7

TEXTO BASE

LINGUAGEM DE PROGRAMAÇÃO

Faculdade IMPACTA

Texto base

7

Estrutura de repetição indefinida (*while*)

Prof. Me. Lucio Nunes de Lira
Prof. MSc. Rafael Maximo Carreira Ribeiro

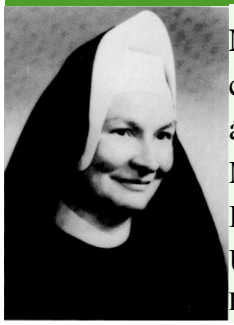
Resumo

Nesta aula os objetivos são: (I) entender a necessidade da estrutura de repetição para controle do fluxo de execução; (II) compreender a estrutura de repetição com quantidade de repetições indefinida: laço while; (III) usar variáveis contadoras, acumuladoras e de sinalização booleana para controlar a execução do while; (IV) conceituar "laço infinito" e suas consequências; (V) utilizar estruturas de repetição combinadas com estruturas de seleção.

7.1. Motivação

Durante nosso curso aprendemos diversos recursos de programação, porém há algo muito importante realizado pelos computadores que precisamos abordar: a repetição de instruções. É natural que programas executem uma sequência de instruções repetidas vezes, aliás essa é uma das principais características em que as máquinas superam os seres humanos. Nesta aula compreenderemos a importância de utilizar uma estrutura de repetição e como escrevê-la na linguagem de programação Python.

VOCÊ CONHECE?



Mary Kenneth Keller, nascida em 1913, foi uma importante freira e cientista da computação, sendo a primeira mulher com doutorado na área e defensora da inclusão de mulheres na computação.

Mary participou do desenvolvimento da linguagem de programação BASIC e fundou um departamento de ciências da computação na Universidade Clarke (Iowa - EUA).

Fonte da imagem: www.zmescience.com/science/woman-computer-science-phd/

7.2. Introdução

Já aprendemos duas formas para controlar o fluxo de execução de um programa: (I) estrutura de controle sequencial, em que o fluxo de execução é dado pela ordem em que as instruções são escritas e; (II) estrutura de controle de seleção ou condicional, onde o fluxo de execução sofre desvios de acordo com o resultado da avaliação de uma condição. Agora aprenderemos a controlar o fluxo de execução de modo que uma sequência de instruções possa ser repetida diversas vezes sem que seja necessário escrevê-la diversas vezes, para tanto usaremos as estruturas de repetição. Para recordar, as três estruturas básicas de controle de fluxo estão ilustradas na Figura 7.1.

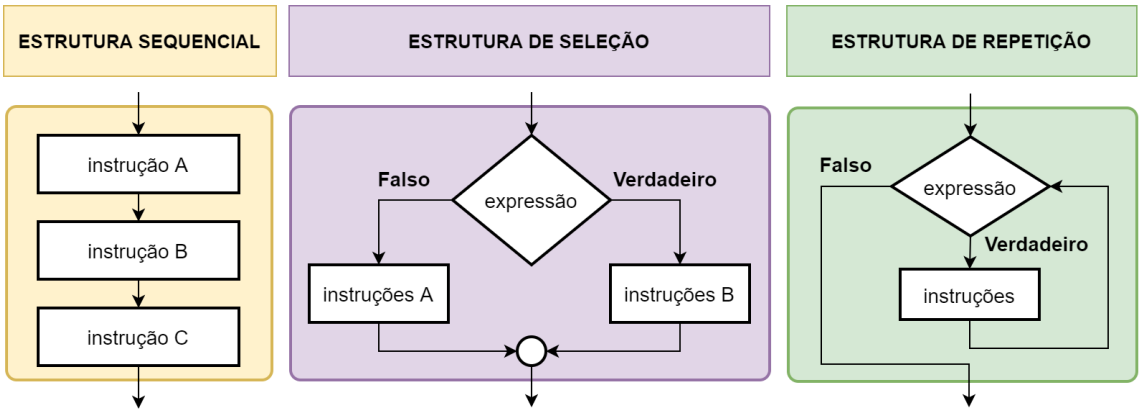


Figura 7.1: Representação em fluxograma das estruturas básicas de controle de fluxo. Fonte: Elaborado pelo autor.

