



FOR, WHILE, LOOPS E ESTRUTURAS DE REPETIÇÃO NO PYTHON

Nesse ebook vamos falar sobre as famosas Estruturas de Repetição (ou loops) do Python: for, while, for com else, for com range, enumerate e muito mais!

Crie Ebooks técnicos incríveis em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código!**



Chega de formatar código no Google Docs



Deixe que nossa IA faça o trabalho pesado

 Syntax Highlight

 Adicione Banners Promocionais

 Edite em Markdown em Tempo Real

 Infográficos feitos por IA

TESTE AGORA 

Salve salve **Pythonista!**

Nesse post vamos falar sobre as famosas **Estruturas de Repetição** (ou *loops*) do Python: `for` , `while` , `for` com `else` , `for` com `range()` e **muito mais!**

Vamos ver também sobre as funções `range()` e `enumerate()` e como construir *loops* com elas!

Loops ou estruturas de repetição são blocos básicos de qualquer linguagem de programação e são muito importantes!

Cada linguagem de programação possui uma sintaxe específica para *loops*.

Vamos ver nesse post como podemos fazer *loops* em Python, pois isso é muito importante no dia a dia do verdadeiro Pythonista!

Então... **Bora pro post!** 

Introdução

As estruturas de repetição são recursos das linguagens de programação responsáveis por executar um bloco de código repetidamente através de determinadas condições específicas.

O Python contém dois tipos de estruturas de repetição: `for` e `while` .

Vamos ver ambos nesse post!

Loops utilizando `for`

Vamos iniciar pelos detalhes das estruturas de repetição, ou melhor dizendo, os loops utilizando o `for`.

O `for` é utilizado para **percorrer** ou **iterar** sobre uma sequência de dados (seja esse uma lista, uma tupla, uma string), executando um **conjunto de instruções** em cada item.

Como você já sabe, o Python utiliza identação para separar blocos de código: nos loops utilizando `for` não é diferente.

Sua sintaxe básica é: `for <nome variável> in <iterável>`. Vamos entender: - `<nome variável>` é o nome da variável que vai receber os elemento de `<iterável>`. - `<iterável>` é o container de dados sobre o qual vamos iterar, podendo ser: uma lista, uma tupla, uma string, um dicionário, entre outros.

Vamos ver um exemplo, para facilitar nossa vida!

```
lista = [1, 2, 3, 4, 5]

for item in lista:
    print(item)
```

Vamos entender passo a passo: - Na primeira iteração, `item` vai receber o valor do primeiro elemento da lista `lista`, que é 1. Portanto `print(item)` vai mostrar o valor 1. - Na segunda iteração, `item` vai receber o valor do segundo elemento da lista `lista`, que é 2. Portanto `print(item)` vai mostrar o valor 2. - E assim por diante até o último valor, que é 5 (*você já entendeu 😊*)

Existe outra forma de se utilizar o `for` que é utilizando a estrutura `for/else`.

Adicionar o `else` ao final do `for` nos possibilita executar um bloco de código após o iterável ter sido **completamente** percorrido.

Vamos ao exemplo! 

```
for item in sequencia:  
    print(item)  
else:  
    print('Todos os items foram exibidos com sucesso')
```

Exemplos de loops com `for`

Agora vamos ver alguns exemplo de como podemos percorrer tipos de dados diferentes utilizando o `for`.

Para iterar sobre **Listas** e imprimir cada item da lista:

```
computador = ['Processador', 'Teclado', 'Mouse']  
  
for item in computador:  
    print(item)
```

A saída será:

```
Processador  
Teclado  
Mouse
```

 **Quer saber *TUDO* sobre *Listas* e *DOMINAR* essa estrutura de dado tão importante do Python? Então já clica aqui e deixa esse post completo sobre *Listas* na espera!**

Podemos também percorrer os dicionários do Python (que são uma estrutura de dados **muito** importante).

Para isso, podemos fazer da seguinte maneira:

```
notas = {  
    'Português': 7,  
    'Matemática': 9,  
    'Lógica': 7,  
    'Algoritmo': 7  
}  
  
for chave, valor in notas.items():  
    print(f'{chave}: {valor}')
```

O resultado será:

```
Potuguês: 7  
Matemática: 9  
Lógica: 7  
Algoritmo: 7
```

👉 **Quer ficar craque TAMBÉM em Dicionários e saber TUDO? Então também dei-xa de lado esse [post completo sobre os Dicionários do Python](#)**

Também podemos percorrer *strings*, pois elas também são um tipo iterável:

```
for caractere in 'Python':  
    print(caractere)
```

O Python vai dividir a string 'Python' nos caracteres que a compoem e o resultado do loop **for** será o seguinte:

P
y
t
h
o
n

Loops utilizando `while`

O `while` é uma estrutura de repetição utilizada quando queremos que determinado bloco de código seja executado **ENQUANTO** (do inglês *while*) determinada condição for satisfeita.

Em outras palavras: só saia da estrutura de repetição quando a condição não for mais satisfeita.

