# Capítulo 5

# Iteração

## 5.1 Atualizando Variáveis

Um modelo comum de declaração de atribuição é uma que atualiza o valor de uma variável, na qual o seu novo valor depende do seu anterior.

```
x = x + 1
```

Isso significa "pegue o valor atual de x, adicione 1 e então atualize x com esse novo valor."

Se você tentar atualizar uma variável que não existe, ocorrerá um erro, já que o Python avalia o lado direito da igualdade antes de atribuir um valor a `x`:

```
>>> x = x + 1
NameError: name 'x' is not defined
           (nome 'x' não definido)
```

Antes de se poder atualizar uma variável, é preciso *inicializá-la*, normalmente com uma simples atribuição:

```
>>> x = 0
>>> x = x + 1
```

Atualizar uma variável adicionando 1 é chamado de um *incremento*; subtraindo 1 é chamado de um *decremento*.

## 5.2 A declaração `while`

Frequentemente, computadores são utilizados para automatizar tarefas repetitivas. Repetir tarefas, idênticas ou similares, sem cometer erros, é algo que computadores fazem bem melhor que pessoas. Devido à iteração ser tão comum, Python disponibiliza diversos recursos para torná-la mais fácil.

Uma das formas de iteração em Python é a declaração `while`. Abaixo, temos um programa simples que faz uma contagem regressiva, partindo de cinco, e depois diz "Lançar!".

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Lançar!')
```

A declaração `while` quase pode ser lida como se fosse um texto comum (escrito em português). Em português, teríamos algo como: "Enquanto `n` for maior que 0, mostre o valor de `n` e então subtraia 1 desse valor. Quando atingir o valor 0, saia da declaração `while` e mostre a palavra `Lançar!`".

De maneira mais formal, o fluxo de execução da declaração `while` seria:

1. Análise da condição, retornando um valor `True` ou `False`.
2. Se a condição for falsa, saia da declaração `while` e prossiga com as declarações seguintes.
3. Se a condição for verdadeira, execute o bloco `while` e então retorne ao passo 1.

Esse bloco de instruções é chamado de *laço*, pois o terceiro passo faz retornar ao primeiro. Nomeamos de *iteração* cada vez em que as instruções dentro de um laço são executadas. Para o programa anterior, podemos dizer que "ele teve cinco iterações", ou seja, o corpo do laço foi executado cinco vezes.

O corpo do laço normalmente deve alterar o valor de uma ou de mais variáveis, de forma que, eventualmente, a condição de execução se torne falsa e a repetição se encerre. Chamamos essa variável que tem seu valor alterado em cada execução do laço e controla quando ele deve terminar de *variável de iteração*. Se não houver uma variável de iteração, o loop se repetirá eternamente, resultando em um *laço infinito*.

## 5.3 Laços infinitos

Uma fonte de diversão sem fim para programadores é observar que as instruções no shampoo, "Ensaboe, enxague, repita", são um laço infinito pois não há *variável de iteração* lhe dizendo quantas vezes executar essa sequência.

Numa `contagem regressiva`, nós podemos provar que o laço termina, pois sabemos que o valor de `n` é finito, e podemos ver que esse valor fica cada vez menor durante o laço, então eventualmente chegaremos ao 0. Outras vezes, um laço é claramente infinito, pois não existe variável de iteração.

Às vezes você não sabe que é hora de terminar o laço até chegar ao meio dele. Nesse caso você pode escrever um laço infinito de propósito e então usar o comando `break` para interrompê-lo.

Este laço abaixo é obviamente um *laço infinito*, pois a expressão lógica na estrutura `while` é simplesmente a a constante lógica `True`:

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('Pronto!')
```

Se você cometer o erro de executar esse código, você aprenderá rapidamente como parar um processo do Python em execução no seu sistema ou achar onde é o botão de desligar do seu computador. Esse programa rodará eternamente ou até a sua bateria acabar, pois a expressão lógica no topo do laço é sempre verdadeira, graças ao fato de que a expressão é o valor constante `True`.

