RACIOCÍNIO ALGORÍTMICO

Camila Andrade Santos

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Estruturas de repetição em Python

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Reconhecer situações de uso de estruturas de repetição.
- Diferenciar o funcionamento das estruturas de repetição.
- Usar estruturas de repetição na composição de algoritmos.

Introdução

A grande maioria das aplicações atuais está conectada à internet e captura diversos dados a todo momento. Com isso, quando é necessário acessar tais dados, há uma quantidade massiva de informações a extrair deles. Assim, se você precisasse produzir uma linha de código diferente para ler, atuar e gravar cada linha de uma tabela no banco de dados, por exemplo, dificilmente conseguiria concluir um sistema.

Para essa e outras aplicações, as linguagens de programação disponibilizam estruturas de repetições. Assim, em vez de o desenvolvedor repetir linhas de código, determinado trecho dentro de uma estrutura de repetição é executado diversas vezes em tempo de execução. Com isso, frequentemente passa a ser necessária a implementação de apenas uma linha de código.

Neste capítulo, você vai estudar sobre as estruturas de repetição `while` e `for` na linguagem de programação Python. A diferença dessa linguagem é que ela não possui outras estruturas de repetição, como o `while` de outras linguagens. Além disso, você vai ver situações de uso, sintaxe das instruções e exemplos práticos.

1 Aplicação de estruturas de repetição

Ao produzir *softwares*, às vezes você precisa, dentro da lógica de programação, executar mais de uma vez o mesmo comando ou um conjunto de comandos, de acordo com uma condição (MANZANO; OLIVEIRA, 2010). Para isso, as linguagens de programação disponibilizam estruturas de repetição.

Imagine um código responsável por calcular a tabuada de determinado número inserido pelo usuário. Observe a Figura 1, que exemplifica duas soluções para o código da tabuada do valor 3.

```
print(3*1)
print(3*2)
print(3*3)
print(3*4)
print(3*5)
print(3*6)
print(3*7)
print(3*8)
print(3*9)
```

```
for i in range(1,10):
    print(3*i)
```

Figura 1. Exemplo de repetição de código em duas soluções para implementação da tabuada do 3.

No código da esquerda, temos a tabuada do 3, mas com uma repetição de código. Porém, quando queremos desenvolver um código limpo, essa prática não é recomendada. A repetição de código pode aumentar os custos de manutenção do sistema, uma vez que se torna difícil a compreensão do código. Além disso, caso você queira alterar algo no cálculo, por exemplo, precisará modificar nove linhas, e não apenas uma.

Já no código da direita, temos o mesmo resultado da tabuada do 3, mas com um código mais limpo e otimizado. Para isso, utilizamos a estrutura de repetição `for` em que a variável `i` assume valores de 1 a 9, incrementados de 1 em 1 durante a execução do laço de repetição. Dentro desse laço, consta o trecho de código que será repetido — no caso do exemplo, ele é representado pela impressão do 3 multiplicado pelo valor da variável `i`. Com isso, ao final da execução, temos o mesmo resultado, mas de forma mais rápida e prática.

Tais trechos que são repetidos durante a execução do código podem ter um número de repetições indeterminado. No entanto, ele deve ser necessariamente finito; caso contrário, o seu código pode entrar em *looping* infinito. Com isso,

em um sistema *web*, por exemplo, a repetição infinita pode pesar o servidor, e a aplicação pode ficar fora do ar por tempo indeterminado.

Porém, há casos em que o *looping* infinito é desejado, como em aplicações com microcontroladores, que normalmente executam o código de forma estruturada. Nesses casos, há a necessidade da verificação constante das portas e dos módulos externos. Com isso, o código fica em *looping* do momento em que o *hardware* é ligado até o seu desligamento (MATTHES, 2015).

2 Estruturas de repetição `while` e `for`

Em Python, existem duas possibilidades para construir laços de repetição: utilizando a instrução `while` ou a instrução `for`. A estrutura de repetição `while` repete um bloco de código enquanto a condição da instrução for verdadeira. Utilizamos essa instrução quando não sabemos quantas vezes determinado bloco de código precisa ser repetido. Assim, a condição da instrução servirá como parada para sair do *looping* (MENEZES, 2010). Em Python, o formato da estrutura de repetição é apresentado a seguir:

```
while < condição >:
     bloco de código
```

Como você pode observar, utilizamos a palavra reservada `while`, seguida da condição para continuar a execução do código e pelo sinal de dois-pontos (:). Abaixo da declaração da instrução, temos o código a ser executado repetidas vezes. Observe o exemplo do código:

```
x = 1
while x<=3:
    print(x)
    x = x + 1
```

Esse código efetua o incremento da variável `x`, iniciando em 1; a condição de parada do laço `while` é "`x` maior que 3", ou seja, o código será repetido enquanto `x` for menor ou igual a 3. Na linha 1, declaramos a variável `x` recebendo o valor 1. Na linha 2, combinamos uma estrutura condicional com uma estrutura de repetição, ou seja, perguntamos se `x` é menor ou igual a 3. Enquanto a resposta for verdadeira, o trecho de código abaixo dessa linha será executado.