Sua sintaxe básica é:

```
while <condição>:  
    # Bloco a ser executado
```

Aqui, `<condição>` é uma expressão que pode ser reduzida à `True` ou `False`, podendo ser: - A verificação do valor de uma variável; - Determinada estrutura de dados alcançar um tamanho; - O retorno de uma função se igualar a determinado valor; - Algum valor externo ser alterado (por exemplo um valor armazenado em Banco de Dados).

Vamos entender melhor com um exemplo:

```
contador = 0

while contador < 10:
    print(f'Valor do contador é {contador}')
    contador += 1
```

Não entendeu essa notação no print `print(f'Valor do contador é {contador}')`? Essas são as chamadas f-strings e são formas muito Pythônicas de se formatar strings no Python! Quer saber mais, então adivinha... Claro que sim! Clique aqui para acessar nosso post completo sobre f-strings. 😊

Resultando em:

```
Valor do contador é 1
Valor do contador é 2
Valor do contador é 3
Valor do contador é 4
Valor do contador é 5
Valor do contador é 6
Valor do contador é 7
Valor do contador é 8
Valor do contador é 9
Valor do contador é 10
```

Ou seja, a variável `contador` está sendo incrementada a cada vez que o `while` executa seu código.

Quando ele alcançar o valor 10, a condição `contador < 10` não será mais satisfeita, finalizando o bloco `while`!

Assim como no `for`, podemos utilizar o `else` também nos loops `while`.

Vamos usar o mesmo código do exemplo acima para você entender a diferença:

```
contador = 0

while contador < 10:
    contador += 1
    print(f'Valor do contador é {contador}')
else:
    print(f'Fim do while e o valor do contador é {contador}')
```

O que resultará em:

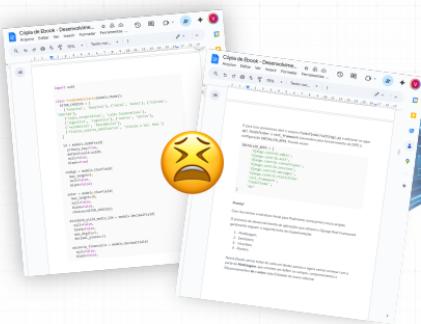
```
Valor do contador é 1
Valor do contador é 2
Valor do contador é 3
Valor do contador é 4
Valor do contador é 5
Valor do contador é 6
Valor do contador é 7
Valor do contador é 8
Valor do contador é 9
Valor do contador é 10
Fim do while e o valor do contador é 10
```



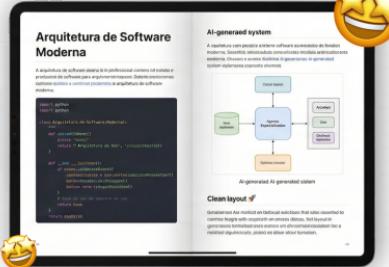
*Estou construindo o **DevBook**, uma plataforma que usa IA para criar e-books técnicos — com código formatado e exportação em PDF. Depois de ler, dá uma passada lá!*

Crie Ebooks técnicos incríveis em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código!**



Chega de formatar código no Google Docs



Deixe que nossa IA faça o trabalho pesado

Syntax Highlight

Adicione Banners Promocionais

Edite em Markdown em Tempo Real

Infográficos feitos por IA

TESTE AGORA! PRIMEIRO CAPÍTULO 100% GRÁTIS

Auxiliadores

Existem 3 comandos que nos auxiliam quando queremos alterar o fluxo de uma estrutura de repetição.

São eles: `break` , `continue` e `pass` .

Esses auxiliares não funcionam diretamente com o `while` , e por isso encaixar eles no bloco principal do `while` pode ser tanto quanto inútil, já que a condição especificada encerra o loop.

Auxiliador `break`

É usado para finalizar um *loop*, isto é, é usado para parar sua execução.

Geralmente vem acompanhado de alguma condição para isso, com um `if`.

Veja um exemplo:

```
for num in range(10):
    # Se o número for igual a = 5, devemos parar o loop
    if num == 5:
        # Break faz o loop finalizar
        break
    else:
        print(num)
```

Saída do código acima:

```
0
1
2
3
4
```

Percebeu que o *loop* não chegou ao final?! Para isso utilizamos o `break` ! 😊

Já com `while`, também podemos utilizar o `break` em uma condição utilizando `if`, assim:

```
num = 0
while num < 5:
    num += 1

    if num == 3:
        break

    print(num)
```

Quando a variável atribuir o valor 4 o laço é finalizado pelo `break`, encerrando o *loop*. Resultando em:

```
1  
2  
3
```

Auxiliador **continue**

Funciona de maneira similar ao `break`, contudo ao invés de encerrar o *loop* ele pula todo código que estiver abaixo dele (dentro do *loop*) partindo para a próxima iteração.

