Capítulo 4: Condicionais e recursividade

4.1 O operador módulo

O **operador módulo** trabalha com inteiros (e expressões que têm inteiros como resultado) e produz o resto da divisão do primeiro pelo segundo. Em Python, o operador módulo é um símbolo de porcentagem (%). A sintaxe é a mesma que a de outros operadores:

```
>>> quociente = 7 / 3
>>> print quociente
2
>>> resto = 7 % 3
>>> print resto
1
```

Então, 7 dividido por 3 é 2 e o resto é 1.

O operador módulo se revela surpreendentemente útil. Por exemplo, você pode checar se um número é divisível por outro - se x % y dá zero, então x é divisível por y.

Você também pode extrair o algarismo ou algarismos mais à direita de um número. Por exemplo, x % 10 resulta o algarismo mais à direita de x (na base 10). Similarmente, x % 100 resulta nos dois dígitos mais à direita.

4.2 Expressões booleanas

Uma expressão booleana é uma expressão que é verdadeira (True) ou é falsa (False). Em Python, uma expressão que é verdadeira tem o valor 1, e uma expressão que é falsa tem o valor 0.

O operador == compara dois valores e produz uma expressão booleana:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

No primeiro comando, os dois operadores são iguais, então a expressão avalia como **True** (verdadeiro); no segundo comando, 5 não é igual a 6, então temos **False** (falso).
O operador == é um dos operadores de comparação; os outros são:

```
x != y      # x é diferente de y
x < y       # x é menor que y
x > y       # x é maior que y
x >= y      # x é maior ou igual a y
x <= y      # x é menor ou igual a y
```

Embora esses operadores provavelmente sejam familiares a você, os símbolos em Python são diferentes dos símbolos da matemática. Um erro comum é usar um sinal de igual sozinho (=) em vez de um duplo (==). Lembre-se de que = é um operador de atribuição e == é um operador de comparação. Também não existem coisas como =< ou =>.

4.3 Operadores lógicos

Existem três operadores lógicos: and, or, not (e, ou, não). A semântica (significado) destes operadores é similar aos seus significados em inglês (ou português). Por exemplo, x > 0 and x < 10 é verdadeiro somente se x for maior que 0 *e* menor que 10.

n%2 == 0 or n%3 == 0 é verdadeiro se qualquer das condições for verdadeira, quer dizer, se o número n for divisível por 2 *ou* por 3.

Finalmente, o operador lógico not nega uma expressão booleana, assim, not(x > y) é verdadeiro se (x > y) for falso, quer dizer, se x for menor ou igual a y.

A rigor, os operandos de operadores lógicos deveriam ser expressões booleanas, mas Python não é muito rigoroso. Qualquer número diferente de zero é interpretado como verdadeiro (*True*):

```
>>> x = 5
>>> x and 1
1

>>> y = 0
>>> y and 1
0
```

Em geral, esse tipo de coisa não é considerado de bom estilo. Se você precisa comparar um valor com zero, deve fazê-lo explicitamente.

4.4 Execução condicional

Para poder escrever programas úteis, quase sempre precisamos da habilidade de checar condições e mudar o comportamento do programa de acordo com elas. As **instruções condicionais** nos dão essa habilidade. A forma mais simples é a instrução `if` (se):

```
if x > 0
  print "x é positivo"
```

A expressão booleana depois da instrução `if` é chamada de **condição**. Se ela é verdadeira (*true*), então a instrução endentada é executada. Se não, nada acontece.

Assim como outras instruções compostas, a instrução `if` é constituída de um cabeçalho e de um bloco de instruções:

```
CABECALHO:
  PRIMEIRO COMANDO
  ...
  ULTIMO COMANDO
```

O cabeçalho começa com uma nova linha e termina com dois pontos (:). Os comandos ou instruções endentados que seguem são chamados de **bloco**. A primeira instrução não endentada marca o fim do bloco. Um bloco de comandos dentro de um comando composto ou instrução composta é chamado de **corpo** do comando.

Não existe limite para o número de instruções que podem aparecer no corpo de uma instrução if, mas tem que haver pelo menos uma. Ocasionalmente, é útil ter um corpo sem nenhuma instrução (usualmente, como um delimitador de espaço para código que você ainda não escreveu). Nesse caso, você pode usar o comando pass, que indica ao Python: "passe por aqui sem fazer nada".

4.5 Execução alternativa

Um segundo formato da instrução if é a execução alternativa, na qual existem duas possibilidades e a condição determina qual delas será executada. A sintaxe se parece com:

```
if x % 2 == 0:
  print x, "é par"
else:
  print x, "é impar"
```

Se o resto da divisão de x por 2 for 0, então sabemos que x é par, e o programa exibe a mensagem para esta condição. Se a condição é falsa, o segundo grupo de instruções é executado. Desde que a condição deva ser verdadeira (*True*) ou falsa (*False*), precisamente uma das alternativas vai ser executada. As alternativas são chamadas **ramos** (*branches*), porque existem ramificações no fluxo de execução.

Por final, se você precisa checar a paridade de números com frequência, pode colocar este código dentro de uma função:

```
def imprimeParidade(x):
  if x % 2 == 0:
    print x, "é par"
  else:
    print x, "é impar"
```

Para qualquer valor de x, imprimeParidade exibe uma mensagem apropriada. Quando você a chama, pode fornecer uma expressão de resultado inteiro como um argumento:

```
>>> imprimeParidade(17)
>>> imprimeParidade(y+1)
```

4.6 Condicionais encadeados

Às vezes existem mais de duas possibilidades e precisamos de mais que dois ramos. Uma **condicional encadeada** é uma maneira de expressar uma operação dessas:

```
if x < y:
  print x, "é menor que", y
elif x > y:
  print x, "é maior que", y
else:
  print x, "e", y, "são iguais"
```

elif é uma abreviação de “else if” (“senão se”). De novo, precisamente um ramo será executado. Não existe limite para o número de instruções elif, mas se existir uma instrução else ela tem que vir por último:

```
if escolha == 'A':
  funcaoA()
elif escolha == 'B':
  funcaoB()
elif escolha == 'C':
  funcaoC()
else:
  print "Escolha inválida."
```

Cada condição é checada na ordem. Se a primeira é falsa, a próxima é checada, e assim por diante. Se uma delas é verdadeira, o ramo correspondente é executado, e a instrução termina. Mesmo que mais de uma condição seja verdadeira, apenas o primeiro ramo verdadeiro executa.

Como exercício, coloque os exemplos acima em funções chamadas comparar(x, y) e executar(escolha).

4.7 Condicionais aninhados

Um condicional também pode ser aninhado dentro de outra. Poderíamos ter escrito o exemplo tricotômico (dividido em três) como segue:

```
if x == y:
  print x, "e", y, "são iguais"
else:
  if x < y:
    print x, "é menor que", y
  else:
    print x, "é maior que", y
```

O condicional mais externo tem dois ramos. O primeiro ramo contém uma única instrução de saída. O segundo ramo contém outra instrução if, que por sua vez tem dois ramos. Os dois ramos são ambos instruções de saída, embora pudessem conter instruções condicionais também.

Embora a endentação das instruções torne a estrutura aparente, condicionais aninhados tornam-se difíceis de ler rapidamente. Em geral, é uma boa ideia evitar o aninhamento quando for possível.

