Capítulo 3

Execução condicional

3.1 Expressões booleanas

Uma *expressão booleana* é uma expressão que pode ser ou verdadeira ou falsa. Os exemplos a seguir utilizam o operador `==`, que compara dois operandos e produz um resultado positivo `True`, se forem iguais, e um resultado negativo `False`, caso contrário.

```
>>> 5 == 5
True
>>> 5 == 6
False
{}
```

`True` e `False` são valores especiais que pertecem à classe `bool`; eles não são strings:

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

O operador `==` é um dos *operadores de comparação*; outros operadores desse tipo são:

```
x != y               # x é diferente de y
x > y                # x é maior do que y
x < y                # x é menor do que y
x >= y               # x é maior ou igual a y
x <= y               # x é menor ou igual a y
x is y               # x é o mesmo que y
x is not y           # x não é o mesmo que y
```

Apesar dessas expressões provavelmente serem familiares para você, os símbolos em Python são diferentes dos símbolos utilizados na matemática para realizar as mesmas operações. Um erro comum é usar apenas um sinal (=) ao invés de um sinal duplo (==) para comparar uma igualdade. Lembre-se que = é um operador de atribuição e == é um operador de comparação. Vale mencionar que não há operadores =< e =>.

3.2 Operadores lógicos

Existem três operadores lógicos: `and`, `or`, e `not`. A semântica (significado) desses operadores são similares ao seu significado em Inglês. Por exemplo,

`x > 0 and x < 10`

só é verdadeiro se `x` for maior do que 0 *e* menor que 10.

`n%2 == 0 or n%3 == 0` é verdadeiro se alguma das condições é verdadeira, isto é, se o número é divisível por 2 *ou* 3.

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

Rigorosamente falando, os operandos dos operadores lógicos devem ser expressões booleanas, mas o Python não é muito rigoroso. Qualquer número diferente de zero é interpretado como "verdadeiro".

```
>>> 17 and True
True
```

Essa flexibilidade pode ser útil, mas existem algumas sutilezas que podem ser confusas. É bom evitá-los até você ter certeza de que sabe o que está fazendo.

3.3 Execução condicional

A fim de escrever programas úteis, nós sempre necessitamos da habilidade de checar condições e de mudar o comportamento do programa de acordo com elas. *Declarações Condicionais* nos dão essa habilidade. A forma mais simples é a declaração `if`:

```
if x > 0 :
    print('x é positivo')
```

A expressão booleana depois da declaração `if` é chamada de *condição*. Nós terminamos essa declaração com símbolo de dois pontos (:) e a(s) linha(s) depois da declaração `if` são indentadas.

Se a condição lógica é verdadeira, então a declaração indentada é executada. Se a condição lógica for falsa, a condição indentada é ignorada.

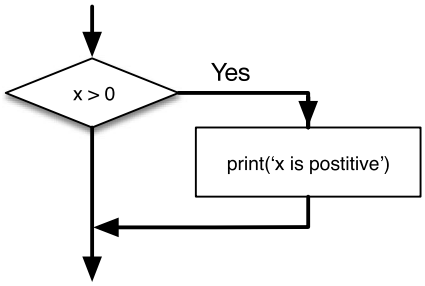

Figure 3.1: If Logic

Declarações `if` têm a mesma estrutura que as definições de funções ou laços `for` [1]. A declaração consiste com uma linha de cabeçalho que acaba com um símbolo de dois pontos (:) seguido de um bloco indentado. Declarações como essas são chamadas de *declarações compostas*, pois elas se alongam por mais de uma linha.

Não existe limite para o número de declarações que se pode aparecer no corpo, mas é necessária que apareça pelo menos uma. Ocasionalmente, é útil deixar um corpo sem declarações (geralmente para guardar o lugar de um código que você ainda não escreveu). Nesse caso, você pode utilizar a declaração `pass`, que não faz nada.

```
if x < 0 :
    pass          # necessitamos tratar os valores negativos!
```

Se você digitar uma declaração `if` no interpretador Python, o prompt mudará de três chevrons para três pontos de modo a indicar que você está no meio de um bloco de declarações, como é mostrado abaixo:

```
>>> x = 3
>>> if x < 10:
...     print('Pequeno')
...
Pequeno
>>>
```

Ao utilizar o interpretador Python, você deverá deixar uma linha em branco no final do bloco, ao contrário o Python retornará um erro:

```
>>> x = 3
>>> if x < 10:
...     print('Pequeno')
... print('Feito')
  File "<stdin>", line 3
    print('Feito')
        ^
SyntaxError: invalid syntax
```

[1] Nós aprenderemos sobre funções no Capítulo 4 e laços no Capítulo 5.