As estruturas de repetição também são conhecidas como *estruturas de iteração*, *malhas de repetição*, *laços* e *loops*. Há um conjunto de problemas que não seriam possíveis de serem resolvidos – ou seriam resolvidos de modo bastante inconveniente – sem o uso de tais estruturas, veremos alguns exemplos no decorrer da leitura.

Os *loops* são aplicados quando precisamos criar algoritmos que demandam a repetição de uma sequência de instruções várias vezes. Vamos iniciar com a análise de um problema simples que desperta a reflexão sobre a necessidade das estruturas de repetição: criar um programa que exiba os cinco primeiros números naturais. Com o conhecimento que já temos, podemos resolver o problema com a Codificação 7.1.

```
print(1)
print(2)
print(3)
print(4)
print(5)
```

Codificação 7.1: Exibição dos cinco primeiros números naturais positivos.

A solução para este problema é simples, no entanto, imagine que ao invés de exibir os cinco primeiros números, o problema exigisse os cinquenta primeiros. Por mais que a lógica aplicada ao novo cenário seja equivalente, a repetição de tantas instruções tornaria o trabalho do programador lento e tedioso, além de aumentar a chance de inserção de instruções incorretas, consequência do aumento do código-fonte.

Agora, modificaremos um pouco o enunciado do problema original. Suponha que devemos exibir os cinco números naturais posteriores ao natural dado pelo usuário como entrada. A Codificação 7.2 é uma solução válida para esse enunciado.

```
x = int(input('valor: '))
print(x + 1)
print(x + 2)
print(x + 3)
print(x + 4)
print(x + 5)
```

Codificação 7.2: Exibição dos cinco naturais posteriores ao da entrada.

A solução pode ser melhorada, tornando todas as instruções de exibição iguais e potencialmente reduzindo erros de digitação do programador, permitindo replicação das instruções apenas com o "copiar e colar". Veja a alteração na Codificação 7.3, em que incrementamos o valor da variável `x` antes de cada exibição.

```
x = int(input('valor: '))

x += 1
print(x)
x += 1
print(x)
x += 1
print(x)
x += 1
print(x)
x += 1
print(x)
```

Codificação 7.3: Alternativa para exibição dos cinco naturais posteriores ao da entrada.

Mesmo que a Codificação 7.3 tenha mais linhas de código do que a Codificação 7.2, a mudança gerou facilidade para o programador, que pode simplesmente replicar o incremento da variável `x` e sua exibição, sem nenhuma alteração. Com isso, caso o enunciado solicitasse os cinquenta naturais posteriores à entrada, ao invés de cinco,

bastaria copiar e colar essas duas instruções mais quarenta e cinco vezes. É trabalhoso? Sim! Mas menos trabalhoso e menos suscetível a erros do que a versão anterior.

Você deve ter observado que tanto a solução para o primeiro problema quanto para o segundo são possíveis de serem implementadas com o conhecimento que temos até o momento, porém são trabalhosas e, eventualmente, tornam-se inviáveis quando a quantidade de repetições é muito grande. Porém, existem problemas que simplesmente são impossíveis de serem resolvidos apenas copiando e colando instruções.

Um exemplo no qual existe esse impedimento é a terceira variação do problema para exibição de números naturais: suponha que o programa deverá exibir uma sequência de números naturais delimitada pelos valores *início* e *fim*, que são dois naturais dados pelo usuário como entrada. Note que o programa deverá exibir a sequência completa, incluindo os extremos *início* e *fim*. Por exemplo, caso o usuário insira o valor `5` como *início* e `12` como *fim*, a saída será `5 6 7 8 9 10 11 12`.

Usando o que sabemos até agora, qual a solução para essa terceira variação do problema? Impossível construí-la! Não há como prever a quantidade de valores exibidos, afinal isso dependerá dos valores de entrada dados pelo usuário, algo que só será descoberto quando o programa estiver em execução. Então, precisamos ampliar nossos recursos de programação para resolver esse problema, precisamos dos laços!

7.3. Tipos de estruturas de repetição

Essencialmente as estruturas de repetição podem ser classificadas em dois tipos:

1) *Estruturas de repetição com quantidade de repetições indefinida:* não é possível determinar quantas vezes as instruções do *loop* serão executadas antes dele ser iniciado;
2) *Estruturas de repetição com quantidade de repetições definida:* é possível determinar quantas vezes as instruções do *loop* serão executadas antes mesmo dele ser iniciado.