Embora isso seja um laço infinito disfuncional, nós podemos continuar usando esse padrão para construir laços úteis, desde que, cuidadosamente, adicionemos um código ao corpo do laço que, utilizando `break`, explicitamente saia dele quando atingimos a condição de saída desejada.

Por exemplo, suponha que você queira entradas do usuário até que ele digite `pronto`. Você pode escrever:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: http://www.py4e.com/code3/copytildone1.py
```

A condição do laço é `True`, que sempre é verdadeira, então ele é executado repetidamente até atingir a estrutura break.

A cada repetição, ele solicita uma entrada do usuário mostrando um sinal de "menor que". Se o usuário digita `pronto`, o comando `break` sai do laço. Caso contrário, o programa ecoa tudo o que o usuário digita e volta para o primeiro passo. Aqui vemos um exemplo da execução:

```
> olá
olá
> encerrar
encerrar
> pronto
Pronto!
```

Essa forma de escrever laços `while` é comum, pois você pode checar a condição em qualquer lugar do bloco (não apenas no topo) e você pode expressar a condição de parada afirmativamente ("pare quando isso acontece") melhor do que negativamente ("continue até que aquilo aconteça").

## 5.4 Finalizando iterações com `continue`

Às vezes, você está em uma iteração de um laço e deseja finalizar essa iteração atual e pular imediatamente para a próxima. Nesse caso, você pode usar o comando `continue` para avançar para a próxima iteração sem executar as instruções restantes da iteração atual.

Aqui está um exemplo de laço que copia as entradas do usuário até ele digitar "pronto", mas considera as linhas que começam com o caractere cerquilha (#) como linhas que não devem ser exibidas (como os comentários em Python).

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: http://www.py4e.com/code3/copytildone2.py
```

Aqui temos um exemplo da execução desse novo programa com `continue` adicionado.

```
> olá
olá
> # não exiba isso
> exiba isso!
exiba isso!
> pronto
Pronto!
```

Todas as linhas são exibidas exceto a que começa com o sinal cerquilha, pois quando o `continue` é executado, ele termina a iteração atual e pula de volta para o topo do bloco `while` para começar a próxima iteração, pulando, portanto, o comando `print`.

## 5.5 Definindo um laço usando `for`

Às vezes, queremos fazer um laço por meio de um conjunto de coisas, como uma lista de palavras, as linhas em um arquivo, ou uma lista de números. Quando temos uma lista de coisas para iterar sobre, podemos construir um laço definido usando uma declaração `for`. Nós chamamos a declaração `while` como um laço indefinido, porque ela simplesmente permanece iterando até que alguma condição se torne falsa, enquanto o laço `for` itera sobre um conjunto de itens conhecidos até que sejam executadas tantas iterações quanto o número de itens nesse conjunto.

A sintaxe de um laço `for` é similar a de um laço `while`, em que há uma declaração `for` e um corpo de repetição:

```
amigos = [‘Jose’, ‘Gleice’, ‘Sara’]
For amigo in amigos:
    print('Feliz Ano Novo', amigo)
print('Feito!')
```

Em termos Python, a variável `amigos` é uma lista [examinaremos as listas com mais detalhes em um capítulo posterior.] de três cadeias de caracteres e o laço `for` percorre a lista e executa o corpo uma vez para cada uma das três palavras na lista, resultando nesta saída:

```
Feliz Ano Novo: Jose
Feliz Ano Novo: Gleice
Feliz Ano Novo: Sara
Feito!
```

Traduzir esse loop `for` para o português não é tão direto quanto o `while`, mas se você pensar em amigos como um *lista*, ele interpreta assim: “executar as instruções no corpo do `for` uma vez para cada amigo *in* lista de amigos nomeados.”

Olhando para o laço `for`, *for* e *in* são palavras-chave reservadas do Python e `amigo` e `amigos` são variáveis.

```
for amigo in amigos:
    Print ('Feliz Ano Novo: ', amigo)
```

Em particular, `amigo` é a *variável de iteração* o laço `for`. A variável `amigo` muda a cada iteração do laço e controla quando o laço deve ser concluído. A *variável de iteração* percorre sucessivamente as três palavras armazenadas na variável `amigos`.