Operadores lógicos frequentemente fornecem uma maneira de simplificar instruções condicionais aninhadas. Por exemplo, podemos reescrever o código a seguir usando uma única condicional:

```
if 0 < x:
  if x < 10:
    print "x é um número positivo de um só algarismo."
```

A instrução print é executada somente se a fizermos passar por ambos os condicionais, então, podemos usar um operador and:

```
if 0 < x and x < 10:
  print "x é um número positivo de um só algarismo."
```

Esses tipos de condições são comuns, assim, Python provê uma sintaxe alternativa que é similar à notação matemática:

```
if 0 < x < 10:
  print "x é um número positivo de um só algarismo."
```

4.8 A instrução return

O comando return permite terminar a execução de uma função antes que ela alcance seu fim. Uma razão para usá-lo é se você detectar uma condição de erro:

```
import math
def imprimeLogaritmo(x):
  if x <= 0:
    print "Somente números positivos, por favor."
    return
  resultado = math.log(x)
  print "O log de x é ", resultado
```

A função imprimeLogaritmo recebe um parâmetro de nome x. A primeira coisa que ela faz é checar se x é menor ou igual a 0, neste caso ela exibe uma mensagem de erro e então usa return para sair da função. O fluxo de execução imediatamente retorna ao ponto chamador, quer dizer, de onde a função foi chamada, e as linhas restantes da função não são executadas.

Lembre-se que para usar uma função do módulo de matemática, math, você tem de importá-lo.

4.9 Recursividade

Já mencionamos que é válido uma função chamar outra função, e você viu vários exemplos disso. Mas ainda não tínhamos dito que também é válido uma função chamar a si mesma.

Talvez não seja óbvio porque isso é bom, mas trata-se de uma das coisas mais mágicas e interessantes que um programa pode fazer. Por exemplo, dê uma olhada na seguinte função:

```
def contagemRegressiva(n):
  if n == 0:
    print "Fogo!"
  else:
    print n
    contagemRegressiva(n-1)
```

contagemRegressiva espera que o parâmetro, n, seja um inteiro positivo. Se n for 0, ela produz como saída a palavra "Fogo!". De outro modo, ela produz como saída n e então chama uma função de nome contagemRegressiva – ela mesma – passando n-1 como argumento.

O que acontece se chamarmos essa função da seguinte maneira:

```
>>> contagemRegressiva(3)
```

A execução de contagemRegressiva começa com n=3, e desde que n não é 0, produz como saída o valor 3, e então chama a si mesma...

A execução de contagemRegressiva começa com n=2, e desde que n não é 0, produz como saída o valor 2, e então chama a si mesma...

A execução de contagemRegressiva começa com n=1, e desde que n não é 0, produz como saída o valor 1, e então chama a si mesma...

A execução de contagemRegressiva começa com n=0, e desde que n é 0, produz como saída a palavra "Fogo!" e então retorna.

A contagemRegressiva que tem n=1 retorna.

A contagemRegressiva que tem n=2 retorna.

A contagemRegressiva que tem n=1 retorna.

E então estamos de volta em __main__ (que viagem!). Assim, a saída completa se parece com:

```
3
2
1
Fogo!
```

Como um segundo exemplo, dê uma olhada novamente nas funções novaLinha e tresLinhas:

```
def novaLinha():
  print
def tresLinhas():
  novaLinha()
  novaLinha()
  novaLinha()
```

Muito embora isso funcione, não seria muito útil se precisássemos gerar como saída 2 novas linhas, ou 106. Uma alternativa melhor seria esta:

```
def nLinhas(n):
  if n > 0:
    print
    nLinhas(n-1)
```

Esse programa é similar a contagemRegressiva. Sempre que n for maior que 0, ele gera como saída uma nova linha e então chama a si mesmo para gerar como saída n-1 linhas adicionais.

Deste modo, o número total de novas linhas é 1 + (n-1) que, se você estudou álgebra direitinho, vem a ser o próprio n.

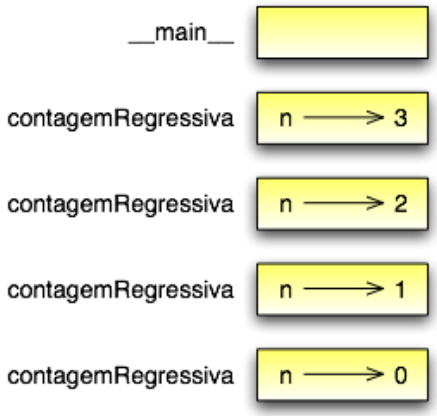
O processo de uma função chamando a si mesma é chamado de **recursividade**, e tais funções são ditas recursivas.

4.10 Diagramas de pilha para funções recursivas

Na Seção 3.11, usamos um diagrama de pilha para representar o estado de um programa durante uma chamada de função. O mesmo tipo de diagrama pode ajudar a interpretar uma função recursiva.

Toda vez que uma função é chamada, Python cria um novo quadro (*frame*) para a função, que contém as variáveis locais e parâmetros da função. Para uma função recursiva, terá que existir mais de um quadro na pilha ao mesmo tempo.

Esta figura mostra um diagrama de pilha para contagemRegressiva, chamada com n = 3:



Como de costume, no topo da pilha está o quadro para __main__. Ele está vazio porque nem criamos qualquer variável em __main__ nem passamos qualquer valor para ele.

Os quatro quadros contagemRegressiva têm valores diferentes para o parâmetro n. A parte mais em baixo na pilha, onde n=0, é chamada de **caso base**. Ele não faz uma chamada recursiva, então não há mais quadros.

Como exercício, desenhe um diagrama de pilha para nLinhas chamada com n=4.

4.11 Recursividade infinita

Se uma recursividade nunca chega ao caso base, ela prossegue fazendo chamadas recursivas para sempre, e o programa nunca termina. Isto é conhecido como recursividade infinita, e geralmente não é considerada uma boa ideia. Aqui está um programa mínimo com uma recursividade infinita:

```
def recursiva():
  recursiva()
```

Na maioria dos ambientes de programação, um programa com recursividade infinita na verdade não roda para sempre. Python reporta uma mensagem de erro quando a profundidade máxima de recursividade é alcançada:

```
File "<stdin>", line 2, in recursiva
(98 repetitions omitted)
File "<stdin>", line 2, in recursiva
RuntimeError: Maximum recursion depth exceeded
```

Este traceback é um pouco maior do que aquele que vimos no capítulo anterior. Quando o erro ocorre, existem 100 quadros recursiva na pilha!

Como exercício, escreva uma função com recursividade infinita e rode-a no interpretador Python.

4.12 Entrada pelo teclado

Os programas que temos escrito até agora são um pouco crus, no sentido de não aceitarem dados entrados pelo usuário. Eles simplesmente fazem a mesma coisa todas as vezes.

Python fornece funções nativas que pegam entradas pelo teclado. A mais simples é chamada raw_input. Quando esta função é chamada, o programa pára e espera que o usuário digite alguma coisa. Quando o usuário aperta a tecla Enter ou Return, o programa prossegue e a função raw_input retorna o que o usuário digitou como uma string:

```
>>> entrada = raw_input()
O que você está esperando?

>>> print entrada
O que você está esperando?
```

Antes de chamar raw_input, é uma boa ideia exibir uma mensagem dizendo ao usuário o que ele deve entrar. Esta mensagem é uma como se fosse uma pergunta (*prompt*). Esta pergunta pode ser enviada como um argumento para raw_input:

```
>>> nome = raw_input("Qual... é o seu nome? ")
Qual... é o seu nome? Arthur, Rei dos Bretões!

>>> print nome
Arthur, Rei dos Bretões!
```

Se esperamos que a entrada seja um inteiro, podemos usar a função input:

```
pergunta = "Qual... é a velocidade de vôo de uma andorinha?\n"
velocidade = input(pergunta)
```

Se o usuário digita uma string de números, ela é convertida para um inteiro e atribuída a velocidade. Infelizmente, se o usuário digitar um caractere que não seja um número, o programa trava:

```
>>> velocidade = input(pergunta)

Qual... é a velocidade de vôo de uma andorinha?
De qual você fala, uma andorinha Africana ou uma Europeia?
SyntaxError: invalid syntax
```

Para evitar esse tipo de erro, geralmente é bom usar raw_input para pegar uma string e, então, usar funções de conversão para converter para outros tipos.