As linguagens de programação podem ter mais de um *loop* de cada tipo, permitindo ao programador decidir qual o mais adequado de acordo com o contexto, porém a razão de existência de todos os laços é a mesma: executar uma sequência de instruções repetidas vezes. Em Python temos um único laço de cada tipo: `while` e `for`.

7.4. Estrutura de repetição *while*

Inicialmente abordaremos somente o representante da estrutura de repetição com quantidade de repetições **indefinida**, que em Python é representada pelo comando `while`. A estrutura do comando `while` é similar à do `if`, conforme a Codificação 7.4.

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

Codificação 7.4: Estrutura do laço de repetição `while`.

As mesmas regras que aprendemos para as estruturas de seleção se aplicam ao *loop* `while`, portanto toda instrução que estiver indentada no `<bloco de código>` pertence ao *loop* e só será executada caso `<condição>` resulte em `True`, caso resulte em `False` o laço é encerrado e a instrução imediatamente seguinte ao seu bloco será a próxima a ser executada. Aparentemente, um fluxo de execução semelhante ao do `if`.

Porém, o `while` possui uma notável diferença em relação ao `if`. No `while`, após a execução da última instrução de seu bloco de código, o fluxo de execução é deslocado novamente para sua condição, que será avaliada mais uma vez, repetindo todo o procedimento explicado no parágrafo anterior. Podemos visualizar o funcionamento da estrutura de repetição `while` no fluxograma ilustrado Figura 7.2.

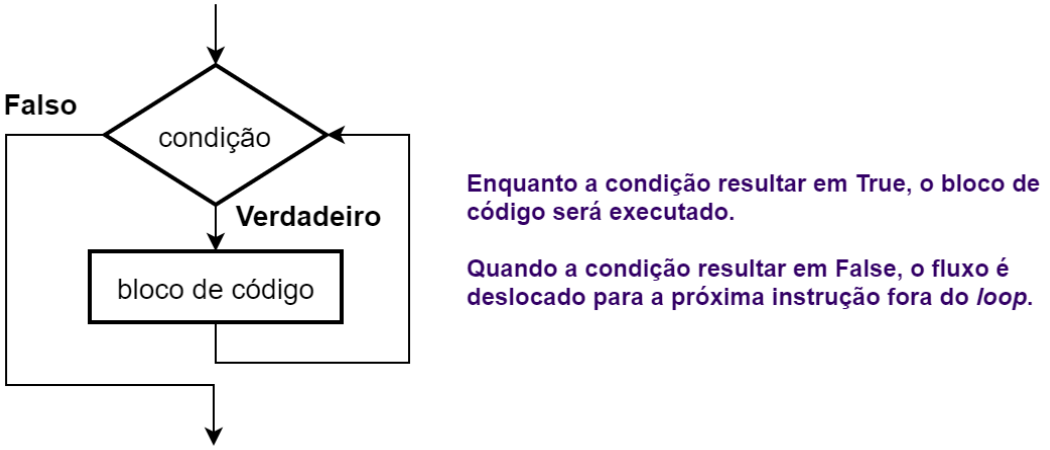


Figura 7.2: Representação em fluxograma da estrutura de repetição `while`.
Fonte: Elaborado pelo autor.

Para exemplificar o funcionamento do `while`, reescreveremos com um laço a solução para a primeira variação do problema de números naturais (Codificação 7.1). Para ampliar o entendimento, acrescentaremos a exibição da frase “tchau!” como uma instrução imediatamente posterior ao fim do *loop*, conforme Codificação 7.5.

```
x = 1
while x <= 5:
    print(x)
    x += 1
print('tchau!') # instrução imediatamente posterior ao laço.
```

Codificação 7.5: Exibição dos cinco primeiros números naturais positivos.

O bloco de código do `while` será executado exatamente cinco vezes, pois após a quinta execução de `x += 1` a variável `x` estará com o valor `6`, fazendo `x <= 5` resultar em `False`. No entanto, é importante observar que a condição do laço será sempre avaliada uma vez mais, neste caso exatamente seis vezes, pois apenas na sexta avaliação que a condição resultará em `False` e, consequentemente, encerrará o *loop*. A instrução que exibe "tchau!" é executada só uma vez, pois está fora do `while`.