## 5.6 Padrões de laço

Muitas vezes usamos um laço `for` ou `while` para percorrer uma lista de itens ou conteúdos de um arquivo, nesse caso nós estamos procurando por algo como o maior ou menor valor dos dados que nós percorremos.

Esses laços são geralmente construídos:

- Inicializando uma ou mais variáveis antes que o laço comece
- Realizando algum cálculo em cada item no corpo do laço, possivelmente alterando as variáveis no corpo do loop
- Olhando para as variáveis resultantes quando o loop é concluído

Utilizaremos uma lista de números para demonstrar os conceitos e a construção desses padrões de loop.

### 5.6.1 Contando e somando repetições

Por exemplo, para contar o número de itens em uma lista, escreveríamos o seguinte laço `for`:

```
contador = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    contador = contador + 1
print('Contagem: ', contador)
```

Nós ajustamos `contador` para zero antes do laço começar, então nós escrevemos um laço `for` para percorrer a lista de números. Nossa variável de *iteração* é chamada de `itervar` e enquanto nós não usamos o `itervar` no laço, ele controla o laço e faz com que o corpo desse laço seja executado uma vez para cada um dos valores na lista.

No corpo da repetição, adicionamos 1 ao valor atual de `contador` para cada um dos valores na lista. Enquanto o laço está executando, o valor de `contador` é o número de valores que percorremos "até agora".

Depois que a repetição é concluída, o valor de `contador` é o número total de itens. O número total "cai do céu" no final do ciclo. Nós construímos o laço para que tenhamos o que queremos quando ele terminar.

Outro laço semelhante que calcula o total de um conjunto de números é o seguinte:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print('Total: ', total)
```

Nesse loop, *usamos* a *variável de iteração*. Em vez de simplesmente adicionar um a `contagem`, como na repetição anterior, nós adicionamos o número real (3, 41, 12, etc.) ao total durante cada iteração de repetição. Se você pensar na variável `total`, ela contém o "total de valores percorridos até agora". Então, antes que a repetição comece,`total` é zero porque ainda não vimos nenhum valor. Durante todo o laço, `total` é o total atualizado e, ao final do laço, `total` é o valor de todos os valores na lista.

Conforme o laço é executado, `total` acumula a soma dos elementos; uma variável usada dessa forma às vezes é chamada de *acumulador*.

Nem o laço de contagem nem o de soma são particularmente úteis na prática, porque existem funções internas `len()` e `sum()` que calculam o número de itens em uma lista e o total dos itens em uma lista, respectivamente.

### 5.6.2 Repetições máximas e mínimas

Para achar o máximo valor em uma lista ou sequência, construímos a seguinte repetição:

```
máximo = None
print('Antes:', máximo)
for itervar in [3, 41, 12, 9, 74, 15]:
    if máximo is None or itervar > máximo :
        máximo = itervar
    print('Laço:', itervar, máximo)
print('Máximo:', máximo)
```

Quando o programa é executado, a saída é a seguinte:

```
Antes: None
Laço: 3 3
Laço: 41 41
Laço: 12 41
Laço: 9 41
Laço: 74 74
Laço: 15 74
Maximo: 74
```

A variável `máximo` é melhor considerada como o "maior valor que vimos até agora". Antes da repetição, definimos `máximo` como a constante `None`. `None` é uma constante de valor especial que podemos armazenar em uma variável para marcá-la como "vazia".

Antes do início do laço, o maior valor que temos até agora é `None`, já que nós ainda não percorremos nenhum valor. Enquanto o laço está sendo executado, se o `máximo` for `None`, então nós pegamos o primeiro valor que vemos como o maior até agora. Você pode ver na primeira iteração quando o valor do `itervar` é 3, já que `máximo` é `None`, nós imediatamente definimos `máximo` como sendo 3.

Após a primeira iteração, `máximo` não é mais `None`, então a segunda parte da expressão lógica composta que verifica se `itervar > máximo` é acionada apenas quando vemos um valor que é maior que o "maior até agora". Quando vemos um novo "ainda maior", nós levamos esse novo valor para o `máximo`. Você pode ver na saída do programa que `máximo` progride de 3 para 41 e de 41 para 74.

No final do laço, verificamos todos os valores e a variável `máximo` agora contém o maior valor da lista.

Para calcular o menor número, o código é muito semelhante com uma pequena alteração:

