Estrutura de repetição for loop, para que serve?

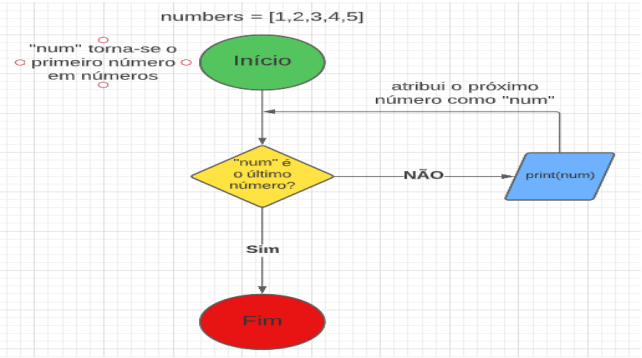
Na linguagem de programação Python, os laços de repetição "for" também são chamados de "loops definidos" porque executam a instrução um certo número de vezes. Isso contrasta com os loops while, ou loops indefinidos, que executam uma ação até que uma condição seja atendida e eles sejam instruídos a parar.

For loops são úteis quando você deseja executar o mesmo código para cada item em uma determinada sequência. Com um loop for, você pode iterar sobre qualquer dado iterável, como listas, conjuntos, tuplas, dicionários, intervalos e até strings.

Para entendermos melhor o funcionamento de um for em Python, vejamos a seguir, um exemplo utilizando um fluxograma. Sabemos que um loop for repete instruções enquanto o último item no intervalo ainda não foi alcançado. Logo, Vamos criar um loop for simples usando Python. Este loop imprime os números de uma lista:

```
>>numbers = [1, 2, 3, 4, 5]
>>for number in numbers:
>>    print(number)
```

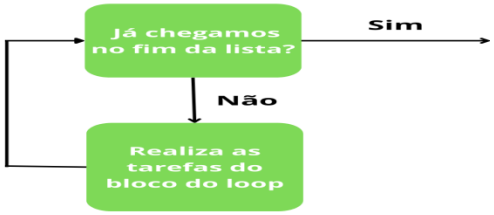
Aqui, a instrução **print(number)** é executada enquanto houver números restantes na lista. O fluxograma que descreve o processo é o seguinte:



O que é iteração?

A iteração é um processo de repetição que executa o mesmo bloco de código até o **for** chegar no final da iteração, ou seja, no último valor. De outro modo, a iteração pode chegar ao fim quando tem a sua condição falsa.

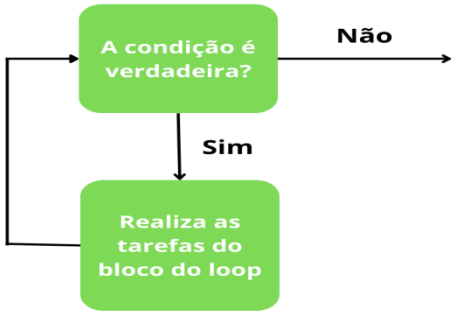
O que é iteração definida?



A **iteração definida** é uma repetição que tem como base um objeto iterável, por exemplo, uma **lista**, um **objeto**, uma **Collection**, um **Iterator**, etc. Nessa iteração, realizamos as operações de um bloco de código do início ao fim do objeto iterável.

Essa iteração pode ser utilizada de diversas formas, conseguimos ler os valores individuais, editá-los e até mesmo usá-los para realizar alguma operação única. No Python, o **for** é o único capaz de realizar essa lógica.

O que é iteração indefinida?



A iteração indefinida realiza operações repetitivas com base em uma condicional, semelhante a declaração **if** e **else**.

Essa iteração pode ser muito perigosa já que se a condicional não tiver o valor falso, o loop não se fechará e poderá até mesmo travar o processamento da aplicação. Aqui, o **while** é o único capaz de realizar essa lógica no Python.

É importante entender o conceito de **iterável** em Python. Basicamente, um objeto iterável é aquele que tem a capacidade de retornar cada um de seus elementos de forma individual. Uma forma de identificar se um objeto é iterável é conferir se ele implementa os métodos **__iter__()**, que obtém um objeto iterador e o **__next__()**, que faz a passagem para o próximo elemento.

Outro conceito importante é o de **iterador**, que além de representar um item do fluxo de dados, também é um objeto iterável e, portanto, implementa as funções **__iter__()** e **__next__()**. Essa implementação é transparente para nós, pois ela é feita e controlada pelo for.

Na prática, quando o laço **for** é inicializado, ele executa a instrução ou o bloco de códigos uma vez e utiliza uma referência, que funciona como um índice, para indicar o próximo elemento da sequência. Internamente, ele usa as funções **__iter__()** e **__next__()** para que a estrutura de repetição funcione. Por isso, os objetos manipulados com essa estrutura precisam ser iteráveis.