A segunda variação do problema de números naturais, cuja primeira solução consta na Codificação 7.2 e uma versão melhorada na Codificação 7.3, seria facilmente solucionada com `while`, conforme Codificação 7.6.

```
x = int(input('valor: '))
exibidos = 0
while exibidos < 5:
    x += 1
    print(x)
    exibidos += 1
```

Codificação 7.6: Exibição dos cinco primeiros naturais posteriores ao da entrada.

Na Codificação 7.6 usamos a variável `exibidos` para verificar quantos naturais foram exibidos até o momento em que a condição é avaliada. Ela começa em zero, pois antes do laço, nenhum número foi exibido ainda, e é incrementada no final do laço. Veja que para exibir os cinquenta naturais posteriores ao da entrada, basta alterar a condição para `exibidos < 50`. Se o enunciado solicitasse os primeiros cinquenta mil números posteriores, bastaria alterar uma única instrução. Você entendeu o poder dos laços!

Por fim, você deve se recordar da terceira variação do problema de números naturais, em que dois valores (*início* e *fim*) são dados pelo usuário como entrada e que devemos exibir todos os valores do intervalo de naturais [*início*.,*fim*]. Agora, com o uso de um laço, é possível resolvê-lo facilmente, como consta na Codificação 7.7.

```
inicio = int(input('início: '))
fim = int(input('fim: '))
x = inicio
while x <= fim:
    print(x)
    x += 1
```

Codificação 7.7: Exibição de todos os naturais de *início* até *fim*, inclusive os extremos.

Observe que agora colocamos o incremento da variável `x` no final do bloco do laço, isso é comum, pois ela agora é a variável que controlará a execução do laço e uma forma de interpretar este laço é: (a) primeiro fazemos as operações necessárias com o `x`

atual (neste caso apenas a exibição); (b) depois de terminar, passamos para o próximo valor de `x` (neste caso um incremento de 1); (c) voltamos à condição para verificar se `x` ainda atende à condição; (d) caso atenda (condição resulta True) executamos mais uma vez o laço, repetindo o processo; (e) caso não atenda (condição resulta False), encerramos o laço e passamos para a próxima instrução fora do bloco do `while`.

VAMOS PRATICAR!

1) Teste no Python Tutor as soluções propostas para as três variações do problema de números naturais, tanto aquelas com *loop* `while` quanto aquelas sem *loop*. Certifique-se de ter entendido completamente o fluxo de execução do programa.

2) Crie um programa que exiba todos os números inteiros de 100 até 200 em ordem crescente. Depois crie outro programa que exiba de 200 até 100 em ordem decrescente.

3) Crie um programa que leia um número natural `n` dado pelo usuário e exiba só os `n` primeiros pares a partir do `0`. Por exemplo, se `n=6` será exibido `0 2 4 6 8 10`.

4) Crie um programa que solicite ao usuário dois números naturais `x` e `y`, o programa deverá exibir o quociente da divisão inteira de `x` por `y` sem usar os operadores de divisão e multiplicação. Por exemplo, se `x=7` e `y=2` a resposta será 3, pois podemos raciocinar que o quociente da divisão inteira de `x` por `y` é dado pela quantidade de vezes que `y` pode ser subtraído de `x` sem que `x` se torne negativo.

7.5. Variável contadora

Para construir laços de repetição muitas vezes utilizamos uma *variável contadora*, que é útil para *contar* quantas vezes o bloco de instruções do laço foi executado e/ou *controlar* quantas vezes ele será executado. A variável contadora é incrementada/decrementada por um *valor constante*, geralmente de um em um.

A Codificação 7.8 é um exemplo de programa que usa uma variável contadora apenas para contabilizar a quantidade de vezes que o bloco de instruções do *loop* foi executado. Note que a variável `contador` não influencia o número de repetições do `while`, isto é, ela não é usada como *variável de controle* deste laço.

```
executa = input('Executar o bloco do laço: ')
contador = 0
while executa == 'sim':
    contador += 1
    executa = input('Executar o bloco do laço de novo: ')
print(f'O bloco do laço foi executado {contador} vezes')
```

Codificação 7.8: Programa com variável contadora apenas para contagem de repetições.

