Lógica e Escolhas

MC102-2018s1-Aula04-180308

Arthur J. Catto, PhD

08 de março de 2018

1 Lógica e Escolhas

Vimos como resolver um problema com uma sequência de ações:

```
hora_atual = int(input("Que horas são? (0-23) "))
horas_espera = int(input("Quantas horas você vai esperar? (>=0) "))
hora_alarme = (hora_atual + horas_espera) % 24
print("O alarme vai tocar às", hora_alarme, "horas.")
```

Mas nem sempre se consegue resolver um problema dessa maneira...

1.1 Exemplo: *Ler dois valores inteiros e exibir o maior deles*

Suponha que os valores lidos sejam associados a duas variáveis a e b, por exemplo.

Você consegue encontrar uma heurística que, sozinha, resolva todos os casos possíveis?

Você consegue imaginar uma heurística que resolva pelo menos alguns casos?

- Sim, o comando `print(a)` resolve o problema desde que `a > b`.

Para garantir que `print(a)` seja executado somente se `a > b`, usamos um *comando condicional*.

1.1.1 Comando condicional

Um comando condicional é um comando capaz de executar um grupo de um ou mais comandos, desde que uma dada condição seja satisfeita.

```
if condição:
    bloco
```

Um comando condicional permite resolver um problema quando não se encontra uma heurística que, sozinha, resolva todos os casos possíveis.

Quando isso acontecer...

- Procure uma heurística que resolva pelo menos alguns casos e faça com que ela seja executada apenas quando ocorrer um deles.

- Procure outra heurística que resolva pelo menos alguns dos casos que não foram cobertos e faça com que ela seja executada apenas quando ocorrer um deles.
- Repita esse processo até ter coberto todos os casos possíveis.

No exemplo dado...

```
In [ ]: a = int(input("Primeiro valor? "))
        b = int(input("Segundo valor? "))
        if a > b:
            print("O maior valor é", a)
        if a < b:
            print("O maior valor é", b)
        if a == b:
            print("Os dois valores são iguais a", a)
```

Todos os casos estão cobertos?
Estamos satisfeitos com o resultado?

O comando condicional tem uma outra forma que permite tratar melhor situações como essa, em que existe um encdeamanto entre as condições. A palavra chave `elif` é uma contração de `else if`.

```
if condição_1:
    bloco_1
elif condição_2:
    bloco_2
#...
else:
    bloco_n
```

Neste modelo, as condições são avaliadas uma a uma, até que uma seja verdadeira, o que provoca a execução do `bloco` correspondente. Se nenhuma condição for verdadeira, o `bloco_n` da cláusula `else` é executado. Nesse caso, se não houver uma cláusula `else`, o comando `if` não tem qualquer efeito.

Nós podemos usar a cláusula `else` para tratar um último caso ou, por exemplo, para apanhar uma exceção.

1.1.2 Uso do `else` para tratar um último caso

```
In [ ]: a = int(input("Primeiro valor? "))
        b = int(input("Segundo valor? "))
        if a > b:
            print("O maior valor é", a)
        elif a < b:
            print("O maior valor é", b)
        else:
            print("Os dois valores são iguais a", a)
```

1.2 Uso do `else` para apanhar uma exceção

```
In [ ]: a = int(input("Primeiro valor? "))
        b = int(input("Segundo valor? "))
        if a > b:
            print("O maior valor é", a)
        elif a < b:
            print("O maior valor é", b)
        elif a == b:
            print("Os dois valores são iguais a", a)
        else:
            print("Tem alguma coisa errada com este computador!")
```

1.3 O tipo `bool`

Num comando condicional, a *condição* é uma *asserção* (ou *expressão lógica*), isto é, uma afirmação que pode ser *verdadeira* ou *falsa*.

Como a maioria das linguagens de alto-nível, Python também tem um tipo de dados para representar valores lógicos: `bool`.

O tipo `bool` possui apenas dois valores (`True` e `False`) e três operadores (`not`, `and` e `or`).

`not` é um operador unário, enquanto `and` e `or` são binários. Essa é também a ordem de precedência entre eles.

`not` nega o valor de seu operando. Assim, `not True` é `False` e `not False` é `True`.

`and` produz `True` se *ambos* operandos forem `True` ou `False`, caso contrário.

`or` produz `False` se *ambos* operandos forem `False` ou `True`, caso contrário.

```
In [1]: not 3 > 5
```

```
Out[1]: True
```

1.4 Operadores relacionais

Uma condição é frequentemente expressa como uma comparação que usa um *operador relacional* e produz um resultado do tipo `bool`.

Há seis operadores relacionais em Python: >, <, >=, <=, == (igual) e != (diferente).

É possível comparar `int` com `int`, `float` com `float`, `int` com `float` e `str` com `str`.