4.13 Glossário

aninhamento (*nesting*)

Estrutura de programa dentro da outra, como um comando condicional dentro de um bloco de outro comando condicional.

bloco (*block*)

Grupo de comandos consecutivos com a mesma endentação.

caso base (*base case*)

Bloco de comando condicional numa função recursiva que não resulta em uma chamada recursiva.

comando composto (*compound statement*)

Comando que consiste de um cabeçalho e um corpo. O cabeçalho termina com um dois-pontos (:). O corpo é endentado em relação ao cabeçalho.

comando condicional (*conditional statement*)

Comando que controla o fluxo de execução dependendo de alguma condição.

condição (*condition*)
A expressão booleana que determina qual bloco será executado num comando condicional.

corpo (*body*)
O bloco que se segue ao cabeçalho em um comando composto.

expressão booleana (*boolean expression*)
Uma expressão que é verdadeira ou falsa.

operador de comparação (*comparison operator*)
Um dos operadores que compara dois valores: `==`, `!=`, `>`, `<`, `>=`, e `<=`.

operador lógico (*logical operator*)
Um dos operadores que combina expressões booleanas: `and`, `or`, e `not`.

operador módulo (*modulus operator*)
Operador denotado por um símbolo de porcentagem (%), que trabalha com inteiros e retorna o resto da divisão de um número por outro.

prompt
Indicação visual que diz ao usuário que o programa está esperando uma entrada de dados.

recursividade (*recursion*)
O processo de chamar a própria função que está sendo executada.

recursividade infinita (*infinite recursion*)
Função que chama a si mesma recursivamente sem nunca chegar ao caso base. Após algum tempo, uma recursividade infinita causa um erro de execução.

Capítulo 5: Funções frutíferas

5.1 Valores de retorno

Algumas das funções nativas do Python que temos usado, como as funções matemáticas, produziram resultados. Chamar a função gerou um novo valor, o qual geralmente atribuímos à uma variável ou usamos como parte de uma expressão:

```
e = math.exp(1.0)
altura = raio * math.sin(angulo)
```

Mas até agora, nenhuma das funções que nós escrevemos retornou um valor.

Neste capítulo, iremos escrever funções que retornam valores, as quais chamaremos de **funções frutíferas**, ou funções que dão frutos, na falta de um nome melhor. O primeiro exemplo é area, que retorna a área de um círculo dado o seu raio:

```
import math
def area(raio):
  temp = math.pi * raio**2
  return temp
```

Já vimos a instrução return antes, mas em uma função frutífera a instrução return inclui um **valor de retorno**. Esta instrução significa: "Retorne imediatamente desta função e use a expressão em seguida como um valor de retorno". A expressão fornecida pode ser arbitrariamente complicada, de modo que poderíamos ter escrito esta função de maneira mais concisa:

```
def area(raio):
  return math.pi * raio**2
```

Por outro lado, variáveis temporárias como temp muitas vezes tornam a depuração mais fácil. Às vezes é útil ter múltiplos comandos return, um em cada ramo de uma condicional:

```
def valorAbsoluto(x):
  if x < 0:
    return -x
  else:
    return x
```

Já que estes comandos return estão em ramos alternativos da condicional, apenas um será executado. Tão logo um seja executado, a função termina sem executar qualquer instrução ou comando subsequente.

O código que aparece depois de uma instrução return, ou em qualquer outro lugar que o fluxo de execução jamais alcance, é chamado código morto (*dead code*).

Em uma função frutífera, é uma boa ideia assegurar que todo caminho possível dentro do programa encontre uma instrução return. Por exemplo:

```
def valorAbsoluto(x):
  if x < 0:
    return -x
  elif x > 0:
    return x
```

Este programa não está correto porque se x for 0, nenhuma das condições será verdadeira, e a função terminará sem encontrar um comando return. Neste caso, o valor de retorno será um valor especial chamado **None**:

```
>>> print valorAbsoluto(0)
None
```

Como exercício, escreva uma função compare que retorne 1 se x > y, 0 se x == y e -1 se x <

5.2 Desenvolvimento de programas

Neste ponto, você deve estar apto a olhar para funções completas e dizer o que elas fazem. Também, se você vem fazendo os exercícios, você escreveu algumas pequenas funções.

Conforme escrever funções maiores, você pode começar a ter mais dificuldade, especialmente com erros em tempo de execução (erros de runtime) ou erros semânticos.

Para lidar com programas de crescente complexidade, vamos sugerir uma técnica chamada desenvolvimento incremental. A meta do desenvolvimento incremental é evitar seções de depuração (*debugging*) muito longas pela adição e teste de somente uma pequena quantidade de código de cada vez.

Como exemplo, suponha que você queira encontrar a distância entre dois pontos, dados pelas coordenadas (x1,y1) e (x2,y2). Pelo teorema de Pitágoras, a distância é:

distancia = V (x2 - x1)2 + (y2 - y1)2 (5.1)

XXX: falta o sinal de raiz e elevar os expoentes desta fórmula

O primeiro passo é considerar como deveria ser uma função distancia em Python. Em outras palavras, quais são as entradas (parâmetros) e qual é a saída (valor de retorno)?

Neste caso, os dois pontos são as entradas, os quais podemos representar usando quatro parâmetros. O valor de retorno é a distância, que é um valor em ponto flutuante.

Já podemos escrever um esboço da função:

```
def distancia(x1, y1, x2, y2):
  return 0.0
```

Obviamente, esta versão da função não calcula distâncias; ela sempre retorna zero. Mas ela está sintaticamente correta, e vai rodar, o que significa que podemos testá-la antes de torná-la mais complicada.

Para testar a nova função, vamos chamá-la com valores hipotéticos:

```
>>> distancia(1, 2, 4, 6)
0.0
```

Escolhemos estes valores de modo que a distância horizontal seja igual a 3 e a distância vertical seja igual a 4; deste modo, o resultado é 5 (a hipotenusa de um triângulo 3-4-5).

Quando testamos uma função, é útil sabermos qual o resultado correto.

Neste ponto, já confirmamos que a função está sintaticamente correta, e podemos começar a adicionar linhas de código. Depois de cada mudança adicionada, testamos a função de novo.

Se um erro ocorre em qualquer ponto, sabemos aonde ele deve estar: nas linhas adicionadas mais recentemente.

Um primeiro passo lógico nesta operação é encontrar as diferenças x2 - x1 e y2 - y1. Nós iremos guardar estes valores em variáveis temporárias chamadas dx e dy e imprimi-las:

```
def distancia(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  print "dx vale", dx
  print "dy vale", dy
  return 0.0
```

Se a função estiver funcionando, as saídas deverão ser 3 e 4. Se é assim, sabemos que a função está recebendo os parâmetros corretos e realizando o primeiro cálculo corretamente. Se não, existem poucas linhas para checar.

Em seguida, calcularemos a soma dos quadrados de dx e dy:

```
def distancia(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  dquadrado = dx**2 + dy**2
  print "dquadrado vale: ", dquadrado
  return 0.0
```

Note que removemos os comandos print que havíamos escrito no passo anterior. Código como este ajuda a escrever o programa, mas não é parte do produto final (em inglês é usado o termo *scaffolding*).