Vamos ao exemplo:

```
for num in range(5):  
    if num == 3:  
        print("Encontrei o 3")  
        # Executa o continue, pulando para o próximo laço  
        continue  
    else:  
        print(num)  
  
    print("Estou abaixo do IF")
```

Repare na saída abaixo. Repare que quando a condição `num == 3` for satisfeita, a string `"Estou abaixo do IF"` não será exibida:

```
0  
Estou abaixo do IF  
1  
Estou abaixo do IF  
2  
Estou abaixo do IF  
Encontrei o 3  
4  
Execução normal
```

Em *loops* com `while` a lógica é a mesma. O `continue` irá finalizar o *loop* atual, iniciando novamente no início do `while`.

Veja o exemplo:

```
num = 0
while num < 5:
    num += 1

    if num == 3:
        continue

    print(num)
```

O resultado desse código é que o 3 não apareça, pois o `print()` que imprime os números está abaixo do `continue`.

Portanto a saída será:

```
1
2
4
5
```

Auxiliador `pass`

O `pass` nada mais é que uma forma de fazer um código que não realiza operação nenhuma.

Tipo o Magikarp do Pokemon 😂 😂 😂

Mas calma, ele tem uma razão de existir no Python!

Como os escopos de Classes, Funções, If/Else e loops `for` / `while` são definidos pela indentação do código (e não por chaves `{}` como geralmente se vê em outras linguagens de programação), usamos o `pass` para dizer ao Python que o bloco de código está vazio.

Veja alguns exemplos:

```
for item in range(5000):
    pass

while False:
    pass

class Classe:
    pass

if True:
    pass
else:
    pass

def funcao():
    pass
```

Caso não utilizemos o `pass`, veja o que acontece:

```
class Classe:

def funcao():
    pass
```

```
File "<stdin>", line 2

    ^
IndentationError: expected an indented block
```

Isso acontece pois o Python entende que as próximas linhas de código fazem parte do mesmo escopo, mas como não estão indentadas um erro `IndentationError` é lançado.

A função `range()`

A função `range` é de grande ajuda quando o tema é repetição, laços, `for` etc...

Ela gera uma sequência de números pela qual podemos iterar!

Com ela, conseguimos especificar: - O inicio de uma sequência; - O passo (ou pulo); e - O valor final da sequência.

Com isso o Python nos entrega uma sequência de números para utilizarmos!

De tão importante que é, temos um post **COMPLETO** sobre essa função tão importante na vida do **verdadeiro Pythonista**!

Então já [clica aqui e acessa nosso post sobre a função `range` agora mesmo!](#)

Função `enumerate()`

Uma dica bem bacana para se usar com o `for`, é a função `enumerate()`.

Ela nos entrega um contador embutido no próprio `for`! 😍

Ao invés de fazer isso:

```
contador = 0
computador = ['Processador', 'Teclado', 'Mouse']

for elemento in computador:
    print(f"Índice={contador} | Valor={elemento}")
    contador += 1
```

Jogue esse `contador` fora e faça **isso**:

```
computador = ['Processador', 'Teclado', 'Mouse']
for indice, valor in enumerate(computador):
    print(f"Índice={indice} | Valor={valor}")
```

Saída:

```
Índice=0 | Valor=Processador
Índice=1 | Valor=Teclado
Índice=2 | Valor=Mouse
```

Esse código é o que chamamos de Pythonico (que segue as melhores práticas da linguagem)! 😊

Outra maneira de iterar sobre os índices, é combinar as funções `range()` e `len()`.

A função `len()` retorna o tamanho de um iterável (lista, tupla, set).

Podemos combiná-los da seguinte forma:

```
computador = ['Processador', 'Teclado', 'Mouse']
for indice in range(len(computador)):
    print(f"Índice={indice} | valor={computador[indice]}")
```

Observe como obteve o mesmo resultado com sua saída:

```
Índice=0 | Valor=Processador  
Índice=1 | Valor=Teclado  
Índice=2 | Valor=Mouse
```

Conclusão

Nesse post vimos um dos pilares da programação Python: as estruturas de repetição com `for` e `while`!

Aprendê-los bem é básico para todos desenvolvedor Python.

Se ficou com alguma dúvida, fique à vontade para deixar um comentário no box aqui embaixo! Será um prazer te responder! 😊

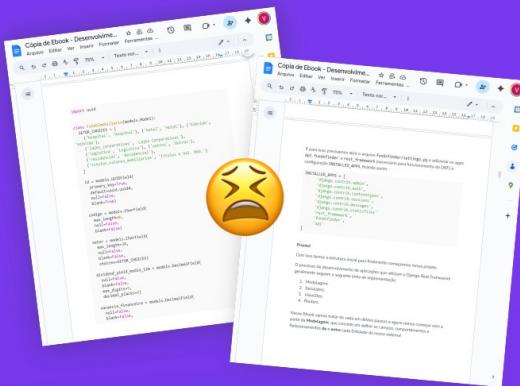
Não se esqueça de conferir!



DevBook

Crie Ebooks técnicos em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código!**



Chega de formatar código no Google Docs



Syntax Highlight



Adicione Banners Promocionais



• Infográficos feitos para...

Deixe que nossa IA faça o trabalho pesado



 Edite em Markdown em Tempo Real

TESTE AGORA



 PRIMEIRO CAPÍTULO 100% GRÁTIS