Uma linha em branco no final do bloco de declaração não é necessária quando se está escrevendo e executando o script, mas pode melhorar a leitura do seu código.

3.4 Execução alternativa

Uma segunda forma da declaração `if` é a *execução alternativa*, na qual existem duas possibilidades e a condição determina qual delas deve ser executada. A sintaxe se dá da seguinte forma:

```
if x%2 == 0 :
    print('x é par')
else :
    print('x é ímpar')
```

Se o resto da divisão de `x` por 2 for 0, então sabemos que `x` é par, e o programa irá exibir uma mensagem para tal efeito. Entretanto, se a condição for falsa, o segundo conjunto de intruções será executado.

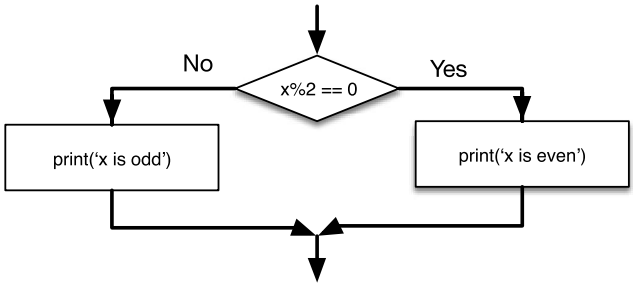

Figure 3.2: If-Then-Else Logic

Como a condição deve ser verdadeira ou falsa, exatamente uma das alternativas será executada. As alternativas são chamadas de *ramificações*, porque são ramificações do fluxo de execução.

3.5 Condições encadeadas

As vezes há mais de duas possibilidades e precisamos de mais de duas ramificações. Uma maneira de expressar uma lógica computacional como essa é por meio de uma *condição encadeada*:

```
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')
```

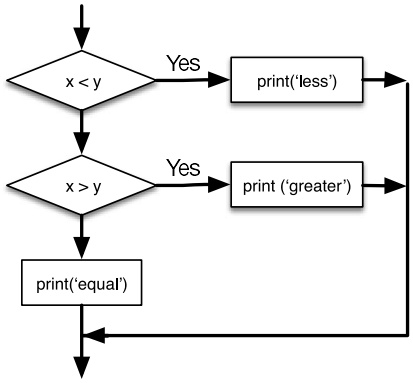

Figure 3.3: If-Then-ElseIf Logic

`elif` é uma abreviação de "else if". Novamente, apenas um caminho será executado.

Não há limite para o número de declarações `elif`. Se houver uma cláusula `else`, ela tem que estar no fim, mas não há a necessidade de haver uma.

```
if choice == 'a':
    print('Mau Palpite')
elif choice == 'b':
    print('Bom Palpite')
elif choice == 'c':
    print('Perto, mas não está correto')
```

Cada condição é verificada em ordem. Se a primeira for falsa, a próxima é verificada, e assim por diante. Se ao menos uma delas for verdadeira, a ramificação correspondente será executada e o bloco condicional terminará. Mesmo que mais de uma condição seja verdadeira, somente a primeira ramificação verdadeira é executada.

3.6 Condições aninhadas

Uma condição pode também ser aninhada com outra. Nós poderiamos ter escrito o exemplo com três ramos assim:

```
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')
```

A condição externa contém dois ramos. O primeiro contém uma instrução simples. O segundo ramo contém outra declaração `if`, que possui mais dois ramos. Esses

dois são ambos declarações simples, embora eles também pudessem ser declarações condicionais.

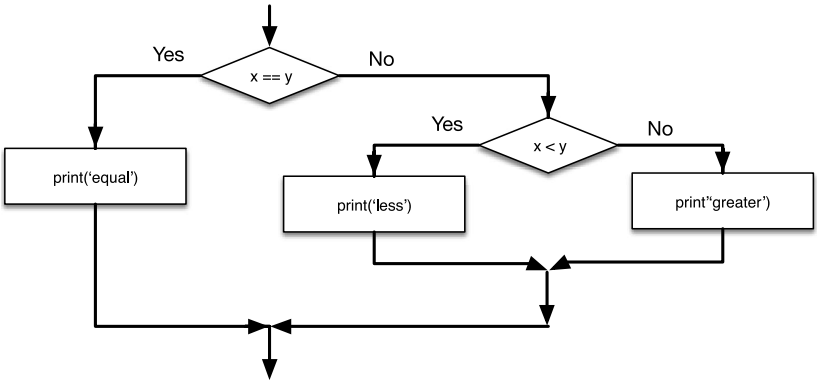

Figure 3.4: Sentenças de if aninhadas

Embora a indentação das sentenças torne a estrutura aparente, *condições aninhadas* são difíceis de serem lidas rapidamente. Em geral, quando possível, é uma boa ideia evitar elas.