Como usar o Python for

Antes de demonstrarmos como o comando for loop atua, é importante fazer um comparativo com outras linguagens de programação, pois essa estrutura funciona um pouco diferente em Python.

Em algumas linguagens de programação, como em C, JavaScript etc, o for loop é utilizado com uma variável do tipo inteiro, que funciona como um índice e serve para controlar a quantidade de vezes que a repetição será feita. É algo como:

```
for i = 1 to 10
    print(i)
```

O for não funciona dessa maneira em Python. O código feito como o do exemplo acima retorna um erro de execução, pois **a estrutura de repetição for só funciona com os tipos de dados que permitem iterações** e o tipo numérico não tem essa característica.

Sintaxe base do Python for?

```
for <item> in <conjunto_de_itens>:
    <bloco_de_codigo>
```

No qual:

- **item**: corresponde a cada elemento presente na variável que permite a iteração;
- **conjunto_de_itens**: pode ser uma lista, uma string, uma tupla, um dicionário ou um objeto que permita iterações.

Veja um código de exemplo:

```
>>frutas = ['Abacaxi', 'Morango', 'Uva']
>>for fruta in frutas:
>>    print(fruta)
```

No código fonte acima utilizamos uma lista, que é um tipo de dados em Python que permite iterações. Perceba que a variável "frutas" foi declarada anteriormente no código para que o laço de repetição funcione. Outra opção seria escrever a lista diretamente na instrução, como mostramos abaixo:

```
>>for fruta in ['Abacaxi', 'Morango', 'Uva']:
>>    print(fruta)
```

Repetindo os caracteres de uma string!

Como mencionamos, um dos tipos de dados que podem ser utilizados com a estrutura de repetição **for** é a **string**. A linguagem Python separa cada caractere como um elemento iterável e podemos utilizar o loop para acessar o seu conteúdo. Veja um exemplo:

```
>>sentenca = "Vamos estudar Python"
>>for letra in sentenca:
>>    print(letra)
```

Ao executarmos o código de exemplo acima, perceba que o caractere de espaço também foi considerado válido e a linha correspondente ficou em branco.

Configurando onde o loop deve parar: break

Por padrão, a estrutura de repetição só termina depois de ler o último elemento da variável iterável. Entretanto, é possível modificar essa condição e interromper o loop no meio do caminho. Para isso, utilizamos a instrução **break**, que encerra a execução do loop ao encontrar uma condição específica.

Devemos utilizar a instrução **break** em conjunto com uma estrutura condicional, como a **if/else** ou até mesmo com outro laço de repetição **for**. Veja como fica a sintaxe da estrutura de repetição quando utilizamos o break:

```
Sintaxe:

for <item> in <conjunto_de_itens>:
    <bloco_de_codigo>
    if <condicao_verdadeira>:
         <outras_instrucoes>
         break
```

Veja em um exemplo prático:

```
>>pessoas = [({'nome': 'João', 'cidade': 'Belo Horizonte'}),
>>           ({'nome': 'Maria', 'cidade': 'São Paulo'}),
>>           ({'nome': 'Pedro', 'cidade': 'Curitiba'})]
>>contador = 0
>>for pessoa in pessoas:
>>    contador += 1
>>    print(contador)
>>    if pessoa['nome'] == 'Maria':
>>       print(pessoa['nome'], "mora em", pessoa['cidade'])
>>       break
```

No código acima, utilizamos o tipo de dados dicionário, que é usado para armazenar valores em pares. Nele, o primeiro elemento corresponde à chave e o segundo, ao valor.

Perceba que criamos uma lista de pessoas e utilizamos a estrutura de repetição **for** para percorrer cada item da lista. Também usamos a estrutura condicional **if** para verificar em cada pessoa da lista a que tem o nome de Maria. Ao encontrá-la, exibimos na tela a frase “Maria mora em São Paulo”. A seguir usamos a instrução **break** para interromper o loop, pois já encontramos a pessoa que queríamos.

Além disso, repare que criamos uma variável chamada contador que imprime o número de vezes que o loop foi executado. Isso foi feito para demonstrar o funcionamento da instrução **break**, pois sem ela o loop seria executado três vezes, já que temos uma lista com três pessoas.

