Capítulo 4: Mais ferramentas de controle de fluxo

Além do comando `while` recém apresentado, Python tem as estruturas usuais de controle de fluxo conhecidas em outras linguagens, com algumas particularidades.

Comando `if`

Provavelmente o mais conhecido comando de controle de fluxo é o `if`. Por exemplo:

```
>>> x = int(raw_input("Favor digitar um inteiro: "))
Favor digitar um inteiro: 42
>>> if x < 0:
...      x = 0
...      print 'Negativo alterado para zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Unidade'
... else:
...      print 'Mais'
...
Mais
```

Pode haver zero ou mais seções `elif`, e a seção `else` é opcional. A palavra-chave `elif` é uma abreviação para 'else if', e é útil para evitar indentação excessiva. Uma sequência `if` ... `elif` ... `elif` ... substitui as construções `switch` ou `case` existentes em outras linguagens.

Comando `for`

O comando `for` em Python difere um tanto do que você talvez esteja acostumado em C ou Pascal. Ao invés de se iterar sobre progressões aritméticas (como em Pascal), ou dar ao usuário o poder de definir tanto o passo da iteração quanto a condição de parada (como em C), o comando `for` de Python itera sobre os itens de qualquer sequência (como uma lista ou uma string), na ordem em que eles aparecem na sequência. Por exemplo:

```
>>> # Medir o tamanho de algumas strings:
>>> a = ['gato', 'janela', 'defenestrar']
>>> for x in a:
...     print x, len(x)
...
gato 4
janela 6
defenestrar 11
>>>
```

Não é seguro modificar a sequência sobre a qual se baseia o laço de iteração (isto pode acontecer

se a sequência for mutável, isto é, uma lista). Se você precisar modificar a lista sobre a qual está iterando (por exemplo, para duplicar itens selecionados), você deve iterar sobre uma cópia da lista ao invés da própria. A notação de fatiamento é bastante conveniente para isso:

```
>>> for x in a[:]: # fazer uma cópia da lista inteira
...     if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrar', 'gato', 'janela', 'defenestrar']
```

A função `range`

Se você precisar iterar sobre sequências numéricas, a função embutida `range` é a resposta. Ela gera listas contendo progressões aritméticas, por exemplo:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

O ponto de parada fornecido nunca é incluído na lista; `range(10)` gera uma lista com 10 valores, exatamente os índices válidos para uma sequência de comprimento 10. É possível iniciar o intervalo em outro número, ou alterar a razão da progressão (inclusive com passo negativo):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Para iterar sobre os índices de uma sequência, combine `range` e `len` da seguinte forma:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Na maioria dos casos como este, porém, é mais conveniente usar a função `enumerate`, veja *Looping Techniques [Capítulo 5]*.

Comandos `break` e `continue`, e cláusulas `else` em laços

O comando `break`, como em C, interrompe o laço `for` ou `while` mais interno.

O comando `continue`, também emprestado de C, avança para a próxima iteração do laço mais interno.

Laços podem ter uma cláusula `else`, que é executada sempre que o laço se encerra por exaustão da lista (no caso do `for`) ou quando a condição se torna falsa (no caso do `while`), mas nunca quando o laço é interrompido por um `break`. Isto é exemplificado no próximo exemplo que procura números primos:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, '=', x, '*', n/x
...             break
...     else:
...         # laço terminou sem encontrar um fator
...         print n, 'é um número primo'
...
2 é um número primo
3 é um número primo
4 = 2 * 2
5 é um número primo
6 = 2 * 3
7 é um número primo
8 = 2 * 4
9 = 3 * 3
```

(Sim, este é o código correto. Olhe atentamente: a cláusula `else` pertence ao laço `for`, e **não** ao comando `if`.)

Comando `pass`

O comando `pass` não faz nada. Ela pode ser usada quando a sintaxe exige um comando mas a semântica do programa não requer nenhuma ação. Por exemplo:

```
>>> while True:
...     pass  # esperar interrupção via teclado (Ctrl+C)
...
```

Isto é usado muitas vezes para se definir classes mínimas:

```
>>> class MinhaClasseVazia:
...     pass
...
```

Outra situação em que `pass` pode ser usado é para reservar o lugar de uma função ou de um bloco condicional, quando você está trabalhando em código novo, o que lhe possibilita continuar a raciocinar em um nível mais abstrato. O comando `pass` é ignorado silenciosamente:

```
>>> def initlog(*args):
...     pass   # Lembrar de implementar isto!
...
```

Definindo Funções

Podemos criar uma função que escreve a série de Fibonacci até um limite arbitrário:

```
>>> def fib(n):    # escrever série de Fibonacci até n
...     """Exibe série de Fibonacci até n"""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Agora invocamos a funçao que acabamos de definir:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

A palavra reservada `def` inicia a *definição* de uma função. Ela deve ser seguida do nome da função e da lista de parâmetros formais entre parênteses. Os comandos que formam o corpo da função começam na linha seguinte e devem ser indentados.

Opcionalmente, a primeira linha lógica do corpo da função pode ser uma string literal, cujo propósito é documentar a função. Se presente, essa string chama-se **docstring**. (Há mais informação sobre docstrings na seção *Strings de documentação [neste capítulo]*.) Existem ferramentas que utilizam docstrings para produzir automaticamente documentação online ou para imprimir, ou ainda permitir que o usuário navegue interativamente pelo código. É uma boa prática incluir sempre docstrings em suas funções, portanto, tente fazer disto um hábito.

A *execução* de uma função gera uma nova tabela de símbolos, usada para as variáveis locais da função. Mais precisamente, toda atribuição a variável dentro da função armazena o valor na tabela de símbolos local. Referências a variáveis são buscadas primeiramente na tabela local, então na tabela de símbolos global e finalmente na tabela de nomes embutidos (built-in). Portanto, não se pode atribuir diretamente um valor a uma variável global dentro de uma função (a menos que se utilize a declaração `global` antes), ainda que variáveis globais possam ser referenciadas livremente.

Os parâmetros reais (argumentos) de uma chamada de função são introduzidos na tabela de símbolos local da função no momento da invocação, portanto, argumentos são passados por valor (onde o *valor* é sempre uma referência para objeto, não o valor do objeto). [1] Quando uma função invoca outra, uma nova tabela de símbolos é criada para tal chamada.

Uma definição de função introduz o nome da função na tabela de símbolos atual. O valor associado ao nome da função tem um tipo que é reconhecido pelo interpretador como uma função definida pelo usuário. Esse valor pode ser atribuído a outros nomes que também podem ser usados como funções. Esse mecanismo serve para renomear funções:

```
>>> fib
```

1 Na verdade, *passagem por referência para objeto* (*call by object reference*) seria uma descrição melhor do que *passagem por valor* (*call-by-value*), pois, se um objeto mutável for passado, o invocador (*caller*) verá as alterações feitas pelo invocado (*callee*), como por exemplo a inserção de itens em uma lista.

```
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Conhecendo outras linguagens, você pode questionar que `fib` não é uma função, mas um procedimento, pois ela não devolve um valor. Na verdade, mesmo funções que não usam o comando `return` devolvem um valor, ainda que pouco interessante. Esse valor é chamado `None` (é um nome embutido). O interpretador interativo evita escrever `None` quando ele é o único resultado de uma expressão. Mas se quiser vê-lo pode usar o comando `print`:

```
>>> fib(0)
>>> print fib(0)
None
```

É fácil escrever uma função que devolve uma lista de números série de Fibonacci, ao invés de exibi-los:

```
>>> def fib2(n): # devolve a série de Fibonacci até n
...     """Devolve uma lista a com série de Fibonacci até n."""
...     resultado = []
...     a, b = 0, 1
...     while a < n:
...         resultado.append(a)    # veja mais adiante
...         a, b = b, a+b
...     return resultado
...
>>> f100 = fib2(100)    # executar
>>> f100                # exibir o resultado
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Este exemplo, como sempre, demonstra algumas características novas:

- O comando `return` termina a função devolvendo um valor. Se não houver uma expressão após o `return`, o valor `None` é devolvido. Se a função chegar ao fim sem o uso explícito do `return`, então também será devolvido o valor `None`.
- O trecho `resultado.append(a)` invoca um *método* do objeto lista `resultado`. Um método é uma função que "pertence" a um objeto e é chamada através de `obj.nome_do_metodo` onde `obj` é um objeto qualquer (pode ser uma expressão), e `nome_do_metodo` é o nome de um método que foi definido pelo tipo do objeto. Tipos diferentes definem métodos diferentes. Métodos de diferentes tipos podem ter o mesmo nome sem ambiguidade. (É possível definir seus próprios tipos de objetos e métodos, utilizando *classes*, veja em *Classes [Capítulo 9]*) O método `append` mostrado no exemplo é definido para objetos do tipo lista; ele adiciona um novo elemento ao final da lista. Neste exemplo, ele equivale a `resultado = resultado + [a]`, só que mais eficiente.

Mais sobre definição de funções

É possível definir funções com um número variável de argumentos. Existem três formas, que podem ser combinadas.

Parâmetros com valores default

A mais útil das três é especificar um valor default para um ou mais parâmetros formais. Isso cria uma função que pode ser invocada com um número menor de argumentos do que ela pode receber. Por exemplo:

```
def confirmar(pergunta, tentativas=4, reclamacao='Sim ou não, por favor!'):
    while True:
        ok = raw_input(pergunta).lower()
        if ok in ('s', 'si', 'sim'):
            return True
        if ok in ('n', 'no', 'não', 'nananinanão'):
            return False
        tentativas = tentativas - 1
        if tentativas == 0:
            raise IOError('usuario nao quer cooperar')
        print reclamacao
```

Essa função pode ser invocada de várias formas:

- fornecendo apenas o argumento obrigatório: `confirmar('Deseja mesmo encerrar?')`
- fornecendo um dos argumentos opcionais: `confirmar('Sobrescrever o arquivo?', 2)`
- ou fornecendo todos os argumentos: `confirmar('Sobrescrever o arquivo?', 2, 'Escolha apenas s ou n')`

Este exemplo também introduz o operador `in`, que verifica se uma sequência contém ou não um determinado valor.

Os valores default são avaliados no momento a definição da função, e no escopo em que a função foi *definida*, portanto:

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

irá exibir `5`.

Aviso importante: Valores default são avaliados apenas uma vez. Isso faz diferença quando o valor default é um objeto mutável como uma lista ou dicionário (N.d.T. dicionários são como arrays associativos ou HashMaps em outras linguagens; ver *Dictionaries [Capítulo 5]*).

Por exemplo, a função a seguir acumula os argumentos passados em chamadas subsequentes:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

Esse código vai exibir:

```
[1]
[1, 2]
[1, 2, 3]
```

Se você não quiser que o valor default seja compartilhado entre chamadas subsequentes, pode reescrever a função assim:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Argumentos nomeados

Funções também podem ser chamadas passando **keyword arguments** (argumentos nomeados) no formato `chave=valor`. Por exemplo, a seguinte função:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

aceita um argumento obrigatório (`voltage`) e três argumentos opcionais (`state`, `action`, e `type`). Esta função pode ser invocada de todas estas formas:

```
parrot(1000)                                    # 1 arg. posicional
parrot(voltage=1000)                            # 1 arg. nomeado
parrot(voltage=1000000, action='VOOOOOM')       # 2 arg. nomeados
parrot(action='VOOOOOM', voltage=1000000)       # 2 arg. nomeados
parrot('a million', 'bereft of life', 'jump')   # 3 arg. posicionais
# 1 arg. positional e 1 arg. nomeado
parrot('a thousand', state='pushing up the daisies')
```

mas todas as invocações a seguir seriam inválidas:

```
parrot()                     # argumento obrigatório faltando
parrot(voltage=5.0, 'dead')  # argumento posicional depois do nomeado
parrot(110, voltage=220)     # valur duplicado para o mesmo argument
parrot(actor='John Cleese')  # argumento nomeado desconhecido
```

Em uma invocação, argumentos nomeados devem vir depois dos argumentos posicionais. Todos os argumentos nomeados passados devem casar com os parâmetros formais definidos pela função (ex. `actor` não é um argumento nomeado válido para a função `parrot`), mas sua ordem é irrelevante. Isto também inclui argumentos obrigatórios (ex.: `parrot(voltage=1000)` funciona). Nenhum parâmetro pode receber mais de um valor. Eis um exemplo que não funciona devido a esta restrição:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Quando o último parâmetro formal usar a sintaxe `**nome`, ele receberá um dicionário (ver *Dictionaries [Capítulo 5]* ou Mapping Types — dict [online]) com todos os parâmetros nomeados passados para a função, exceto aqueles que corresponderam a parâmetros formais definidos antes. Isto pode ser combinado com o parâmetro formal `*nome` (descrito na próxima subseção) que recebe uma tupla (N.d.T. uma sequência de itens, semelhante a uma lista imutável; ver *Tuples and Sequences [Capítulo 5]*) contendo todos argumentos posicionais que não correspondem à lista da parâmetros formais. (`*nome` deve ser declarado antes de `**nome`.) Por exemplo, se definimos uma função como esta:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments:
        print arg
    print "-" * 40
    keys = sorted(keywords.keys())
    for kw in keys:
        print kw, ":", keywords[kw]
```

Ela pode ser invocada assim:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

e, naturalmente, produziria:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Note que criamos uma lista de chaves `keys` ordenando o resultado do método `keys()` do dicionário `keywords` antes de exibir seu conteúdo; se isso não fosse feito, os argumentos seriam exibidos em uma ordem não especificada.

Listas arbitrárias de argumentos

Finalmente, a opção menos usada possibilita que função seja invocada com um número arbitrário de argumentos. Esses argumentos serão empacotados em uma tupla (ver *Tuples and Sequences [Capítulo 5]*). Antes dos argumentos em número variável, zero ou mais argumentos normais podem estar presentes.

```
def escrever_multiplos_items(arquivo, separador, *args):
    arquivo.write(separador.join(args))
```

Desempacotando listas de argumentos

A situação inversa ocorre quando os argumentos já estão numa lista ou tupla mas ela precisa ser explodida para invocarmos uma função que requer argumentos posicionais separados. Por exemplo, a função `range` espera argumentos separados, *start* e *stop*. Se os valores já estiverem juntos em uma lista ou tupla, escreva a chamada de função com o operador `*` para desempacotá-los da sequência:

```
 >>> range(3, 6)      # chamada normal com argumentos separados
 [3, 4, 5]
 >>> args = [3, 6]
 >>> range(*args)     # chamada com argumentos desempacotados de uma lista
 [3, 4, 5]

statement: **
```

Da mesma forma, dicionários podem produzir argumentos nomeados com o operador `**`:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!"
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action":
"VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's
bleedin' demised !
```

Construções lambda

Atendendo a pedidos, algumas características encontradas em linguagens de programação funcionais como Lisp foram adicionadas a Python. Com a palavra reservada `lambda`, pequenas funções anônimas podem ser criadas. Eis uma função que devolve a soma de seus dois argumentos: `lambda a, b: a+b`.

Construções lambda podem ser empregadas em qualquer lugar que exigiria uma função. Sintaticamente, estão restritas a uma única expressão. Semanticamente, são apenas açúcar sintático para a definição de funções normais. Assim como definições de funções aninhadas, construções lambda podem referenciar variáveis do escopo onde são definidas (N.d.T isso significa que Python implementa *closures*, recurso encontrado em Lisp, JavaScript, Ruby etc.):

```
>>> def fazer_incrementador(n):
...     return lambda x: x + n
...
>>> f = fazer_incrementador(42)
>>> f(0)
42
>>> f(1)
43
```

Strings de documentação

A comunidade Python está convencionando o conteúdo e o formato de strings de documentação (*docstrings*).