Operadores lógicos frequentemente indicam um meio de simplificar condições aninhadas. Por exemplo, podemos reescrever o seguinte código utilizando uma declaração condicional simples:

```
if 0 < x:
    if x < 10:
        print('x é um número positivo de 1 dígito.')
```

A instrução `print` é executada apenas se as duas condicionais forem ultrapassadas, logo, podemos obter o mesmo efeito com o operador `and`:

```
if 0 < x and x < 10:
    print('x é um número positivo de 1 dígito.')
```

3.7 Tratando exceções usando try e except

Anteriormente, vimos um trecho de código onde usamos as funções 'input' e 'int' para ler e analisar um número inteiro inserido pelo usuário. Vimos também o quão traiçoeiro fazer isso poderia ser:

```
>>> prompt = "Qual é a velocidade rasante de uma andorinha sem carga?\n"
>>> velocidade = input(prompt)
Qual é a velocidade rasante de uma andorinha?
Seja mais específico, uma andorinha africana ou europeia?
>>> int(velocidade)
ValueError: invalid literal for int() with base 10:
        (conversão literal inválida para int() com base 10)
>>>
```

Quando executamos esse trecho de código no interpretador, nós recebemos um novo aviso do Python, pensamos "oops" e passamos para nossa próxima declaração.

Entretanto, se você colocar esse código num programa em Python e esse erro ocorre, seu programa para imediatamente com um "Traceback" indicando tal erro. Isso faz com que os próximos comandos não sejam executados.

Aqui está um simples programa que converte uma temperatura em Fahrenheit para Celsius:

```
inp = input('Enter Fahrenheit Temperature: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

# Code: http://www.py4e.com/code3/fahren.py
```

Se executarmos esse código e inserirmos uma entrada inválida, ele simplesmente falha, mostrando uma mensagem de erro não amigável:

```
python fahren.py
Insira a temperatura em Fahrenheit:72
22.22222222222222

python fahren.py
Insira a temperatura em Fahrenheit:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(ent)
ValueError: could not convert string to float: 'fred'
           (não foi possível converter string para float: 'fred')
```

Existe uma estrutura de execução condicional interna ao Python para manipular esses tipos de erros esperados e inesperados chamada "try / except". A ideia de 'try' e 'except' é de você saber que alguma sequência de instruções pode ter um problema e você quer adicionar alguns comandos para serem executados caso um erro ocorra. Esses comandos extra (o bloco 'except') são ignorados se não houver erro.

Você pode pensar na ferramenta 'try' e 'except' no Python como sendo uma "apólice de seguro" sobre uma sequência de instruções.

Podemos reescrever nosso conversor de temperatura da seguinte forma:

```
inp = input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Please enter a number')

# Code: http://www.py4e.com/code3/fahren2.py
```

O Python começa executando a sequência de instruções que estão no bloco 'try'. Se tudo ocorrer bem, pula-se o bloco 'except' e o programa continua. Se uma exceção ocorrer no bloco 'try', o Python sai desse bloco e executa os comandos no bloco 'except'.

```
python fahren2.py
Insira a temperatura em Fahrenheit:72
22.22222222222222

python fahren2.py
Insira a temperatura em Fahrenheit:fred
Por favor, insira um número
```

O ato de tratar uma exceção com um comando 'try' é chamado de *capturar* uma exceção. Nesse exemplo, a condição 'except' mostra uma mensagem de erro. Geralmente, capturar uma exceção te dá a chance de resolver um problema, ou de tentar novamente, ou de, pelo menos, encerrar seu programa graciosamente.

3.8 Avaliação de curto-circuito de expressões lógicas

Quando Python está processando uma expressão lógica como `x >= 2 and (x/y) > 2`, a expressão é analisada da esquerda para a direita. Devido à definição de `and`, se `x` é menor que 2, a expressão `x >= 2` é `False` e então toda a expressão também é `False` independentemente se `(x/y) > 2` é considerada como `True` ou `False`.

Quando o Python detecta que não há nada a ser ganho examinando o resto da expressão lógica, a análise é interrompida e não são feitos os cálculos restantes. Quando a interpretação de uma expressão lógica é interrompida porque o valor geral já é conhecido, isso é chamado de *curto-circuito* na análise.