```
mínimo = None
print('Antes:', mínimo)
for itervar in [3, 41, 12, 9, 74, 15]:
    if mínimo is None or itervar < mínimo:
        mínimo = itervar
    print('Laço:', itervar, mínimo)
print('Mínimo:', mínimo)
```

Novamente, `mínimo` é o "menor até agora" antes, durante e depois que o laço é executado. Quando o laço concluir, `mínimo` contém o menor valor da lista.

Novamente, como na contagem e na soma, as funções incorporadas `max()` e `min()` fazem esses laços desnecessários.

A seguir, uma versão simples da função embutida `min()` do Python:

```
def min(valores):
    mínimo = None
    for valor in valores:
        if mínimo is None or valor < mínimo:
            mínimo = valor
    return mínimo
```

Na versão da função para encontrar o menor valor, nós removemos todas os comandos `print`, de modo a ser equivalente à função `min` que já está embutida no Python.

## 5.7 Debugging

Ao começar a escrever programas maiores, você pode se encontrar gastando mais tempo com debugging. Mais código significa mais chances de cometer um erro e mais lugares para que erros se escondam.

Um modo de reduzir seu tempo de debugging é "depurar por bisseção". Por exemplo, se existes 100 linhas em seu programa e você verifica uma de cada vez, isso levaria 100 passos.

Em vez disso, experimente quebrar o problema pela metade. Olhe para o meio do programa, ou perto dele, para um valor intermediário que você pode verificar. Adicione o comando `print` (ou alguma outra coisa que tenha um efeito de verificação) e execute o programa.

Se a verificação no ponto médio estiver incorreta, o problema deve estar na primeira metade do programa. Se está correta, o problema está na segunda metade.

Toda vez que você faz um teste como esse, você reduz ao meio o número de linhas que você tem que pesquisar. Após seis passos (o que é muito menos que 100), você ficaria com uma ou duas linhas de código, pelo menos em teoria.

Na prática nem sempre é claro qual é o "meio do programa" e nem sempre é possível verificá-lo. Não faz sentido contar linhas e encontrar o ponto médio exato. Em vez disso, pense sobre lugares no programa onde pode haver erros e lugares onde é fácil fazer um teste. Em seguida, escolha um ponto em que você acha que as chances são as mesmas que o erro esteja antes ou depois do teste.

## 5.8 Glossário

Acumulador: Uma variável usada em um laço para adicionar ou acumular um resultado. Contador: Uma variável usada em um laço para contar o número de vezes que alguma coisa aconteceu. Nós inicializamos um contador em zero e incrementamos o contador cada vez que queremos "contar" alguma coisa. Decremento : Uma

atualização que diminuiu o valor de uma variável. Inicializar: Uma atribuição que fornece um valor para uma variável que vai ser atualizada. Incremento: Uma atualização que aumenta o valor de uma variável (geralmente em um) . Laço infinito: Um laço cuja condição de término nunca é satisfeita, ou para qual não existe condição de término. Iteração: Execução repetida de um conjunto de instruções usando uma função que chama a si mesma ou a um laço. Iteração

## 5.9 Exercícios

**Exercício 1: Escreva um programa que lê repetitivamente números até que o usuário digite "pronto". Quando "pronto" for digitado, mostre a soma total, a quantidade e a média dos números digitados. Se o usuário digitar qualquer coisa que não seja um número, detecte o erro usando o `trye` o `except` e mostre na tela uma mensagem de erro e pule para o próximo número.**

```
Digite um número: 4
Digite um número: 5
Digite um número: dado errado
Entrada Inválida
Digite um número: 7
Digite um número: pronto
16 3 5.333333333333333
```

**Exercício 2: Escreva outro programa que pede por uma lista de números como mostrada acima e mostra, no final, o máximo e o mínimo dos números ao invés da média.**