A primeira linha deve ser um resumo curto e conciso do propósito do objeto. Por brevidade, não deve explicitamente se referir ao nome ou tipo do objeto, uma vez que estas informações estão disponíveis por outros meios (exceto se o nome da função for o próprio verbo que descreve a finalidade da função). Essa linha deve começar com letra maiúscula e terminar com ponto.

Se existem mais linhas na string de documentação, a segunda linha deve estar em branco, separando visualmente o resumo do resto da descrição. As linhas seguintes devem conter um ou mais parágrafos descrevendo as convenções de chamada ao objeto, seus efeitos colaterais, etc.

O parser do Python não remove a indentação de comentários multi-linha. Portanto, ferramentas que processem strings de documentação precisam lidar com isso, quando desejável. Existe uma convenção para isso. A primeira linha não vazia após a linha de sumário determina a indentação para o resto da string de documentação. (Não podemos usar a primeira linha para isso porque ela em geral está adjacente às aspas que iniciam a string, portanto sua indentação real não fica aparente.) Espaços em branco ou tabs "equivalentes" a esta indentação são então removidos do início das demais linhas da string. Linhas indentação menor não devem ocorrer, mas se ocorrerem, todos os espaços à sua esquerda são removidos. Para determinar a indentação, normalmente considera-se que um caractere tab equivale a 8 espaços.

Eis um exemplo de uma docstring multi-linha:

```
>>> def minha_funcao():
...     """Não faz nada, mas é documentada.
...
...     Realmente ela não faz nada.
...     """
...     pass
...
>>> print minha_funcao.__doc__
```

```
Não faz nada, mas é documentada.

    Realmente ela não faz nada.
```

Intermezzo: estilo de codificação

Agora que você está prestes a escrever peças mais longas e complexas em Python, é uma bom momento para falar sobre *estilo de codificação*. A maioria das linguagens podem ser escritas (ou *formatadas*) em diferentes estilos; alguns são mais legíveis do que outros. Tornar o seu código mais fácil de ler, para os outros, é sempre uma boa ideia, e adotar um estilo de codificação agradável ajuda bastante.

Em Python, o PEP 8 tornou-se o guia de estilo adotado pela maioria dos projetos; ele promove um estilo de codificação muito legível e visualmente agradável. Todo desenvolvedor Python deve lê-lo em algum momento; aqui estão os pontos mais importantes selecionados para você:

- Use 4 espaços de recuo, e nenhum tab.
  4 espaços são um bom meio termo entre indentação estreita (permite maior profundidade de aninhamento) e indentação larga (mais fácil de ler). Tabs trazem complicações; é melhor não usar.
- Quebre as linhas de modo que não excedam 79 caracteres.
  Isso ajuda os usuários com telas pequenas e torna possível abrir vários arquivos de código lado a lado em telas maiores.
- Deixe linhas em branco para separar as funções e classes, e grandes blocos de código dentro de funções.
- Quando possível, coloque comentários em uma linha própria.
- Escreva docstrings.
- Use espaços ao redor de operadores e após vírgulas, mas não diretamente dentro de parênteses, colchetes e chaves: `a = f(1, 2) + g(3, 4)`.
- Nomeie suas classes e funções de modo consistente; a convenção é usar `CamelCase` (literalmente, *CaixaCamelo*) para classes e `caixa_baixa_com_underscores` para funções e métodos. Sempre use `self` como nome do primeiro parâmetro formal dos métodos de instância (veja *A First Look at Classes [Capítulo 9]* para saber mais sobre classes e métodos).
- Não use codificações exóticas se o seu código é feito para ser usado em um contexto internacional. ASCII puro funciona bem em qualquer caso. (N.d.T. para programadores de língua portuguesa, UTF-8 é atualmente a melhor opção, e já se tornou o default em Python 3 conforme o PEP 3120).

Notas