A Codificação 7.9 é um exemplo de programa que usa uma variável contadora para controlar a quantidade de vezes que o bloco de instruções do laço será executado. Note que a variável `contador` influencia o número de repetições do `while`, isto é, agora ela é usada também como *variável de controle* deste laço.

```
contador = 10
while contador > 0:
    print(contador)
    contador -= 1
print('Fogo!')
```

Codificação 7.9: Programa em que a variável contadora controla o número de repetições.

VAMOS PRATICAR!

Crie um programa que peça letras como entrada, uma por vez, até que seja lida a letra `'x'`, ao final, o programa deve exibir a quantidade de letras lidas sem contabilizar `'x'`. Observação: lembre-se que o Python diferencia maiúsculas e minúsculas.

7.6. Variável acumuladora

Uma *variável acumuladora* é utilizada para acumular valores que, em geral, *não são constantes*, ou seja, seu incremento ou decremento pode ser variável.

Vejamos um exemplo de problema que precisa de uma variável acumuladora: *"crie um programa que receba como entrada os preços de itens comprados em um supermercado por um cliente, no final o programa deverá exibir o total da compra. Para informar que não há mais itens a serem comprados, o cliente digitará o valor `-1`"*. A Codificação 7.10 é uma solução válida para esse enunciado.

```
total = 0 # variável acumuladora

preco = float(input('preço do item: '))
while preco != -1:
    total += preco
    preco = float(input('preço do item: '))

print(f'Total da compra: R$ {total:.2f}')
```

Codificação 7.10: Programa que utiliza uma variável acumuladora.

Vejamos outro exemplo de problema com duas variáveis acumuladoras, porém com uma delas sendo decrementada: *"crie um programa que receba como entrada o crédito de um cliente e depois o preço de itens comprados por esse cliente, o programa*

deverá parar de solicitar novos preços quando o crédito disponível for insuficiente para pagar por um deles. Ao final, exiba o total da compra e o crédito restante". A Codificação 7.11 é uma solução válida para esse enunciado.

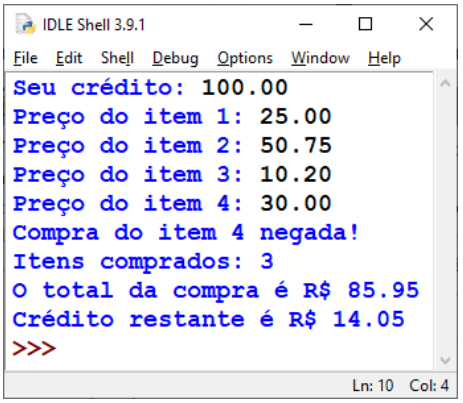
```
credito = float(input('Seu crédito: ')) # variável acumuladora
total = 0 # variável acumuladora

preco = float(input('Preço do item: '))
while credito >= preco:
    total += preco
    credito -= preco
    preco = float(input('Preço do item: '))

print(f'Total da compra: R$ {total:.2f}')
print(f'Crédito restante: R$ {credito:.2f}')
```

Codificação 7.11: Programa que utiliza duas variáveis acumuladoras.

Observe que a variável `credito`, além de acumuladora, também é usada como *variável de controle* do laço. Neste exemplo, temos duas variáveis usadas para controlar a execução do *loop*.

VAMOS PRATICAR!

Refaça a solução da Codificação 7.11 de modo que todo produto inserido pelo usuário seja numerado sequencialmente, iniciando em 1, e que o programa exiba a mensagem "*Compra do item X negada!*", após a leitura do item que extrapolar o valor do crédito, onde X é o número do item. Acrescente as instruções necessárias para, no final, exibir a quantidade de itens que puderam ser comprados. À direita há um exemplo de execução.



7.7. Variável *flag* booleana

As vezes criamos laços em que a condição é dada por uma variável com valor booleano. Nestes casos dizemos que a variável usada para controlar o laço é uma *flag* booleana, pois ela sinaliza se o laço deve ou não ser encerrado. Também é possível criar uma condição com mais de uma *flag* booleana. Veja um exemplo na Codificação 7.12.

```
total = 0
quero_comprar = True  # será usada como flag booleana no loop.

while quero_comprar:
    preco = float(input('Preço: '))
    total += preco
    opcao = input('Continuar comprando (s/n)? ')
    if opcao != 's':
        quero_comprar = False

print(f'Total da compra: R$ {total:.2f}')
```

Codificação 7.12: Programa com *loop* controlado por *flag* booleana.