De novo, nós vamos rodar o programa neste estágio e checar a saída (que deveria ser 25). Finalmente, se nós tínhamos importado o módulo matemático math, podemos usar a função sqrt para computar e retornar o resultado:

```
def distancia(x1, x2, y1, y2):
  dx = x2 - x1
  dy = y2 - y1
  dquadrado = dx**2 + dy**2
  resultado = math.sqrt(dquadrado)
  return resultado
```

Se isto funcionar corretamente, você conseguiu. Caso contrário, talvez fosse preciso imprimir (exibir) o valor de resultado antes da instrução return.

Enquanto for iniciante, você deve acrescentar apenas uma ou duas linhas de código de cada vez. Conforme ganhar mais experiência, você se verá escrevendo e depurando pedaços maiores. De qualquer modo, o processo de desenvolvimento incremental pode poupar um bocado de tempo de depuração.

Os aspectos chave do processo são:

1. Comece com um programa que funciona e faça pequenas mudanças incrementais. Em qualquer ponto do processo, se houver um erro, você saberá exatamente onde ele está.
2. Use variáveis temporárias para manter valores intermediários de modo que você possa exibi-los e checá-los.
3. Uma vez que o programa funcione, você pode querer remover algum código muleta, ou algum *scaffolding* ou consolidar múltiplos comandos dentro de expressões compostas, mas somente se isto não tornar o programa difícil de ler.

Como um exercício, use o desenvolvimento incremental para escrever uma função chamada hipotenusa que retorna a medida da hipotenusa de um triângulo retângulo dadas as medidas dos dois catetos como parâmetros. Registre cada estágio do desenvolvimento incremental conforme você avance.

5.3 Composição

Conforme você poderia esperar agora, você pode chamar uma função de dentro de outra. Esta habilidade é chamada de **composição**.

Como um exemplo, vamos escrever uma função que recebe dois pontos, o centro de um círculo e um ponto em seu perímetro, e calcula a área do círculo.

Assuma que o ponto do centro está guardado nas variáveis `xc` e `yc`, e que o ponto do perímetro está nas variáveis `xp` e `yp`. O primeiro passo é encontrar o raio do círculo, o qual é a entre os dois pontos. Felizmente, temos uma função, `distancia`, que faz isto:

```
Raio = distancia(xc, yc, xp, yp)
```

O segundo passo é encontrar a área de um círculo com o raio dado e retorná-la:

```
resultado = area(raio)

return resultado
```

Juntando tudo numa função, temos:

```
def area2(xc, yc, xp, yp):
  raio = distancia(xc, yc, xp, yp)
  resultado = area(raio)
  return resultado
```

Chamamos à esta função de `area2` para distinguir da função `area`, definida anteriormente. Só pode existir uma única função com um determinado nome em um determinado módulo.

As variáveis temporárias `raio` e `resultado` são úteis para o desenvolvimento e para depuração (*debugging*), mas uma vez que o programa esteja funcionando, podemos torná-lo mais conciso através da composição das chamadas de função:

```
def area2(xc, yc, xp, yp):
  return area(distancia(xc, yc, xp, yp))
```

Como exercício, escreva uma função ``coeficienteAngular(x1, y1, x2, y2)`` que retorne a coeficiente angular de uma linha dados os pontos (x1, y1) e (x2, y2). Depois use esta função em uma função chamada ``cortaY(x1, y1, x2, y2)`` que retorne a interseção da linha com o eixo y, dados os pontos (x1, y1) e (x2, y2).

5.4 Funções booleanas

Funções podem retornar valores booleanos, o que muitas vezes é conveniente por ocultar testes complicados dentro de funções. Por exemplo:

```
def ehDivisivel(x, y):
  If  x % y == 0:
    return True  # é verdadeiro (True), é divisível
  else:
    return False # é falso (False), não é divisível
```

O nome desta função é ehDivisivel (“é divisível”). É comum dar a uma função booleana nomes que soem como perguntas sim/não. ehDivisivel retorna ou **True** ou **False** para indicar se x é ou não é divisível por y.

Podemos tornar a função mais concisa se tirarmos vantagem do fato de a condição da instrução if ser ela mesma uma expressão booleana. Podemos retorná-la diretamente, evitando totalmente o if:

```
def ehDivisivel(x, y):
  return x % y == 0
```

Esta sessão mostra a nova função em ação:

```
>>> ehDivisivel(6, 4)
False

>>> ehDivisivel(6, 3)
True
```

Funções booleanas são frequentemente usadas em comandos condicionais:

```
if ehDivisivel(x, y):
  print "x é divisível por y"
else:
  print "x não é divisível por y"
```

Mas a comparação extra é desnecessária.

Como exercício, escreva uma função estaEntre(x, y, z) que retorne **True** se y < x < z ou **False**, se não.

5.5 Mais recursividade

Até aqui, você aprendeu apenas um pequeno subconjunto da linguagem Python, mas pode ser que te interesse saber que este pequeno subconjunto é uma linguagem de programação completa, o que significa que qualquer coisa que possa ser traduzida em operação computacional pode ser expressa nesta linguagem. Qualquer programa já escrito pode ser reescrito usando somente os aspectos da linguagem que você aprendeu até agora (usualmente, você precisaria de uns poucos comandos para controlar dispositivos como o teclado, mouse, discos, etc., mas isto é tudo).

Provar esta afirmação é um exercício nada trivial, que foi alcançado pela primeira vez por Alan Turing, um dos primeiros cientistas da computação (alguém poderia dizer que ele foi um matemático, mas muitos dos primeiros cientistas da computação começaram como matemáticos). Por isso, ficou conhecido como Tese de Turing. Se você fizer um curso em Teoria da Computação, você terá chance de ver a prova.

Para te dar uma ideia do que você pode fazer com as ferramentas que aprendeu a usar até agora, vamos avaliar algumas funções matemáticas recursivamente definidas. Uma definição recursiva é similar à uma definição circular, no sentido de que a definição faz referência à coisa que está sendo definida. Uma verdadeira definição circular não é muito útil:

vorpal: adjetivo usado para descrever algo que é vorpal.

Se você visse esta definição em um dicionário, ficaria confuso. Por outro lado, se você procurasse pela definição da função matemática fatorial, você encontraria algo assim:

0! = 1
n! = n.(n-1)!

Esta definição diz que o fatorial de 0 é 1, e que o fatorial de qualquer outro valor, n, é n multiplicado pelo fatorial de n-1.

Assim, 3! (lê-se “3 fatorial” ou “fatorial de 3”) é 3 vezes 2!, o qual é 2 vezes 1!, o qual é 1 vezes 0!. Colocando tudo isso junto, 3! igual 3 vezes 2 vezes 1 vezes 1, o que é 6.