Interrompendo o loop e continuando no próximo objeto: continue

Assim como podemos interromper a execução da estrutura de repetição, também podemos pular para o próximo item. Isso é feito por meio da instrução **continue** em conjunto com uma

validação, que pode ser feita com uma estrutura condicional ou outro laço de repetição. Veja o exemplo abaixo:

```
>>pessoas = [({'nome': 'João', 'cidade': 'Belo Horizonte'}),
>>           ({'nome': 'Maria', 'cidade': 'São Paulo'}),
>>           ({'nome': 'Pedro', 'cidade': 'Curitiba'})]
>>contador = 0
>>for pessoa in pessoas:
>>    contador += 1
>>    if pessoa['nome'] == 'Maria':
>>       continue
>>    print(contador)
>>    print(pessoa['nome'], "mora em", pessoa['cidade'])
```

Perceba que modificamos o código do exemplo anterior. Agora, pulamos para o próximo item ao encontrarmos a pessoa de nome “Maria”. Veja que exibimos na tela o contador e a frase com o nome e a cidade de cada elemento da lista. Como utilizamos a instrução **continue**, a exibição do contador e a frase não foram exibidas para o segundo elemento.

Retornando uma sequência de números: range().

Antes de falarmos sobre o **range()** na estrutura de repetição **for**, vamos ver brevemente o que ele faz. A função **range()** retorna uma série de números consecutivos. Por padrão, ela inicia no número 0 e é incrementada adicionando 1.

O comando range(4), por exemplo, retornará o seguinte valor : “0, 1, 2, 3”, pois ao chegar ao número 4, o loop será concluído. A sintaxe da função **range()** é:

```
range(início, parada, incremento)
```

No qual:

- **início**: é um valor opcional e corresponde a partir de qual número o range será iniciado;
- **parada**: é um valor obrigatório e indica o número de parada do range;
- **incremento**: é opcional e indica o valor que queremos adicionar entre um item e outro.

A função **range()** é utilizada na estrutura de repetição **for** para executarmos um determinado conjunto de instruções pela quantidade de vezes indicadas na função. Veja um exemplo:

```
>>for numero in range(10):
>>    if numero % 2 == 0:
>>       print("O número", numero, "é par")
```

No código acima, utilizamos a função **range()** para descobrir os números pares em um determinado intervalo numérico. Perceba que no resultado do processamento o número 10 não foi listado. Como mencionamos, a função **range()** inicia a contagem a partir do número 0 e o valor 10 corresponde ao ponto de parada do loop.

Portanto, se quiséssemos considerar o número 10 nesse código, teríamos que adequar o valor da função para range(11). Também podemos definir um escopo diferente para o range. Veja o código a seguir:

```
>>for numero in range(10, 21):
>>    if numero % 2 == 0:
>>        print("O número", numero, "é par")
```

Perceba que nesse exemplo utilizamos o range(10, 21). Na prática, dissemos ao compilador para considerar que o primeiro número será o 10 e o ponto de parada será no número 21. Nesse caso, o número 20 foi listado como número par.

Executando um código quando o loop chega ao fim: else no for.

A estrutura de repetição **for** também pode ser utilizada com a cláusula **else**. Na prática, ela funciona quando o loop é encerrado sem nenhuma interrupção, como se utilizássemos a instrução **break**. É importante dizer que a cláusula **else** na estrutura de repetição **for** é opcional. Veja a sintaxe do loop for com a instrução **else**:

```
Sintaxe:

for <item> in <conjunto_de_itens>:
   <bloco_de_codigo>
else:
   <novo_bloco_de_codigo>
```

Exemplo:

```
>>frutas = ['Abacaxi', 'Saputí', 'Caju']
>>for fruta in frutas:
>>    print(fruta)
>>else:
>>    print("Laço de repetição finalizado.")
```

Laços aninhados

Existem situações em que precisamos percorrer outra variável iterável dentro de uma estrutura de repetição. Para isso, utilizamos um loop dentro do outro. Vale ressaltar que é importante ter cuidado com esse tipo de implementação para não criar códigos com muitas repetições aninhadas, pois ele se torna confuso e de difícil manutenção. Entretanto, é preciso entender como utilizar laços aninhados. Veja um exemplo:

```
>>for numero_coluna1 in range(2, 5):
>>    print("Tabuada do ", numero_coluna1)
>>    for numero_coluna2 in range(11):
>>        print(numero_coluna1, "x", numero_coluna2, " = ",
              numero_coluna1 * numero_coluna2)
```

No código acima utilizamos a estrutura de repetição **for** para construir a tabuada dos números 2, 3 e 4. Perceba que também utilizamos a função **range()** para delimitar os números da tabuada.

Como executar um loop sem conteúdo: pass

Quando utilizamos a estrutura de repetição **for** na linguagem Python, não podemos deixá-la em branco, ou seja, sem executar nenhuma instrução. Porém isso pode ser necessário, ainda mais durante a etapa de desenvolvimento de uma aplicação, em que queremos deixar a implementação interna para um segundo momento, mas queremos sinalizar que será preciso ter um loop naquele ponto do código.

Uma forma de fazer isso é por meio da instrução **pass**. Na prática, ela permite a criação da instrução sem que nenhuma ação seja executada. Veja um exemplo:

FONTE: disponível em: https://blog.betrybe.com/python/python-for/ acessando em 03nov2022.