- No caso de `str`, a comparação é lexicográfica e usa a tabela ASCII.
- Nessa tabela, as letras estão na ordem do alfabeto inglês e maiúsculas são "menores"do que as minúsculas.

```
In [2]: 'abc' > "ABC"
```

```
Out[2]: True
```

1.5 Expressões contendo operadores lógicos e relacionais

Os operadores aritméticos têm precedência sobre os operadores relacionais.

Os operadores relacionais têm precedência sobre os operadores lógicos.

`not` tem precedência sobre `and`, que tem precedência sobre `or`.

```
In [3]: 3 + 5 > 10 and 5 / 0 > 0
```

```
Out[3]: False
```

1.6 Particularidades de Python

1.6.1 `bool` é mapeado sobre `int`

`True` é mapeado sobre `1` e `False` é mapeado sobre `0`. Com isso...

```
In [4]: 5 + (3 < 5)
```

```
Out[4]: 6
```

1.6.2 `int`, `float` e `str` também podem ser mapeados sobre `bool`

Quando aparecem numa expressão lógica, os zeros dos tipos `int` e `float` são considerados `False` e o mesmo vale para a `str` vazia.

Todos os demais valores de `int`, `float` e `str` são considerados `True`.

```
In [5]: s = ""
        if s:
            print("Ok")
        else:
            print("Escreva alguma coisa.")
```

```
Out[5]: Escreva alguma coisa.
```

1.6.3 A avaliação de expressões lógicas é *preguiçosa* (ou em *curto-circuito*)

Isto quer dizer que uma expressão é avaliada apenas até que se possa ter certeza sobre seu resultado.

- Quando `x` é avaliado como `False`, o resultado de `x and y` é `x` (e `y` não é avaliado). Caso contrário, o resultado é `y`.
- Quando `x` é avaliado como `True`, o resultado de `x or y` é `x` (e `y` não é avaliado). Caso contrário, o resultado é `y`.

```
In [6]: 5 or 4
```

```
Out[6]: 5
```

```
In [7]: 5 and 4
```

```
Out[7]: 4
```

1.6.4 Há uma forma "aumentada"de atribuição

Comandos como `x = x + 1` são tão comuns que Python implementa uma notação abreviada: `x += 1`.

```
In [8]: a = 5
        a /= 8
        a

Out[8]: 0.625
```

1.6.5 Operadores relacionais podem ser encadeados

Expressões como $0 \leq x$ `and` $x < n$ são tão comuns que Python permite encadear as comparações: $0 \leq x < n$.

```
In [9]: 3 <= 5 < 10

Out[9]: True
```

Dica

- Evite efeitos colaterais e comandos *"originais e criativos"* porque os resultados podem ser surpreendentes.

```
In [10]: 3 < 5 == True

Out[10]: False

In [11]: 3 < 5 and 5 == True

Out[11]: False
```

1.6.6 É possível criar uma expressão condicional

Um comando como

```
if condição:
    x = valor_cond_ok
else:
    x = valor_cond_nok
```

é equivalente à seguinte *expressão condicional* (que também é avaliada de forma preguiçosa)

```
x = valor_cond_ok if condição else valor_cond_nok
```

Assim, *fazer x igual ao maior entre a e b* pode ser expresso como `x = a if a > b else b`.

1.7 Exemplo: *Ordenar três valores inteiros*

Problema
Dados três valores inteiros exibi-los em ordem crescente.

Entrada
Três valores inteiros quaisquer que devem ser associados a três variáveis `a`, `b` e `c`.

Saída
Um comando `print(a, b, c)` que exiba os mesmos valores lidos, mas em ordem crescente.

É possível resolver esse problema usando um único comando?

- Não.

É possível resolver esse problema usando uma sequência de comandos?

- Sim, o problema pode ser resolvido por uma sequência de comandos, cada um deles desfazendo uma possível inversão entre os valores lidos.

Como é que se desfaz uma inversão?

- Permutando os valores de duas variáveis que estejam fora de ordem.

Como é que se permutam os valores de duas variáveis `x` e `y`?

- Usando uma variável auxiliar ou uma *atribuição múltipla*.

```
t = x
x = y
y = t
```

ou

```
x, y = y, x
```

Nota: numa atribuição múltipla, primeiro são avaliadas todas as expressões, na ordem em que aparecem no comando, e depois são feitas todas as associações, na mesma ordem.

1.7.1 Solução: Ordenar três valores inteiros

```
In [ ]: x = int(input("Primeiro valor? "))
        y = int(input("Segundo valor? "))
        z = int(input("Terceiro valor? "))
        if x > y:
            x, y = y, x
        if x > z:
            x, z = z, x
        if y > z:
            y, z = z, y
        print(x, y, z)
```

Como você se convence de que essa solução está correta?

1. No início da solução nada se sabe sobre x, y e z.
2. Depois do primeiro `if`, sabemos que $x \leq y$.
3. Depois do segundo `if`, sabemos também que $x \leq z$. Portanto, x é o menor dos três valores.
4. Depois do terceiro `if`, sabemos que $y \leq z$.
5. Portanto, $x \leq y \leq z$ e o problema está resolvido.