Se você pode escrever uma definição recursiva de alguma coisa, você geralmente pode escrever um programa em Python para executá-la. O primeiro passo é decidir quais são os parâmetros para esta função. Com pouco esforço, você deverá concluir que fatorial recebe um único parâmetro:

```
def fatorial(n):
```

Se acontece de o argumento ser 0, tudo o que temos de fazer é retornar 1:

```
def fatorial(n):
  if n == 0:
    return 1
```

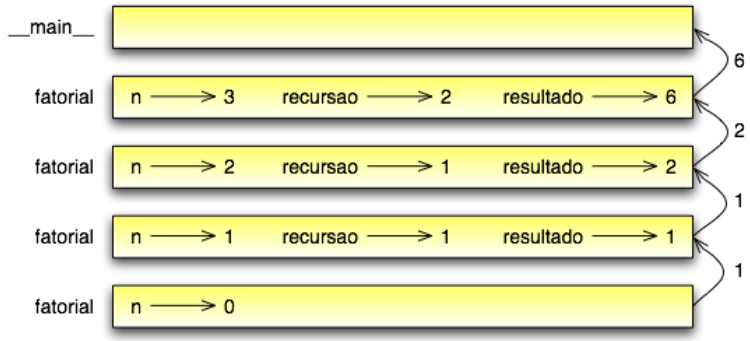
Por outro lado, e esta é a parte interessante, temos que fazer uma chamada recursiva para encontrar o fatorial de n-1 e então multiplicá-lo por n:

```
def fatorial(n):
  if n == 0:
    return 1
  else:
    recursivo = fatorial(n-1)
    resultado = n * recursivo
    return resultado
```

O fluxo de execução para este programa é similar ao fluxo de contagemRegressiva na Seção 4.9. Se chamarmos fatorial com o valor 3:
Já que 3 não é 0, tomamos o segundo ramo e calculamos o fatorial de n-1 ...
Já que 2 não é 0, tomamos o segundo ramo e calculamos o fatorial de n-1 ...
Já que 1 não é 0, tomamos o segundo ramo e calculamos o fatorial de n-1 ...
Já que 0 *é* 0, tomamos o primeiro ramo e retornamos 1 sem fazer mais qualquer chamada recursiva.

O valor retornado (1) é multiplicado por n, que é 1, e o resultado é retornado.
O valor retornado (1) é multiplicado por n, que é 2, e o resultado é retornado.
O valor retornado (2) é multiplicado por n, que é 1, e o resultado, 6, se torna o valor de retorno da chamada de função que iniciou todo o processo.

Eis o diagrama de pilha para esta sequência de chamadas de função:



Os valores de retorno são mostrados sendo passados de volta para cima da pilha. Em cada quadro, o valor de retorno é o valor de resultado, o qual é o produto de n por recursivo.

5.6 Voto de confiança (Leap of faith)

Seguir o fluxo de execução é uma maneira de ler programas, mas que pode rapidamente se transformar em um labirinto. Uma alternativa é o que chamamos de “voto de confiança”.

Quando você tem uma chamada de função, em vez de seguir o fluxo de execução, você *assume* que a função funciona corretamente e retorna o valor apropriado.

De fato, você está agora mesmo praticando este voto de confiança ao usar as funções nativas. Quando você chama math.cos ou math.exp, você não examina a implementação destas funções. Você apenas assume que elas funcionam porque as pessoas que escreveram as bibliotecas nativas eram bons programadores.

O mesmo também é verdade quando você chama uma de suas próprias funções. Por exemplo, na Seção 5.4, escrevemos a função chamada ehDivisivel que determina se um número é divisível por outro. Uma vez que nos convencemos que esta função está correta – ao testar e examinar o código – podemos usar a função sem examinar o código novamente.

O mesmo também é verdadeiro para programas recursivos. Quando você tem uma chamada recursiva, em vez de seguir o fluxo de execução, você poderia assumir que a chamada recursiva funciona (produz o resultado correto) e então perguntar-se, “Assumindo que eu possa encontrar o fatorial de *n-1*, posso calcular o fatorial de *n*?” Neste caso, é claro que você pode, multiplicando por *n*.

Naturalmente, é um pouco estranho que uma função funcione corretamente se você ainda nem terminou de escrevê-la, mas é por isso que se chama voto de confiança.

5.7 Mais um exemplo

No exemplo anterior, usamos variáveis temporárias para deixar claros os passos e tornar o código mais fácil de depurar, mas poderíamos ter economizado algumas linhas:

```
def fatorial(n):
  if n == 0:
    return 1
  else:
    return n * fatorial(n-1)
```

De agora em diante, tenderemos a utilizar um formato mais conciso, mas recomendamos que você use a versão mais explícita enquanto estiver desenvolvendo código. Quando ele estiver funcionando, você pode enxugá-lo se estiver se sentindo inspirado.

Depois de fatorial, o exemplo mais comum de uma função matemática definida recursivamente é fibonacci, a qual tem a seguinte definição:

```
fibonacci(0) = 1
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);
```

Traduzido em Python, parecerá assim:

```
def fibonacci(n):
  if n == 0 or n == 1:
    return 1
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

Se você tentar seguir o fluxo de execução aqui, mesmo para valores bem pequenos de n, sua cabeça explodirá. Mas, de acordo com o voto de confiança, se você assume que as duas chamadas recursivas funcionam corretamente, então é claro que você terá o resultado correto ao juntá-las.

5.8 Checagem de tipos

O que acontece se chamamos fatorial e damos a ela 1.5 como argumento?:

```
>>> fatorial (1.5)
```

```
RuntimeError: Maximum recursion depth exceeded
```

Parece um caso de recursividade infinita. Mas o que será que é de fato? Existe um caso base – quando n == 0. O problema é que o valor de n *nunca encontra* o caso base.

Na primeira chamada recursiva, o valor de n é 0.5. Na próxima, ele é igual a -0.5. Daí em diante, ele se torna cada vez menor, mas jamais será 0.

Temos então duas alternativas. Podemos tentar generalizar a função fatorial para que funcione com números em ponto flutuante, ou fazemos fatorialrealizar a checagem de tipo de seus parâmetros. A primeira é chamada função gamma e está um pouco além do escopo deste livro. Sendo assim, ficaremos com a segunda.

Podemos usar type para comparar o tipo do parâmetro com o tipo de um valor inteiro conhecido (como 1). Ao mesmo tempo em que fazemos isto, podemos nos certificar também de que o parâmetro seja positivo:

```
def fatorial (n):
  if type(n) != type(1):
    print "Fatorial somente é definido para inteiros."
    return -1
  elif n < 0:
    print "Fatorial somente é definido para inteiros positivos."
    return -1
  elif n ==0:
    return 1
  else:
    return n * fatorial(n-1)
```

Agora temos três casos base. O primeiro pega os não-inteiros. O segundo pega os inteiros negativos. Em ambos os casos, o programa exibe uma mensagem de erro e retorna um valor especial, -1, para indicar que alguma coisa saiu errada:

```
>>> fatorial ("Fred")
Fatorial somente é definido para inteiros.
-1
```

```
>>> fatorial (-2)
Fatorial somente é definido para inteiros positivos.
-1
```

Se passarmos pelas duas checagens, então saberemos que n é um inteiro positivo, e poderemos provar que a recursividade encontra seu término.

Este programa demonstra um padrão (*pattern*) chamado às vezes de **guardião**. As duas primeiras condicionais atuam como guardiãs, protegendo o código que vem em seguida de valores que poderiam causar um erro. Os guardiões tornam possível garantir a correção do código.

5.9 Glossário

reatribuição (*multiple assignment*)
quando mais de um valor é atribuído a mesma variável durante a execução do programa.

> N.T.: O termo *multiple assignment* (ou atribuição múltipla) é usado com mais frequência para descrever a sintaxe `a = b = c`. Por este motivo optamos pelo termo reatribuição no contexto da seção 6.1 desse capítulo.

iteração (*iteration*)
execução repetida de um conjunto de comandos/instruções (statements) usando uma chamada recursiva de função ou um laço (loop).

laço (*loop*)
um comando/instrução ou conjunto de comandos/instruções que executam repetidamente até que uma condição de interrupção seja atingida.

laço infinito (*infinite loop*)
um laço em que a condição de interrupção nunca será atingida.

corpo (*body*)
o conjunto de comandos/instruções que pertencem a um laço.

variável de laço (*loop variable*)
uma variável usada como parte da condição de interrupção do laço.

tabulação (*tab*)
um carácter especial que faz com que o cursor mova-se para a próxima parada estabelecida de tabulação na linha atual.

nova-linha (*newline*)
um carácter especial que faz com que o cursor mova-se para o início da próxima linha.

cursor (*cursor*)
um marcador invisível que determina onde o próximo carácter var ser impresso.

sequência de escape (*escape sequence*)
um carácter de escape () seguido por um ou mais caracteres imprimíveis, usados para definir um carácter não imprimível.

encapsular (*encapsulate*)
quando um programa grande e complexo é dividido em componentes (como funções) e estes são isolados um do outro (pelo uso de variáveis locais, por exemplo).

generalizar (*generalize*)
quando algo que é desnecessariamente específico (como um valor constante) é substituído por algo apropriadamente geral (como uma variável ou um parâmetro). Generalizações dão maior versatilidade ao código, maior possibilidade de reuso, e em algumas situações até mesmo maior facilidade para escrevê-lo.

plano de desenvolvimento (*development plan*)
um processo definido para desenvolvimento de um programa. Neste capítulo, nós demonstramos um estilo de desenvolvimento baseado em escrever código para executar tarefas simples e específicas, usando encapsulamento e generalização.

Capítulo 6: Interação

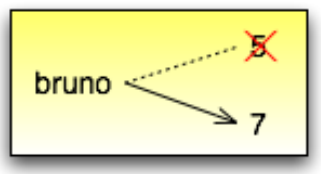
6.1 Reatribuições

Como você talvez já tenha descoberto, é permitido fazer mais de uma atribuição à mesma variável. Uma nova atribuição faz uma variável existente referir-se a um novo valor (sem se referir mais ao antigo).:

```
bruno = 5
print bruno,
bruno = 7
print bruno
```

A saída deste programa é `5 7`, porque na primeira vez que `bruno` é impresso, seu valor é 5 e na segunda vez, seu valor é 7. A vírgula no final do primeiro comando `print` suprime a nova linha no final da saída, que é o motivo pelo qual as duas saídas aparecem na mesma linha.

Veja uma **reatribuição** em um diagrama de estado:



Com a reatribuição torna-se ainda mais importante distinguir entre uma operação de atribuição e um comando de igualdade. Como Python usa o sinal de igual (`=`) para atribuição, existe a tendência de lermos um comando como `a = b` como um comando de igualdade. Mas não é!

Em primeiro lugar, igualdade é comutativa e atribuição não é. Por exemplo, em matemática, se a = 7 então 7 = a. Mas em Python, o comando `a = 7` é permitido e `7 = a` não é.

Além disso, em matemática, uma expressão de igualdade é sempre verdadeira. Se a = b agora, então, a será sempre igual a b. Em Python, um comando de atribuição pode tornar duas variáveis iguais, mas elas não têm que permanecer assim:

```
a = 5
b = a # a e b agora são iguais
b = 3 # a e b não são mais iguais
```

A terceira linha muda o valor de `a` mas não muda o valor de `b`, então, elas não são mais iguais. (Em algumas linguagens de programação, um símbolo diferente é usado para atribuição, como `<-` ou `:=`, para evitar confusão.)

Embora a reatribuição seja freqüentemente útil, você deve usá-la com cautela. Se o valor das variáveis muda freqüentemente, isto pode fazer o código difícil de ler e de depurar.

6.2 O comando while

Os computadores são muito utilizados para automatizar tarefas repetitivas. Repetir tarefas idênticas ou similares sem cometer erros é uma coisa que os computadores fazem bem e que as pessoas fazem poorly.

Vimos dois programas, `nLinhas` e `contagemRegressiva`, que usam recursividade (recursão) para fazer a repetição, que também é chamada **iteração**. Porque a iteração é muito comum, Python tem várias características para torná-la mais fácil. A primeira delas em que vamos dar uma olhada é o comando `while`.

Aqui está como fica `contagemRegressiva` com um comando `while`:

```
def contagemRegressiva(n):
  while n > 0:
    print n
    n = n-1
  print "Fogo!"
```

Desde que removemos a chamada recursiva, esta função não é recursiva.

Você quase pode ler o comando `while` como se fosse Inglês. Ele significa, “Enquanto (while) `n` for maior do que `0`, siga exibindo o valor de `n` e diminuindo `1` do valor de `n`. Quando chegar a `0`, exiba a palavra `Fogo!`“.

Mais formalmente, aqui está o fluxo de execução para um comando `while`:

1. Teste a condição, resultando 0 ou 1.
2. Se a condição for falsa (0), saia do comando while e continue a execução a partir do próximo comando.
3. Se a condição for verdadeira (1), execute cada um dos comandos dentro do corpo e volte ao passo 1.

O corpo consiste de todos os comandos abaixo do cabeçalho, com a mesma endentação.

Este tipo de fluxo é chamado de um **loop** (ou laço) porque o terceiro passo cria um “loop” ou um laço de volta ao topo. Note que se a condição for falsa na primeira vez que entrarmos no loop, os comandos dentro do loop jamais serão executados.

O corpo do loop poderia alterar o valor de uma ou mais variáveis de modo que eventualmente a condição se torne falsa e o loop termine. Se não for assim, o loop se repetirá para sempre, o que é chamado de um **loop infinito**. Uma fonte de diversão sem fim para os cientistas da computação é a observação de que as instruções da embalagem de shampoo, “Lave, enxágüe, repita” é um loop infinito.

No caso de `contagemRegressiva`, podemos provar que o loop terminará porque sabemos que o valor de n é finito, e podemos ver que o valor de n diminui dentro de cada repetição (iteração) do loop, então, eventualmente chegaremos ao 0. Em outros casos, isto não é tão simples de afirmar:

```
def sequencia(n):
  while n != 1:
    print n,
    if n%2 == 0:          # n é par
      n = n/2
    else:                 # n é impar
      n = n*3+1
```

A condição para este loop é `n != 1`, então o loop vai continuar até que `n` seja `1`, o que tornará a condição falsa.

Dentro de cada repetição (iteração) do loop, o programa gera o valor de `n` e então checa se ele é par ou impar. Se ele for par, o valor de `n` é dividido por `2`. Se ele for impar, o valor é substituído por `n*3+1`. Por exemplo, se o valor inicial (o argumento passado para `seqüência`) for 3, a seqüência resultante será `3`, `10`, `5`, `16`, `8`, `4`, `2`, `1`.

Já que `n` às vezes aumenta e às vezes diminui, não existe uma prova óbvia de que `n` jamais venha a alcançar 1, ou de que o programa termine. Para alguns valores particulares de `n`, podemos provar o término. Por exemplo, se o valor inicial for uma potência de dois, então o valor de `n` será par dentro de cada repetição (iteração) do loop até que alcance 1. O exemplo anterior termina com uma dessas seqüências começando em 16.

Valores específicos à parte, A questão interessante é se há como provarmos que este programa termina para todos os valores de `n`. Até hoje, ninguém foi capaz de provar que sim ou que não!

Como um exercício, reescreva a função `nLinhas` da seção 4.9 usando iteração em vez de recursão.

6.3 Tabelas

Uma das coisas para qual os loops são bons é para gerar dados tabulares. Antes que os computadores estivessem readily disponíveis, as pessoas tinham que calcular logaritmos, senos, cossenos e outras funções matemáticas à mão. Para tornar isto mais fácil, os livros de matemática continham longas tabelas listando os valores destas funções. Criar as tabelas era demorado e entediante, e elas tendiam a ser cheias de erros.

Quando os computadores entraram em cena, uma das reações iniciais foi “Isto é ótimo! Podemos usar computadores para geras as tabelas, assim não haverá erros.” Isto veio a se tornar verdade (na maioria das vezes) mas shortsighted. Rapidamente, porém, computadores e calculadoras tornaram-se tão pervasivos que as tabelas ficaram obsoletas.

Bem, quase. Para algumas operações, os computadores usam tabelas de valores para conseguir uma resposta aproximada e então realizar cálculos para melhorar a aproximação. Em alguns casos, têm havido erros nas tabelas underlying, o caso mais famoso sendo o da tabela usada pelo processador Pentium da Intel para executar a divisão em ponto-flutuante.

Embora uma tabela de logaritmos não seja mais tão útil quanto já foi um dia, ela ainda dá um bom exemplo de iteração. O seguinte programa gera uma sequência de valores na coluna da esquerda e seus respectivos logaritmos na coluna da direita:

```
x = 1.0
while x < 10.0:
  print x, '\t', math.log(x)
  x = x + 1.0
```

A string `'\t'` representa um caracter de **tabulação**.

Conforme caracteres e strings vão sendo mostrados na tela, um ponteiro invisível chamado **cursor** marca aonde aparecerá o próximo caractere. Depois de um comando `print`, o cursor normalmente vai para o início de uma nova linha.

O caractere de tabulação desloca o cursor para a direita até que ele encontre uma das marcas de tabulação. Tabulação é útil para fazer colunas de texto line up, como na saída do programa anterior:

```
1.0   0.0
2.0   0.69314718056
3.0   1.09861228867
4.0   1.38629436112
5.0   1.60943791243
6.0   1.79175946923
7.0   1.94591014906
8.0   2.07944154168
9.0   2.19722457734
```

Se estes valores parecem odd, lembre-se que a função `log` usa a base `e`. Já que potências de dois são tão importantes em ciência da computação, nós frequentemente temos que achar logaritmos referentes à base 2. Para fazermos isso, podemos usar a seguinte fórmula:

(XXX diagramar fórmula matemática)

log2 x = loge x (6.1)
loge 2

Alterando o comando de saída para:

```
print x, '\t', math.log(x)/math.log(2.0)
```

o que resultará em:

```
1.0   0.0
2.0   1.0
3.0   1.58496250072
4.0   2.0
5.0   2.32192809489
6.0   2.58496250072
7.0   2.80735492206
8.0   3.0
9.0   3.16992500144
```

Podemos ver que 1, 2, 4 e 8 são potências de dois porque seus logaritmos na base 2 são números redondos. Se precisássemos encontrar os logaritmos de outras potências de dois, poderíamos modificar o programa deste modo:

```
x = 1.0
while x < 100.0:
  print x, '\t', math.log(x)/math.log(2.0)
  x = x * 2.0
```

Agora, em vez de somar algo a x a cada iteração do loop, o que resulta numa sequência aritmética, nós multiplicamos x por algo, resultando numa sequência geométrica. O resultado é:

```
1.0   0.0
2.0   1.0
4.0   2.0
8.0   3.0
16.0  4.0
32.0  5.0
64.0  6.0
```

Por causa do caractere de tabulação entre as colunas, a posição da segunda coluna não depende do número de dígitos na primeira coluna.

Tabelas de logaritmos podem não ser mais úteis, mas para cientistas da computação, conhecer as potências de dois é!

Como um exercício, modifique este programa de modo que ele produza as potências de dois acima de 65.535 (ou seja, 216). Imprima e memorize-as.

O caractere de barra invertida em `'\t'` indica o início de uma sequência de escape. Sequências de escape são usadas para representar caracteres invisíveis como de tabulação e de nova linha. A sequência `\n` representa uma nova linha.

Uma sequência de escape pode aparecer em qualquer lugar em uma string; no exemplo, a sequência de escape de tabulação é a única coisa dentro da string.

Como você acha que se representa uma barra invertida em uma string?

Como um exercício, escreva um única string que

produza
esta
saída.

6.4 Tabelas de duas dimensões (ou bi-dimensionais)

Uma tabela de duas dimensões é uma tabela em que você lê o valor na interseção entre uma linha e uma coluna. Uma tabela de multiplicação é um bom exemplo. Digamos que você queira imprimir uma tabela de multiplicação de 1 a 6.

Uma boa maneira de começar é escrever um loop que imprima os múltiplos de 2, todos em uma linha:

```
i = 1
while i <= 6:
  print 2*i, '   ',
  i = i + 1
print
```

A primeira linha inicializa a variável chamada `i`, a qual age como um contador ou **variável de controle do loop**. Conforme o loop é executado, o valor de `i` é incrementado de 1 a 6. Quando `i` for 7, o loop termina. A cada repetição (iteração) do loop, é mostrado o valor de `2*i`, seguido de três espaços.

De novo, a vírgula no comando `print` suprime a nova linha. Depois que o loop se completa, o segundo comando `print` inicia uma nova linha.

A saída do programa é:

```
2    4    6    8    10    12
```

Até aqui, tudo bem. O próximo passo é **encapsular** e **generalizar**.

6.5 Encapsulamento e generalização

Encapsulamento é o processo de wrapping um pedaço de código em uma função, permitindo que você tire vantagem de todas as coisas para as quais as funções são boas. Você já viu dois exemplos de encapsulamento: `imprimeParidade` na seção 4.5; e `eDivisivel` na seção 5.4

Generalização significa tomar algo que é específico, tal como imprimir os múltiplos de 2, e torná-lo mais geral, tal como imprimir os múltiplos de qualquer inteiro.

Esta função encapsula o loop anterior e generaliza-o para imprimir múltiplos de n:

```
def imprimeMultiplos(n):
i = 1
while i <= 6:
  print n*i, '\t ',
  i = i + 1
print
```

Para encapsular, tudo o que tivemos que fazer foi adicionar a primeira linha, que declara o nome de uma função e sua lista de parâmetros. Para generalizar, tudo o que tivemos que fazer foi substituir o valor 2 pelo parâmetro `n`.

Se chamarmos esta função com o argumento 2, teremos a mesma saída que antes. Com o argumento `3`, a saída é:

```
3    6    9    12    15    18
```

Com o argumento `4`, a saída é:

```
4   8   12   16   20   24
```

Agora você provavelmente pode adivinhar como imprimir uma tabela de multiplicação - chamando `imprimeMultiplos` repetidamente com argumentos diferentes. De fato, podemos usar um outro loop:

```
i = 1
while i <= 6:
  imprimeMultiplos(i)
  i = i + 1
```

Note o quanto este loop é parecido com aquele dentro de `imprimeMultiplos`. Tudo o que fiz foi substituir o comando `print` pela chamada à função.

A saída deste programa é uma tabela de multiplicação:

```
1   2   3   4   5   6
2   4   6   8   10  12
3   6   9   12  15  18
4   8   12  16  20  24
5   10  15  20  25  30
6   12  18  24  30  36
```

6.6 Mais encapsulamento

Para demonstrar de novo o encapsulamento, vamos pegar o código do final da seção 6.5 e acondicioná-lo, envolvê-lo em uma função:

```
def imprimeTabMult():
  i = 1
  while i <= 6:
    imprimeMultiplos(i)
    i = i + 1
```

Este processo é um **plano de desenvolvimento** comum. Nós desenvolvemos código escrevendo linhas de código fora de qualquer função, ou digitando-as no interpretador. Quando temos o código funcionando, extraímos ele e o embalamos em uma função.

Este plano de desenvolvimento é particularmente útil se você não sabe, quando você começa a escrever, como dividir o programa em funções. Esta técnica permite a você projetar enquanto desenvolve.

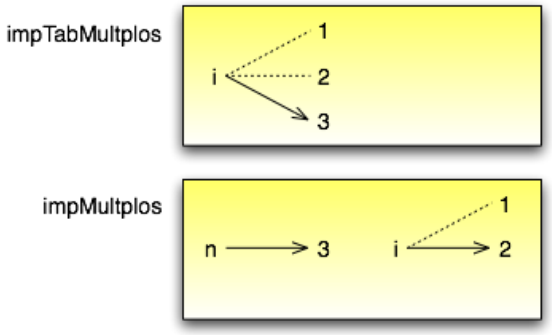
6.7 Variáveis locais

Você pode estar pensando como podemos utilizar a mesma variável, i, em ambos, imprimeMultiplos e imprimeTabMult. Isto não causaria problemas quando uma das funções mudasse o valor da variável?

A resposta é não, porque o i em imprimeMultiplos e o i em imprimeTabMult não são a mesma variável.

Variáveis criadas dentro de uma definição de função são locais; você não pode acessar uma variável local de fora da função em que ela "mora". Isto significa que você é livre para ter múltiplas variáveis com o mesmo nome, desde que elas não estejam dentro da mesma função.

O diagrama de pilha para este programa mostra que duas variáveis chamadas i não são a mesma variável. Elas podem se referir a valores diferentes e alterar o valor de uma não afeta à outra.



O valor de i em imprimeTabMult vai de 1 a 6. No diagrama, i agora é 3. Na próxima iteração do loop i será 4. A cada iteração do loop, imprimeTabMultchama imprimeMultiplos com o valor corrente de i como argumento. O valor é atribuído ao parâmetro n.

Dentro de imprimeMultiplos, o valor de i vai de 1 a 6. No diagrama, i agora é 2. Mudar esta variável não tem efeito sobre o valor de i em imprimeTabMult.

É comum e perfeitamente legal ter variáveis locais diferentes com o mesmo nome. Em particular, nomes como i e j são muito usados para variáveis de controle de loop. Se você evitar utilizá-los em uma função só porque você já os usou em outro lugar, você provavelmente tornará seu programa mais difícil de ler.

6.8 Mais generalização

Como um outro exemplo de generalização, imagine que você precise de um programa que possa imprimir uma tabela de multiplicação de qualquer tamanho, não apenas uma tabela de seis por seis. Você poderia adicionar um parâmetro a imprimeTabMult:

```
def imprimeTabMult(altura):
  i = 1
  while i <= altura:
    imprimeMultiplos(i)
    i = i + 1
```

Nós substituímos o valor 6 pelo parâmetro altura. Se chamarmos imprimeTabMult com o argumento 7, ela mostra:

```
1    2     3     4     5     6
2    4     6     8     10    12
3    6     9     12    15    18
4    8     12    16    20    24
5    10    15    20    25    30
6    12    18    24    30    36
7    14    21    28    35    42
```

Isto é bom, exceto que nós provavelmente quereríamos que a tabela fosse quadrada - com o mesmo número de linhas e colunas. Para fazer isso, adicionamos outro parâmetro a imprimeMultiplos para especificar quantas colunas a tabela deveria ter.

Só para confundir, chamamos este novo parâmetro de altura, demonstrando que diferentes funções podem ter parâmetros com o mesmo nome (como acontece com as variáveis locais). Aqui está o programa completo:

```
def imprimeMultiplos(n, altura):
  i = 1
  while i  <= altura:
    print n*i, 't',
    i = i + 1
  print

def imprimeTabMult(altura):
  i = 1
  while i <= altura:
  imprimeMultiplos(i, altura)
  i = i + 1
```

Note que quando adicionamos um novo parâmetro, temos que mudar a primeira linha da função (o cabeçalho da função), e nós também temos que mudar o lugar de onde a função é chamada em imprimeTabMult.

Como esperado, este programa gera uma tabela quadrada de sete por sete:

```
1    2     3     4     5     6     7
2    4     6     8     10    12    14
3    6     9     12    15    18    21
4    8     12    16    20    24    28
5    10    15    20    25    30    35
6    12    18    24    30    36    42
```

```
7    14    21    28    35    42    49
```

Quando você generaliza uma função apropriadamente, você muitas vezes tem um programa com capacidades que você não planejou. Por exemplo, você pode ter notado que, porque ab = ba, todas as entradas na tabela aparecem duas vezes. Você poderia economizar tinta imprimindo somente a metade da tabela. Para fazer isso, você tem que mudar apenas uma linha em `imprimeTabMult`. Mude:

```
imprimeTabMult(i, altura)
```

para:

```
imprimeTabMult(i, i)
```

e você terá:

```
1
2    4
3    6    9
4    8    12    16
5    10    15    20    25
6    12    18    24    30    36
7    14    21    28    35    42    49
```

Como um exercício, trace a execução desta versão de `imprimeTabMult` e explique como ela funciona.

6.9 Funções

- Há pouco tempo mencionamos “todas as coisas para as quais as funções são boas.” Agora, você pode estar pensando que coisas exatamente são estas. Aqui estão algumas delas:
- Dar um nome para uma seqüência de comandos torna seu programa mais fácil de ler e de depurar.
- Dividir um programa longo em funções permite que você separe partes do programa, depure-as isoladamente, e então as componha em um todo.
- Funções facilitam tanto recursão quanto iteração.
- Funções bem projetadas são freqüentemente úteis para muitos programas. Uma vez que você escreva e depure uma, você pode reutilizá-la.

6.10 Glossário

reatribuição (*multiple assignment* [1])

quando mais de um valor é atribuído a mesma variável durante a execução do programa.

> N.T.: O termo *multiple assignment* (ou atribuição múltipla) é usado com mais frequência para descrever a sintaxe `a = b = c`. Por este motivo optamos pelo termo reatribuição no contexto da seção 6.1 desse capítulo.

iteração (*iteration*)

execução repetida de um conjunto de comandos/instruções (statements) usando uma chamada recursiva de função ou um laço (loop).

laço (*loop*)
um comando/instrução ou conjunto de comandos/instruções que executam repetidamente até que uma condição de interrupção seja atingida.

laço infinito (*infinite loop*)

um laço em que a condição de interrupção nunca será atingida.

corpo (*body*)

o conjunto de comandos/instruções que pertencem a um laço.

variável de laço (*loop variable*)

uma variável usada como parte da condição de interrupção do laço.

tabulação (*tab*)

um carácter especial que faz com que o cursor mova-se para a próxima parada estabelecida de tabulação na linha atual.

nova-linha (*newline*)

um carácter especial que faz com que o cursor mova-se para o início da próxima linha.

cursor (*cursor*)

um marcador invisível que determina onde o próximo carácter var ser impresso.

sequência de escape (*escape sequence*)

um carácter de escape () seguido por um ou mais caracteres imprimíveis, usados para definir um carácter não imprimível.

encapsular (*encapsulate*)

quando um programa grande e complexo é dividido em componentes (como funções) e estes são isolados um do outro (pelo uso de variáveis locais, por exemplo).

generalizar (*generalize*)

quando algo que é desnecessariamente específico (como um valor constante) é substituído por algo apropriadamente geral (como uma variável ou um parâmetro). Generalizações dão maior versatilidade ao código, maior possibilidade de reuso, e em algumas situações até mesmo maior facilidade para escrevê-lo.

plano de desenvolvimento (*development plan*)

um processo definido para desenvolvimento de um programa. Neste capítulo, nós demonstramos um estilo de desenvolvimento baseado em escrever código para executar tarefas simples e específicas, usando encapsulamento e generalização.