colocado entre tags <H3></H3>, para ênfase, e o tag </TH> é aplicado para fechar cada célula.
- **Linha 24:** fechamos a primeira linha da tabela.
- **Linha 25:** iniciamos um loop para percorrer cada semana do mês.
- **Linha 26:** abrimos a linha da tabela correspondente a uma das semanas.
- **Linha 27:** zeramos o contador que permitirá identificar sábados e domingos (a função monthcalendar retorna semanas com início às segundas-feiras; em nossa contagem, segunda-feira será o dia zero).
- **Linha 28:** iniciamos outro loop, agora para percorrer cada dia da semana.
- **Linhas 29 a 34:** a cor do fundo da próxima célula é definida assim: a célula de hoje é rosa (pink); sábados e domingos (dias 5 e 6 na semana) serão verde-claro (light green) e os demais dias serão azul-claro (light blue)
- **Linha 35:** abrimos a célula do dia, com a cor escolhida. Ela será fechada na linha 38.

- **Linhas 36 e 37:** se o número do dia é diferente de zero, colocamos o conteúdo da célula. Lembre-se que a função `monthcalendar` completa com zeros as semanas do primeiro e último dia do mês.
- **Linha 39:** incrementamos o contador de dia da semana. Aqui se encerra o bloco que percorre os dias.
- **Linha 40:** fechamos a linha da tabela. Fim do loop que corresponde às semanas.

Próximas paradas

Aqui termina nosso primeiro contato com programação aplicada à Web. Neste capítulo você instalou seu próprio servidor Apache, e implementou seus primeiros programas CGI. Os princípios que você está aprendendo aqui se aplicam a qualquer tecnologia de geração de páginas dinâmicas. Para construir os exemplos, lançamos mão de código HTML. Não é nosso objetivo aqui abordar esta outra linguagem, mas não podemos fazer coisas interessantes na Web sem conhecer um pouco dela. Vamos manter o uso de HTML em um nível bem elementar, e continuaremos explicando os tags mais importantes de cada exemplo, mas seu aproveitamento poderá ser melhor se você estudar por conta própria. O tutorial do Caique, que você encontra no site da [Magnet](#), é mais que suficiente para acompanhar o restante deste curso.

No próximo número, aperfeiçoaremos nosso calendário para torná-lo interativo: em vez de mostrar sempre o mês atual, vamos permitir que o usuário escolha um mês e um ano qualquer, e também navegue para frente e para trás de mês em mês. Isso nos levará a explorar o uso de URLs com argumentos e formulários em HTML. Até lá!

PS. A revista MAGNET deixou de ser publicada, então este tutorial não teve continuação...

- 1 Algumas referências ao *site* Magnet que ficaram fora de contexto deste tutorial aqui no [PythonBrasil](#), no entanto, deve ficar claro que os direitos e a autoria deste tutorial é de Luciano Ramalho e foi publicado inicialmente no *site* <http://www.magnet.com.br/>