Quando for falsa, a execução sairá do laço de repetição. Na linha 3, imprimimos o valor de `x` e, na linha 4, fazemos o incremento do valor de `x`: ele recebe a soma do valor dele mesmo mais 1. Após a execução da linha 4, o código retorna para a linha 2, e o novo valor de `x` da soma é atribuído para o `x` da condição. Com isso, a condição é verificada para decidir se as linhas 3 e 4 serão executadas novamente. Veja o teste de mesa desse código no Quadro 1.

Quadro 1. Teste de mesa do código anterior

Iteração	Valor de x	Condição x<=3
1	1	Verdadeiro
2	2	Verdadeiro
3	3	Verdadeiro
4	4	Falso

Observe que, na quarta iteração do código, o valor retornado é falso para a condição; logo, a execução sai do laço de repetição. Agora, imagine que o mesmo código vai repetir o trecho de código de acordo com o valor de parada que o usuário inserir, como mostra a Figura 2.

```
numeroFim = int(input("Digite o número para a condição de parada: "))
x = 1
while x<=numeroFim:
    print(x)
    x = x + 1
```

Figura 2. Conceito de contador em Python.

Observe que adicionamos a variável `numeroFim`, que recebe do usuário um número para ser utilizado como condição de parada. Isso significa que o código agora não está mais fixo somente a três repetições verdadeiras, e sim a quantas repetições o usuário desejar. A finalidade da variável `x` dentro do laço de repetição é que, a cada execução, ela é atualizada e incrementada de 1 em 1. Dependendo da necessidade, a variável também pode assumir outros valores de incremento.

Quando existe um incremento dentro de um laço de repetição, dizemos que essa variável tem o papel de **contador** dentro de uma estrutura de repetição. Trata-se de uma variável auxiliar que "contará" as execuções e auxiliará no momento de calcular a condição de parada do laço (MENEZES, 2010).

Além das variáveis contadoras, também pode ser muito útil, em laços de repetição, utilizar variáveis acumuladoras, isto é, variáveis que acumulam valores (MENEZES, 2010). Você pode utilizá-las, por exemplo, em situações em que deseja acumular o resultado de uma soma. Observe o exemplo da Figura 3.

```
contador = 1
soma = 0
while contador <= 5:
    numero = int(input("Digite o número a ser acumulado: "))
    soma = soma + numero
    contador = contador + 1
print("Soma: ", soma)
```

Figura 3. Exemplo de acumulador em Python.

Na linha 1, declaramos a variável `contador`, que iniciará com valor 1. Na linha 2, declaramos a variável `soma`, que é inicializada com o valor 0. Na linha 3, temos a condição de que o trecho do código será executado enquanto a variável `contador` for igual ou menor a 5. Na linha 4, a variável `numero` armazena o valor a ser inserido pelo usuário. Esse valor será acumulado na variável `soma` (linha 5), que recebe a soma do seu valor antigo com o valor inserido pelo usuário. Portanto, se em `soma` havia o valor 10 e o usuário digitou 5, a variável `soma` vai assumir o valor 15. Na linha 6, temos o contador, responsável por contar as execuções do *loop*.

De forma geral, pode haver confusões na distinção entre uma variável acumuladora e uma contadora, pois ambas podem assumir operações de soma sobre os seus próprios valores. Como você pode observar na Figura 3, a variável `contador` também está acumulando o valor que ela assume mais o valor 1. Assim, você pode compreender melhor a distinção dos dois conceitos analisando o código e observando a finalidade de ambos os tipos de variáveis. No contador, há um valor de incremento definido, que é utilizado na condição de repetição do laço; já a variável acumuladora acumula a soma de valores aleatórios. Na linha 7, imprimimos na tela o valor da variável acumuladora.

Em algumas situações, é necessário interromper um laço de repetição por motivos específicos da aplicação. Por exemplo, se durante um intervalo do laço a variável `x` assumir determinado valor, o laço deve ser interrompido, dando sequência ao código. Ainda, quando utilizamos o laço `while`, a estrutura verifica somente a condição de parada no início de cada repetição e, dependendo do problema, interromper o laço no meio da execução seria interessante. A instrução `break` é utilizada para interromper um laço de repetição, independentemente do valor atual da sua condição. Observe a Figura 4.

```
soma = 0
while True:
    parada = int(input("Digite um valor para a soma ou 0 para sair: "))
    if parada == 0:
        break
    soma = soma + parada
print("Soma: ", soma)
```

Figura 4. Utilização da instrução `break` em um laço `while`.