Embora isso possa parecer um detalhe, o comportamento de um curto-circuito leva a uma técnica chamada de *padrão guardião* (*guardian pattern*). Considere o seguinte trecho de código no compilador de Python:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
                   (divisão por zero)
>>>
```

O terceiro cálculo falhou, pois Python estava analisando `(x/y)` e `y` era zero, causando um erro ao executar. Mas o segundo exemplo *não* falhou, pois a primeira parte da expressão `x >= 2` foi considerada `False` então `(x/y)` nunca foi executado devido à regra do *curto-circuito* e não houve erro.

Podemos construir a expressão lógica estrategicamente para posicionar uma avaliação de *guarda* logo antes da avaliação que pode causar o seguinte erro:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
                   (divisão por zero)
>>>
```

Na primeira expressão lógica, `x >= 2` é `False` então a análise para no `and`. Na segunda expressão lógica, `x >= 2` é `True` mas `y != 0` é `False` então nunca chegamos em `(x/y)`.

Na terceira expressão lógica, o `y != 0` é *depois* do cálculo de `(x/y)` então a expressão falha com um erro.

Na segunda expressão, dizemos que `y != 0` age como uma *guarda* para garantir que executaremos apenas `(x/y)` se `y` é diferente de zero.

3.9 Debugging

O "traceback" que o Python exibe quando um erro ocorre contém bastante informação, mas isso pode ser um pouco sufocante. As partes mais úteis são usualmente:

- Que tipo de erro aconteceu, e
- Onde ele ocorreu.

Erros de sintaxe são geralmente fáceis de achar, mas existem algumas "armadilhas". Erros envolvendo espaços em branco podem ser traiçoeiros devido a espaços e "tabs" serem invisíveis e estarmos acostumados a ignorá-los.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
```

```
    ^
IndentationError: unexpected indent
                  (indentação inprevista)
```

Nesse exemplo, o problema está no fato de que a segunda linha está indentada com um espaço. Entretanto, a mensagem de erro aponta para o 'y', o que é enganoso. Comumente, mensagens de erro indicam onde o problema foi descoberto, mas o erro em si pode estar um pouco antes de onde foi apontado, às vezes na linha anterior do código.

De modo geral, mensagens de erro te mostram onde o problema foi encontrado, apesar de muitas vezes este local não ser onde ele foi causado.

3.10 Glossário

Corpo Sequência de instruções dentro de uma instrução composta.
Condicional aninhada Uma condicional que aparece em um dos ramos de outra sentença condicional.
Condicional encadeada Uma condicional com uma série de ramos alternativos.

Condição A expressão booleana em uma sentença condicional que determina qual ramo será executado.
Curto-circuito Quando Python está no meio da verificação de uma expressão lógica e para a verificação porque sabe o resultado final da expressão, sem a necessidade de verificar o restante desta.
Expressão booleana Uma expressão cujo valor é `True` ou `False`.
Operador de comparação Um operador que compara seus operandos: `==`, `!=`, `>`, `<`, `>=`, e `<=`.
Operador lógico Operadores que combinam expressões booleanas: `and`, `or`, e `not`.
Padrão Guardião (*guardian pattern*) Quando construímos uma expressão lógica com comparações adicionais para ganhar vantagem com o comportamento de short-circuit.
Ramo Uma das alternativas de sequência de instruções dentro de uma condicional.

Sentença composta Uma sentença que consiste em um cabeçalho e um corpo. O cabeçalho termina com dois pontos (:). O corpo é indentado relativo ao cabeçalho.
Sentença condicional Uma sentença que controla o fluxo de execução dependendo de alguma condição.
Traceback Uma lista das funções que estão sendo executadas, mostrada quando uma exceção ocorre.

3.11 Exercícios

Exercício 1: Reescreva seu programa de pagamento, para pagar ao funcionário 1.5 vezes o valor da taxa horária de pagamento pelo tempo trabalhado acima de 40 horas

```
Digite as Horas: 45
Digite a taxa: 10
Pagamento: 475.0
```

Exercício 2: Reescreva seu programa de pagamento utilizando `try` e `except`, de forma que o programa consiga lidar com entradas não numéricas graciosamente, mostrando uma mensagem e saindo do programa. A seguir é mostrado duas execuções do programa

```
Digite as Horas: 20
Digite a taxa: nove
Erro, por favor utilize uma entrada numérica

Digite as Horas: quarenta
Erro, por favor utilize uma entrada numérica
```

Exercício 3: Escreva um programa que peça por uma pontuação entre 0.0 e 1.0. Se a pontuação for fora do intervalo, mostre uma mensagem de erro. Se a pontuação estiver entre 0.0 e 1.0, mostre a respectiva nota usando a seguinte tabela

```
Pontuação    Nota
>= 0.9        A
>= 0.8        B
>= 0.7        C
>= 0.6        D
 < 0.6        F

Digite a Pontuação: 0.95
A

Digite a Pontuação: perfeito
Pontuação Inválida

Digite a Pontuação: 10.0
Pontuação Inválida

Digite a Pontuação: 0.75
C

Digite a Pontuação: 0.5
F
```

Rode o programa repetidamente como mostrado acima para testar diferentes valores de entrada.