Capítulo 5: Condicionais e recursividade

O tópico principal deste capítulo é a instrução if, que executa códigos diferentes dependendo do estado do programa. Mas primeiro quero apresentar dois novos operadores: divisão pelo piso e módulo.

5.1 - Divisão pelo piso e módulo

O operador de divisão pelo piso, //, divide dois números e arredonda o resultado para um número inteiro para baixo. Por exemplo, suponha que o tempo de execução de um filme seja de 105 minutos. Você pode querer saber a quanto isso corresponde em horas. A divisão convencional devolve um número de ponto flutuante:

```
>>> minutes = 105
>>> minutes / 60
1.75
```

Mas não é comum escrever horas com pontos decimais. A divisão pelo piso devolve o número inteiro de horas, ignorando a parte fracionária:

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

Para obter o resto, você pode subtrair uma hora em minutos:

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

Uma alternativa é usar o operador módulo, %, que divide dois números e devolve o resto:

```
>>> remainder = minutes % 60
>>> remainder
45
```

O operador módulo é mais útil do que parece. Por exemplo, é possível verificar se um número é divisível por outro – se x % y for zero, então x é divisível por y.

Além disso, você pode extrair o dígito ou dígitos mais à direita de um número. Por exemplo, x % 10 produz o dígito mais à direita de x (na base 10). Da mesma forma x % 100 produz os dois últimos dígitos.

Se estiver usando o Python 2, a divisão funciona de forma diferente. O operador de divisão, /, executa a divisão pelo piso se ambos os operandos forem números inteiros e faz a divisão de ponto flutuante se pelo menos um dos operandos for do tipo float.

5.2 - Expressões booleanas

Uma expressão booleana é uma expressão que pode ser verdadeira ou falsa. Os exemplos seguintes usam o operador ==, que compara dois operandos e produz True se forem iguais e False se não forem:

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

True e False são valores especiais que pertencem ao tipo bool; não são strings:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

O operador == é um dos operadores relacionais; os outros são:

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

Embora essas operações provavelmente sejam familiares para você, os símbolos do Python são diferentes dos símbolos matemáticos. Um erro comum é usar apenas um sinal de igual (=) em vez de um sinal duplo (==). Lembre-se de que = é um operador de atribuição e == é um operador relacional. Não existe =< ou =>.

5.3 - Operadores lógicos

Há três operadores lógicos: and, or e not. A semântica (significado) destes operadores é semelhante ao seu significado em inglês. Por exemplo, x> 0 and x <10 só é verdade se x for maior que 0 e menor que 10.

n%2 == 0 or n%3 == 0 é verdadeiro se uma ou as duas condição(ões) for(em) verdadeira(s), isto é, se o número for divisível por 2 ou 3.

Finalmente, o operador not nega uma expressão booleana, então not (x > y) é verdade se x > y for falso, isto é, se x for menor que ou igual a y.

Falando estritamente, os operandos dos operadores lógicos devem ser expressões booleanas, mas o Python não é muito estrito. Qualquer número que não seja zero é interpretado como True:

```
>>> 42 and True
True
```

Esta flexibilidade tem sua utilidade, mas há algumas sutilezas relativas a ela que podem ser confusas. Assim, pode ser uma boa ideia evitá-la (a menos que saiba o que está fazendo).

5.4 - Execução condicional

Para escrever programas úteis, quase sempre precisamos da capacidade de verificar condições e mudar o comportamento do programa de acordo com elas. Instruções condicionais nos dão esta capacidade. A forma mais simples é a instrução if:

```
if x > 0:
    print('x is positive')
```

A expressão booleana depois do if é chamada de condição. Se for verdadeira, a instrução endentada é executada. Se não, nada acontece.

Instruções if têm a mesma estrutura que definições de função: um cabeçalho seguido de um corpo endentado. Instruções como essa são chamadas de instruções compostas.