Na linha 1, temos a variável responsável por efetuar a operação de soma (variável acumuladora). Na linha 2, temos a declaração da instrução `while`, bem como a sua condição de parada. Passando `True` ou o valor 1 como parâmetro, a condição do laço `while` torna-se verdadeira sempre. Como não há variáveis na condição a serem alteradas, fazendo a condição se tornar falsa, a instrução `while` entra em *looping* infinito.

Na linha 3, temos a instrução `input`, responsável por receber um valor do usuário. Esse usuário, por sua vez, pode digitar qualquer valor e, quando ele for diferente de 0, esse valor será acumulado na variável soma. Quando o valor for igual a 0, o laço é interrompido e o código sai do laço de repetição, executando o restante do código.

Na linha 4, há a condição para sair do código, em que se busca pelo valor inserido pelo usuário igual a 0. Na linha 5, há a instrução de parada `break`, que interrompe o laço de repetição atual. Então, caso houvesse outro laço de repetição circundando o `while`, a execução retornaria para esse laço. Se não entrar na condição `if`, será executada a linha 6, responsável por acumular os valores na variável soma. Quando o laço for interrompido por `break`, a linha 7 será executada, apresentando na tela o resultado do acúmulo de valores inseridos pelo usuário.

Uma estrutura de repetição envolta por outra estrutura de repetição é chamada de **estrutura de repetições aninhadas** (MENEZES, 2010). Considere

como exemplo o código de multiplicação da Figura 1, alterando-o para um código com estruturas de repetição aninhadas que efetua o cálculo da tabuada de 1 a 10. Observe a Figura 5.

```
tabuada = 1
while tabuada <= 10:
    numero = 0
    while numero <= 10:
        mutliplicacao = tabuada * numero
        print(tabuada," * ",numero," = ", mutliplicacao)
        numero = numero +1
    print("-----")
    tabuada = tabuada + 1
```

Figura 5. Tabuada do 1 ao 10.

O código da Figura 5 efetua o cálculo da tabuada de 1 a 10. Na linha 1, temos a variável `tabuada`, que vai iniciar de 1. Na linha 2, temos a condição do primeiro `while` , em que já conseguimos imaginar qual tabuada vai ser incrementada de 1 até 10. Na linha 3, temos a variável `numero`, que indica o valor que a tabuada vai multiplicar, iniciando de 0. Na linha 4, temos a condição do segundo `while`, em que a variável `numero` vai incrementar de 0 a 10.

Na linha 5 está o cálculo de multiplicação em si. Os laços de repetições aninhados dessa linha são responsáveis pela geração da tabuada. Isso se dá porque, enquanto o valor de `tabuada` é 1, o laço de repetição interno será executado e incrementado, gerando 1*0, 1*1, 1*2, e assim por diante. Quando a execução sair do laço interno, vai incrementar o laço externo, gerando os cálculos 2*0, 2*1, 2*2, 2*3, e assim por diante. Ao final, teremos a nossa tabuada.

Além do laço de repetição `while`, em Python, há também o laço de repetição `for`. Observe na Figura 6 a sintaxe da instrução `for` em Python.

```
for <variavel> in <objeto iteravel>:
    bloco de codigo
```

Figura 6. Sintaxe da instrução `for`.

A instrução `for` exige uma variável, que vai receber um valor de um objeto iterável, como uma lista de valores. A instrução `in` indica que a variável vai receber algo que está contido no objeto iterável. Veja dois exemplos de utilização dessa instrução na Figura 7.

```
listaString = ["item1", "item2", "item3", "item4", "item5" ]

# Exemplo 1
for item in listaString:
    print(item)

# Exemplo 2
for i in range(0, len(listaString)):
    print(listaString[i])
```

Figura 7. Exemplos de utilização de instruções de laço de repetição com `for`.

Na linha 1, temos uma lista denominada `listaString`, com cinco itens do tipo texto. No primeiro exemplo de utilização do `for`, a variável `item` vai receber um valor de uma posição da lista, iniciando da primeira e indo até a última posição. Dentro do laço de repetição, basta imprimir a variável `item`, que estará carregada dos valores individuais da lista.