Note que a clareza do nome dado à variável definida como *flag* booleana é muito importante, pois influenciará na forma como o programador entenderá o fluxo de execução. No exemplo da Codificação 7.12, podemos entender a leitura da primeira linha do *loop* como "*enquanto eu quiser comprar, execute o bloco de instruções*". Por isso, é evidente que quando a variável `quero_comprar` receber `False`, poderemos entender como "*não quero comprar*", indicando que o *loop* deve ser encerrado.

7.8. Laço infinito

Um laço infinito é uma estrutura de repetição que nunca é encerrada, ou que é encerrada apenas por um eventual erro no programa ou por um comando externo, como um pedido de finalização feito por meio do gerenciador de tarefas do sistema operacional. Geralmente laços infinitos são gerados por erros de lógica, como uma instrução esquecida pelo programador que faz com que a condição do laço jamais resulte em `False`. Portanto, se quisermos gerar um `while` infinito devemos garantir que sua condição sempre resulte em `True`.

Veja na Codificação 7.13, dois exemplos de laços infinitos, em que o fluxo de execução, uma vez no laço, jamais atingirá a instrução `print('acabou!')`.

```
n = 0
while n <= 10:
    print(n)

print('acabou!')
```

```
n = 0
while n >= 0:
    print(n)
    n += 2
print('acabou')
```

Codificação 7.13: Exemplos de programas com um laço infinito.

VAMOS PRATICAR!

Corrija ambos exemplos da Codificação 7.13 para que a execução seja encerrada, após exibir os números pares no intervalo fechado [0..10].

VOCÊ SABIA?

Laços infinitos podem ser úteis em programas que devem repetir continuamente uma sequência de instruções e, em certo ponto, reiniciar o procedimento, por exemplo, em um relógio. Veja mais em: https://en.wikipedia.org/wiki/Infinite_loop

VAMOS LER!

Existem alguns detalhes do *loop* while que são particulares da linguagem Python. Veja mais em: https://www.programiz.com/python-programming/while-loop.

7.9. Combinação de estruturas de controle de fluxo

Como visto na Codificação 7.12, é bastante comum combinarmos estruturas de repetição com estruturas de seleção. Em algoritmos mais complexos a combinação de estruturas de controle de fluxo ocorre frequentemente.

Um exemplo de combinação de estruturas de controle de seleção e repetição está no seguinte problema: *"Crie uma função que exiba apenas as 21 consoantes minúsculas do alfabeto latino."*. A Codificação 7.14 é uma solução válida para esse enunciado.

```
def consoantes():
    codigo_unicode = ord('a')
    while codigo_unicode <= ord('z'):
        letra = chr(codigo_unicode)
        if (letra != 'a' and letra != 'e' and letra != 'i' and
            letra != 'o' and letra != 'u'):
                print(letra)
        codigo_unicode += 1
```

Codificação 7.14: Função com combinação de laço e seleção.

Outro exemplo de combinação de estruturas de controle de seleção e repetição ocorre na solução para o seguinte enunciado: *"Crie uma função que receba como argumento um número natural `n` e devolva um valor booleano indicando se `n` é primo"*. Um número natural é considerado primo se possui exatamente dois divisores naturais, o número 1 e o próprio *n*. A Codificação 7.15 é uma solução válida para esse problema.

```
def primo(n):
    qtd_divisores = 0
    divisor = 1
    while divisor <= n:
        if n % divisor == 0:
            qtd_divisores += 1
        divisor += 1

    if qtd_divisores == 2:
        return True
    else:
        return False
```

Codificação 7.15: Função com combinação de laço e seleção.

Bibliografia e referências

DOWNEY, A. B. Iteração. *In*: **Pense em Python**. São Paulo: Editora Novatec, 2016. cap. 7. Disponível em: <https://penseallen.github.io/PensePython2e/07-iteracao.html>. Acesso em: 07 de fev. 2021.

PYTHON SOFTWARE FOUNDATION. **Instruções compostas**. 2021. Disponível em: <https://docs.python.org/pt-br/3/reference/compound_stmts.html#>. Acesso em: 08 fev. 2021.

STURTZ, J. Python "while" Loops (Indefinite Iteration). **Real Python**, 2020. Disponível em: <https://realpython.com/python-while-loop/>. Acesso em: 09 fev. 2021.