Não há limite para o número de instruções que podem aparecer no corpo, mas deve haver pelo menos uma. Ocasionalmente, é útil ter um corpo sem instruções (normalmente como um espaço reservado para código que ainda não foi escrito). Neste caso, você pode usar a instrução pass, que não faz nada.

```
if x < 0:
    pass          # A FAZER: lidar com valores negativos!
```

5.5 - Execução alternativa

Uma segunda forma da instrução if é a “execução alternativa”, na qual há duas possibilidades e a condição determina qual será executada. A sintaxe pode ser algo assim:

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

Se o resto quando x for dividido por 2 for 0, então sabemos que x é par e o programa exibe uma mensagem adequada. Se a condição for falsa, o segundo conjunto de instruções é executado. Como a condição deve ser verdadeira ou falsa, exatamente uma das alternativas será executada. As alternativas são chamadas de ramos (branches), porque são ramos no fluxo da execução.

5.6 - Condicionais encadeadas

Às vezes, há mais de duas possibilidades e precisamos de mais que dois ramos. Esta forma de expressar uma operação de computação é uma condicional encadeada:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

elif é uma abreviatura de “else if”. Novamente, exatamente um ramo será executado. Não há nenhum limite para o número de instruções elif. Se houver uma cláusula else, ela deve estar no fim, mas não é preciso haver uma.

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

Cada condição é verificada em ordem. Se a primeira for falsa, a próxima é verificada, e assim por diante. Se uma delas for verdadeira, o ramo correspondente é executado e a instrução é encerrada. Mesmo se mais de uma condição for verdade, só o primeiro ramo verdadeiro é executado.

5.7 - Condicionais aninhadas

Uma condicional também pode ser aninhada dentro de outra. Poderíamos ter escrito o exemplo na seção anterior desta forma:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

A condicional exterior contém dois ramos. O primeiro ramo contém uma instrução simples. O segundo ramo contém outra instrução if, que tem outros dois ramos próprios. Esses dois ramos são instruções simples, embora pudessem ser instruções condicionais também.

Embora a endentação das instruções evidencie a estrutura das condicionais, condicionais aninhadas são difíceis de ler rapidamente. É uma boa ideia evitá-las quando for possível.

Operadores lógicos muitas vezes oferecem uma forma de simplificar instruções condicionais aninhadas. Por exemplo, podemos reescrever o seguinte código usando uma única condicional:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

A instrução print só é executada se a colocarmos depois de ambas as condicionais, então podemos obter o mesmo efeito com o operador and:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

Para este tipo de condição, o Python oferece uma opção mais concisa:

```
if 0 < x < 10:
    print('x is a positive single-digit number.')
```

5.8 - Recursividade

É legal para uma função chamar outra; também é legal para uma função chamar a si própria. Pode não ser óbvio porque isso é uma coisa boa, mas na verdade é uma das coisas mais mágicas que um programa pode fazer. Por exemplo, veja a seguinte função:

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

Se n for 0 ou negativo, a palavra “Blastoff!” é exibida, senão a saída é n e então a função countdown é chamada – por si mesma – passando n-1 como argumento.

O que acontece se chamarmos esta função assim?

```
>>> countdown(3)
```

A execução de countdown inicia com n=3 e como n é maior que 0, ela produz o valor 3 e então chama a si mesma...

A execução de countdown inicia com n=2 e como n é maior que 0, ela produz o valor 2 e então chama a si mesma...

A execução de countdown inicia com n=1 e como n é maior que 0, ela produz o valor 1 e então chama a si mesma...

A execução de countdown inicia com n=0 e como n não é maior que 0, ela produz a palavra “Blastoff!” e então retorna.

O countdown que recebeu n=1 retorna.

O countdown que recebeu n=2 retorna.

O countdown que recebeu n=3 retorna.

E então você está de volta ao `__main__`. Então a saída completa será assim:

```
3
2
1
Blastoff!
```

Uma função que chama a si mesma é dita recursiva; o processo para executá-la é a recursividade.

Como em outro exemplo, podemos escrever uma função que exiba uma string n vezes:

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

Se `n <= 0` a instrução `return` causa a saída da função. O fluxo de execução volta imediatamente a quem fez a chamada, e as linhas restantes da função não são executadas.

O resto da função é similar à countdown: ela mostra s e então chama a si mesma para mostrar s mais n-1 vezes. Então o número de linhas da saída é 1 + (n - 1), até chegar a n.

Para exemplos simples como esse, provavelmente é mais fácil usar um loop for. Mais adiante veremos exemplos que são difíceis de escrever com um loop for e fáceis de escrever com recursividade, então é bom começar cedo.

5.9 - Diagramas da pilha para funções recursivas

Em “Diagrama da pilha”, na página 55, usamos um diagrama da 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.

Cada vez que uma função é chamada, o Python cria um frame para conter as variáveis locais e parâmetros da função. Para uma função recursiva, pode haver mais de um frame na pilha ao mesmo tempo.

A Figura 5.1 mostra um diagrama da pilha para `countdown` chamado com n = 3.

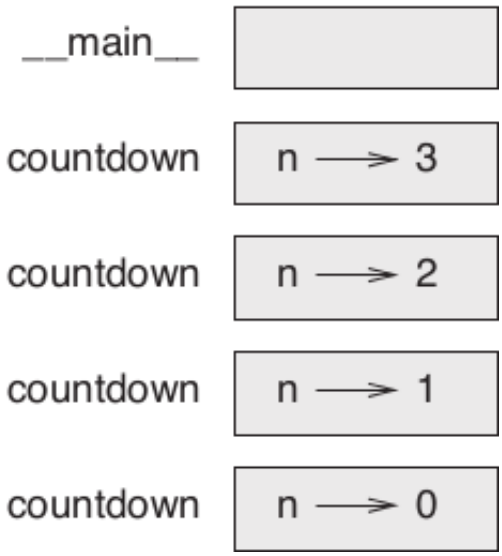

Figura 5.1 – Diagrama da pilha.

Como de hábito, o topo da pilha é o frame de `__main__` . Está vazio porque não criamos nenhuma variável em `__main__` nem passamos argumentos a ela.

Os quatro frames do countdown têm valores diferentes para o parâmetro `n` . O fundo da pilha, onde `n = 0` , é chamado caso-base. Ele não faz uma chamada recursiva, então não há mais frames.

Como exercício, desenhe um diagrama da pilha para `print_n` chamado com `s = 'Hello'` e `n = 2` . Então escreva uma função chamada `do_n` que tome um objeto de função e um número `n` como argumentos e que chame a respectiva função `n` vezes.

5.10 - Recursividade infinita

Se a recursividade nunca atingir um caso-base, continua fazendo chamadas recursivas para sempre, e o programa nunca termina. Isso é conhecido como recursividade infinita e geralmente não é uma boa ideia. Aqui está um programa mínimo com recursividade infinita:

```
def recurse():
    recurse()
```

Na maior parte dos ambientes de programação, um programa com recursividade infinita não é realmente executado para sempre. O Python exibe uma mensagem de erro quando a profundidade máxima de recursividade é atingida:

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                  .
                  .
                  .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

Este traceback é um pouco maior que o que vimos no capítulo anterior. Quando o erro ocorre, há mil frames de recurse na pilha!

Se você escrever em recursividade infinita por engano, confira se a sua função tem um caso-base que não faz uma chamada recursiva. E se houver um caso-base, verifique se você vai mesmo atingi-lo.

5.11 - Entrada de teclado

Os programas que escrevemos até agora não aceitam entradas do usuário. Eles sempre fazem a mesma coisa cada vez.

O Python fornece uma função integrada chamada `input` que interrompe o programa e espera que o usuário digite algo. Quando o usuário pressionar Return ou Enter, o programa volta a ser executado e input retorna o que o usuário digitou como uma string. No Python 2, a mesma função é chamada `raw_input` .

```
>>> text = input()
What are you waiting for?
>>> text
What are you waiting for?
```

Antes de receber entradas do usuário, é uma boa ideia exibir um prompt dizendo ao usuário o que ele deve digitar. input pode ter um prompt como argumento:

```
>>> name = input('What...is your name?\\n')
What...is your name?
Arthur, King of the Britons!
>>> name
Arthur, King of the Britons!
```

A sequência `\n` no final do prompt representa um newline, que é um caractere especial de quebra de linha. É por isso que a entrada do usuário aparece abaixo do prompt.

Se esperar que o usuário digite um número inteiro, você pode tentar converter o valor de retorno para int:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swal
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
42
```

Mas se o usuário digitar algo além de uma série de dígitos, você recebe um erro:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

Veremos como tratar este tipo de erro mais adiante.

5.12 - Depuração

Quando um erro de sintaxe ou de tempo de execução ocorre, a mensagem de erro contém muita informação, às vezes, até demais. As partes mais úteis são normalmente:

- que tipo de erro foi;
- onde ocorreu.

Erros de sintaxe são normalmente fáceis de encontrar, mas há algumas pegadinhas. Erros de whitespace podem ser complicados porque os espaços e tabulações são invisíveis e estamos acostumados a ignorá-los.

```
>>> x = 5
>>> y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

Neste exemplo, o problema é que a segunda linha está endentada por um espaço. Mas a mensagem de erro aponta para y, o que pode ser capcioso. Em geral, mensagens de erro indicam onde o problema foi descoberto, mas o erro real pode estar em outra parte do código, às vezes, em uma linha anterior.

O mesmo acontece com erros em tempo de execução. Suponha que você esteja tentando calcular a proporção de sinal a ruído em decibéis. A fórmula é SNRdb = 10 log10 (Psignal/Pnoise). No Python, você poderia escrever algo assim:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

Ao executar este programa, você recebe uma exceção:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

A mensagem de erro indica a linha 5, mas não há nada de errado com esta linha. Uma opção para encontrar o verdadeiro erro é exibir o valor de ratio, que acaba sendo 0. O problema está na linha 4, que usa a divisão pelo piso em vez da divisão de ponto flutuante.

É preciso ler as mensagens de erro com atenção, mas não assumir que tudo que dizem esteja correto.

5.13 - Glossário

divisão pelo piso
: Um operador, denotado por //, que divide dois números e arredonda o resultado para baixo (em direção ao zero), a um número inteiro.

operador módulo
: Um operador, denotado com um sinal de percentagem (%), que funciona com números inteiros e devolve o resto quando um número é dividido por outro.

expressão booleana
: Uma expressão cujo valor é True (verdadeiro) ou False (falso).

operador relacional
: Um destes operadores, que compara seus operandos: `==`, `!=`, `>`, `<`, `>=` e `<=`.

operador lógico
: Um destes operadores, que combina expressões booleanas: and (e), or (ou) e not (não).

instrução condicional
: Uma instrução que controla o fluxo de execução, dependendo de alguma condição.

condição
: A expressão booleana em uma instrução condicional que determina qual ramo

deve ser executado.

instrução composta
Uma instrução composta de um cabeçalho e um corpo. O cabeçalho termina em dois pontos (:). O corpo é endentado em relação ao cabeçalho.

ramo
Uma das sequências alternativas de instruções em uma instrução condicional.

condicional encadeada
Uma instrução condicional com uma série de ramos alternativos.

condicional aninhada
Uma instrução condicional que aparece em um dos ramos de outra instrução condicional.

instrução de retorno
Uma instrução que faz uma função terminar imediatamente e voltar a quem a chamou.

recursividade
O processo de chamar a função que está sendo executada no momento.

caso-base
Um ramo condicional em uma função recursiva que não faz uma chamada recursiva.

recursividade infinita
Recursividade que não tem um caso-base, ou nunca o atinge. A recursividade infinita eventualmente causa um erro em tempo de execução.

5.14 - Exercícios

Exercício 5.1

O módulo time fornece uma função, também chamada time, que devolve a Hora Média de Greenwich na "época", que é um momento arbitrário usado como ponto de referência. Em sistemas UNIX, a época é primeiro de janeiro de 1970.

```
>>> time.time()
1437746094.5735958
```

Escreva um script que leia a hora atual e a converta em um tempo em horas, minutos e segundos, mais o número de dias desde a época.

Exercício 5.2

O último teorema de Fermat diz que não existem números inteiros a, b e c tais que `a**n + b**n == c**n` para quaisquer valores de n maiores que 2.

1. Escreva uma função chamada check_fermat que receba quatro parâmetros – a, b, c e n – e verifique se o teorema de Fermat se mantém. Se n for maior que 2 e `a**n + b**n == c**n` o programa deve imprimir, "Holy smokes, Fermat was wrong!" Senão o programa deve exibir "No, that doesn't work."

2. Escreva uma função que peça ao usuário para digitar valores para a, b, c e n, os converta em números inteiros e use check_fermat para verificar se violam o teorema de Fermat.

Exercício 5.3

Se você tiver três gravetos, pode ser que consiga arranjá-los em um triângulo ou não. Por exemplo, se um dos gravetos tiver 12 polegadas de comprimento e outros dois tiverem uma polegada de comprimento, não será possível fazer com que os gravetos curtos se encontrem no meio. Há um teste simples para ver se é possível formar um triângulo para quaisquer três comprimentos:

Se algum dos três comprimentos for maior que a soma dos outros dois, então você não pode formar um triângulo. Senão, você pode. (Se a soma de dois comprimentos igualar o terceiro, eles formam um triângulo chamado "degenerado".)

1. Escreva uma função chamada `is_triangle` que receba três números inteiros como argumentos, e que imprima "Yes" ou "No", dependendo da possibilidade de formar ou não um triângulo de gravetos com os comprimentos dados.

2. Escreva uma função que peça ao usuário para digitar três comprimentos de gravetos, os converta em números inteiros e use `is_triangle` para verificar se os gravetos com os comprimentos dados podem formar um triângulo.

Exercício 5.4

Qual é a saída do seguinte programa? Desenhe um diagrama da pilha que mostre o estado do programa quando exibir o resultado.

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)

recurse(3, 0)
```

1. O que aconteceria se você chamasse esta função desta forma: recurse(-1, 0)?
2. Escreva uma docstring que explique tudo o que alguém precisaria saber para usar esta função (e mais nada).

Os seguintes exercícios usam o módulo turtle, descrito no Capítulo 4:

Exercício 5.5

Leia a próxima função e veja se consegue compreender o que ela faz (veja os exemplos no Capítulo 4). Então execute-a e veja se acertou.

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length * n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2 * angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length * n)
```

Exercício 5.6

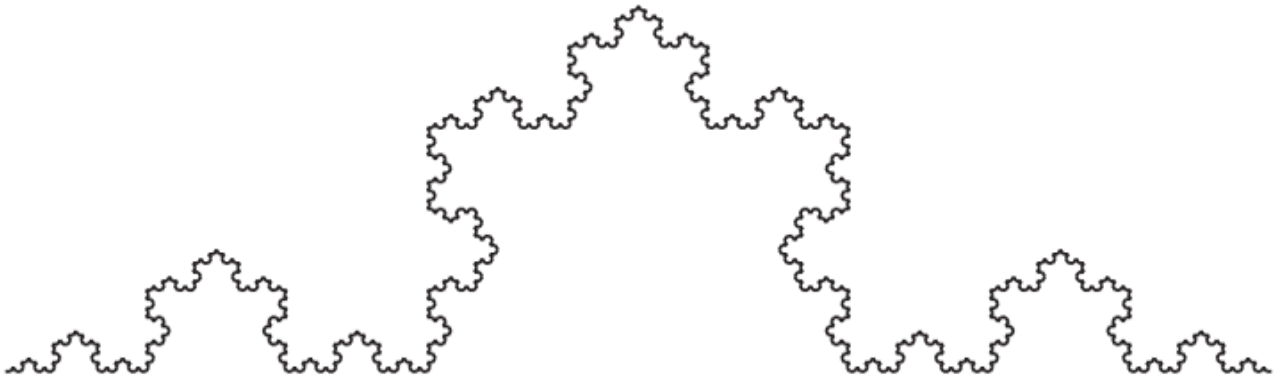

Figura 5.2 – Uma curva de Koch.

A curva de Koch é um fractal que parece com o da Figura 5.2. Para desenhar uma curva de Koch com o comprimento x, tudo o que você tem que fazer é:

1. Desenhe uma curva de Koch com o comprimento x/3.
2. Vire 60 graus à esquerda.
3. Desenhe uma curva de Koch com o comprimento x/3.
4. Vire 120 graus à direita.
5. Desenhe uma curva de Koch com o comprimento x/3.
6. Vire 60 graus à esquerda.
7. Desenhe uma curva de Koch com o comprimento x/3.

A exceção é se x for menor que 3: neste caso, você pode desenhar apenas uma linha reta com o comprimento x.

1. Escreva uma função chamada koch que receba um turtle e um comprimento como parâmetros, e use o turtle para desenhar uma curva de Koch com o comprimento dado.
2. Escreva uma função chamada snowflake que desenhe três curvas de Koch para fazer o traçado de um floco de neve.

   Solução: http://thinkpython2.com/code/koch.py.
3. A curva de Koch pode ser generalizada de vários modos. Veja exemplos em http://en.wikipedia.org/wiki/Koch_snowflake e implemente o seu favorito.