No segundo exemplo com o laço `for`, a variável `i` vai receber o índice da posição da lista, e não mais um valor. A posição da lista é definida pelo objeto iterável da instrução `range`, a qual recebe como parâmetro o valor de início, que no nosso exemplo é 0, e o valor de fim, que no exemplo é o tamanho da lista (5). Em alguns casos, é possível passar o incremento, isto é, de 1 em 1, de 2 em 2, e assim por diante. Quando esse parâmetro não é passado, assume-se o incremento padrão, que é 1. Nesse tipo de utilização, precisamos, dentro da instrução `for`, chamar a lista e indicar qual a posição que queremos imprimir — no nosso caso, seria o valor da variável `i`.

3 Exemplos de utilização das instruções `while` e `for`

Imagine uma lanchonete que possui um sistema para armazenar os pedidos dos clientes de modo que sejam atendidos por ordem de venda. Considere

que há três pedidos na fila, e que você deseja acessá-los em ordem. Observe o código na Figura 8.

```
listaPedido=["Hamburguer, Refrigerante e Batata", "Hamburguer", "3 Hamburguer, Refrigerante e Sorvete"]
listaCliente=["Karla", "Paulo", "José"]

for indice in range(0, len(listaPedido)):
    print("Número do Pedido ",indice)
    print("Cliente ", listaCliente[indice])
    print("Itens do Pedido ",listaPedido[indice])
    print("------------------------")
```

```
========== RESTART: D:/Repetição/exemploFastFood.py ==========
Número do Pedido  0
Cliente  Karla
Itens do Pedido  Hamburguer, Refrigerante e Batata
------------------------
Número do Pedido  1
Cliente  Paulo
Itens do Pedido  Hamburguer
------------------------
Número do Pedido  2
Cliente  José
Itens do Pedido  3 Hamburguer, Refrigerante e Sorvete
------------------------
```

Figura 8. Exemplo de acesso a pedidos utilizando a estrutura de repetição `for`.

Nesse código, temos uma lista com os pedidos e uma lista com os clientes. Além disso, cada posição de uma lista está relacionada à posição da outra, ou seja, o cliente que está na posição 0 da `listaCliente` fez o pedido da posição 0 da `listaPedido`. Nesse caso, utilizamos a instrução `for`, em que o iterador é dado pela instrução `range`, de modo que conseguimos acessar as duas listas no mesmo laço `for`.

Se utilizássemos a outra forma do `for`, em que a variável armazena o valor da lista, precisaríamos criar duas instruções `for` (uma para cada lista), o que faria com que o código não fosse otimizado. Além disso, como estamos trabalhando com a posição dos índices da lista, conseguimos acessar o número do pedido de acordo com a sua posição na lista.

Agora observe um exemplo de um sistema responsável por monitorar a temperatura ambiente de um local. Ao ligar o dispositivo, o usuário configura em qual temperatura o ambiente entra em estado crítico, e deseja receber um alerta quando a temperatura alcançar o índice configurado. Observe o código na Figura 9.

```
tempConfig=float(input("Digite a temperatura em estado crítico "))
temperaturaAmbiente = 0
while True:
    temperaturaAmbiente = temperaturaAmbiente + 5
    if temperaturaAmbiente > tempConfig:
        print(" O ambiente está com a temperatura em estado crítico !!!" )
        break
    print(" O ambiente está com a temperatura controlada")


================ RESTART: D:/Repetição/exemploTemperatura.py ================
Digite a temperatura em estado crítico 35
 O ambiente está com a temperatura controlada
 O ambiente está com a temperatura controlada
 O ambiente está com a temperatura controlada
 O ambiente está com a temperatura controlada
 O ambiente está com a temperatura controlada
 O ambiente está com a temperatura controlada
 O ambiente está com a temperatura controlada
 O ambiente está com a temperatura em estado crítico !!!
```

Figura 9. Exemplo de monitoramento de temperatura ambiente utilizando o laço `while`.

O usuário inicialmente configura qual a temperatura tida como crítica. No exemplo, digitamos a temperatura de 35°C. Em situações reais, o incremento da variável `temperaturaAmbiente` se dará por sensores específicos para tal finalidade. Para exemplificar a situação, indicamos que essa temperatura subiu rapidamente de 5 em 5 graus. Dessa forma, temos que o sistema se manteve com temperatura controlada em sete iterações e, na oitava, entrou em estado crítico, ou seja, a `temperaturaAmbiente` encontrava-se maior do que a `tempConfig`. Assim, o alerta foi emitido e o sistema foi pausado.

Referências

MANZANO, J. A. N. G.; OLIVEIRA, J. F. *Algoritmos:* lógica para desenvolvimento de programação de computadores. 23. ed. São Paulo: Érica, 2010.

MATTHES, E. *Python crash course:* a hands-on, project-based introduction to programming. San Francisco: No Starch Press, 2015.

MENEZES, N. N. C. *Introdução à programação com Python:* algoritmos e lógica de programação para iniciantes. São Paulo: Novatec, 2010.

Fique atento

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS