8

TEXTO BASE

LINGUAGEM DE PROGRAMAÇÃO

Faculdade IMPACTA

Texto base

8

Laços aninhados e interrupção de laços

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

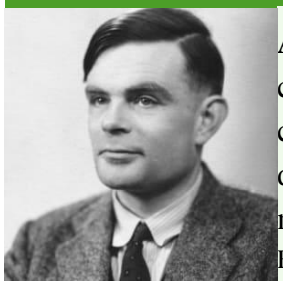
Resumo

Nesta aula os objetivos são: (I) aprender os comandos de encerramento e interrupção de laços; (II) simular a estrutura de repetição repeat…until usando while; (III) utilizar laços de repetição para validação de entradas; (IV) entender a necessidade das estruturas de repetição aninhadas; (V) resolver problemas com repetições aninhadas.

8.1. Motivação

Há situações em que um laço de repetição deve ser interrompido caso algo específico ocorra e Python possui comandos que podem nos ajudar nisso. Usando os comandos de interrupção de laços também poderemos simular um tipo de estrutura de repetição não embutida na linguagem Python, porém útil para a resolução elegante de alguns tipos de problemas! Com a simulação dessa nova estrutura de repetição poderemos realizar validações de dados de entrada, algo muito importante quando lidamos com interação com o usuário. Por fim, você já imaginou repetir uma repetição? Acredite, esse tipo de situação é muito comum na programação e vamos estudá-la!

VOCÊ CONHECE?



Allan Turing, nascido em 1912, foi um matemático, cientista da computação e criptoanalista responsável pela formalização de conceitos gerais de algoritmos e computação. Dentre seus feitos, desenvolveu técnicas para acelerar a quebra de codificações de mensagens alemãs interceptadas durante a 2ª Guerra Mundial.
Fonte da imagem: https://www.biography.com/scientist/alan-turing

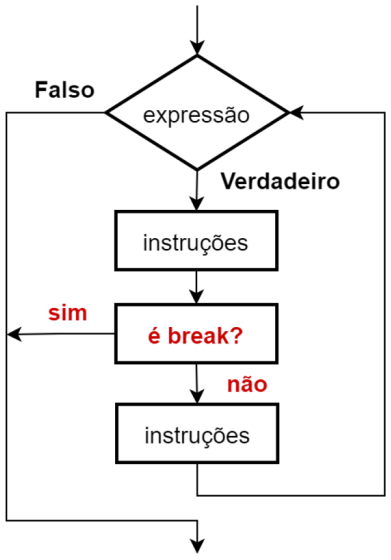
8.1. Introdução

Até agora aprendemos três formas básicas para controlar o fluxo de execução de um programa: (I) estrutura de controle sequencial; (II) estrutura de controle de seleção ou condicional e; (III) estrutura de controle de repetição.

Nesta aula nos aprofundaremos nas estruturas de repetição. Veremos como encerrar um *loop* por meio do comando `break` e como interromper apenas uma rodada com o comando `continue`. Com esses novos recursos, conseguimos facilmente simular uma estrutura de repetição que em outras linguagens de programação necessitam de comandos específicos, a *repeat…until*. O conceito desta estrutura pode ser aplicado, por exemplo, para a validação de dados de entrada, isto é, para verificar se os dados fornecidos como entrada pelo usuário são válidos de acordo com o requisitado pelo programa. Por fim, usaremos estruturas de repetição aninhadas, que são laços dentro do bloco de instruções de outros laços, algo recorrente em soluções para problemas mais complexos, em que um laço precisa ser repetido diversas vezes.

8.2. Comando *break*

A execução de um comando `break` *encerra* o *loop* mais interno em que está contido. O comando `break` geralmente está condicionado à um `if`, de modo que só seja executado quando algo específico ocorrer. A Figura 8.1 ilustra simplificadamente o deslocamento do fluxo de execução proporcionado por esse comando.



Caso o comando **break** seja executado o laço é encerrado e o fluxo de execução será direcionado para a próxima instrução fora dele.

Portanto, é comum que o comando **break** esteja dentro do bloco de instruções de uma estrutura de seleção, de modo que só seja executado se uma condição específica for satisfeita.

Figura 8.1: Representação simplificada do comando `break` em um fluxograma.
Fonte: Elaborado pelo autor.

Um problema simples para observarmos o funcionamento do comando `break` é este: *"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 um dos itens. Ao final, exiba o crédito restante". A Codificação 8.1 é uma solução válida para esse enunciado.

```
credito = float(input('Seu crédito: R$ '))
while credito > 0:
    item = float(input('Preço do item: R$ '))
    if item > credito:
        print('Compra negada! Ultrapassa seu crédito.')
        break
    credito -= item
print(f'Crédito restante: R$ {credito:.2f}')
```

Codificação 8.1: Exemplo de funcionamento do comando `break`.

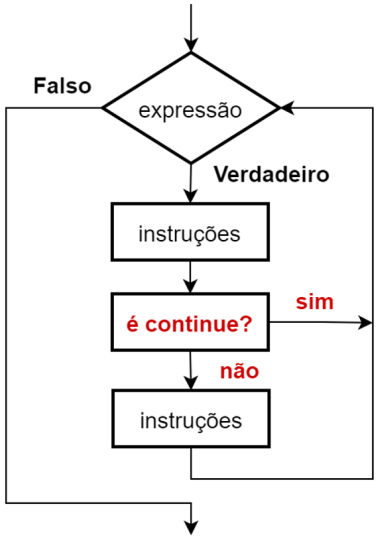
Outro exemplo de problema que pode ser beneficiado com o uso de `break` é o seguinte: *"Crie um programa que receba como entrada um número real `x`, um caractere `op` simbolizando um operador aritmético (`+`, `-`, `*` ou `/`) e outro real `y`. O programa exibirá o resultado da expressão `x op y`. O procedimento deve ser repetido enquanto o usuário desejar, porém deve ser encerrado antes de ocorrer um erro por causa de uma divisão por zero."*. A Codificação 8.2 é uma solução válida para esse enunciado.

```
calcular = input('Calcular (s/n)? ')
while calcular == 's':
    x = float(input('x: '))
    op = input('operador: ')
    y = float(input('y: '))
    if op == '+':
        resultado = x + y
    elif op == '-':
        resultado = x - y
    elif op == '*':
        resultado = x * y
    elif op == '/':
        if y == 0:
            print('Divisão por zero!\n')
            break
        else:
            resultado = x / y
    print(f'{x} {op} {y} = {resultado}\n')
    calcular = input('Calcular (s/n)? ')
print('Calculadora encerrada')
```

Codificação 8.2: Calculadora com prevenção de erro de divisão por zero.

8.3. Comando *continue*

A execução de um comando `continue` *interrompe a rodada atual* do *loop* mais interno em que está contido. O comando `continue` geralmente está condicionado à um `if`, de modo que só seja executado quando algo específico ocorrer. A Figura 8.2 ilustra simplificadamente o deslocamento do fluxo de execução gerado por esse comando.



Caso o comando **`continue`** seja executado a rodada atual do laço é interrompida e o fluxo de execução será direcionado para a condição do *loop*.

Portanto, é comum que o comando **`continue`** esteja dentro do bloco de instruções de uma estrutura de seleção, de modo que só seja executado se uma condição específica for satisfeita.

Figura 8.2: Representação simplificada do comando `continue` em um fluxograma. Fonte: Elaborado pelo autor.

Observaremos o funcionamento do comando `continue` na Codificação 8.3 que resolve o seguinte problema: *"crie um programa que receba como entrada o crédito de um cliente e os preços de itens comprados por ele. Se o preço de um item for superior ao valor do crédito, negue a compra do item, exiba o crédito restante e continue a compra. O programa deve parar de solicitar novos itens quando o crédito for zerado"*.

```
credito = float(input('Seu crédito: R$ '))
while credito > 0:
    item = float(input('Preço do item: R$ '))
    if item > credito:
        print(f'Compra negada! Restam: R$ {credito:.2f}')
        continue
    credito -= item
```

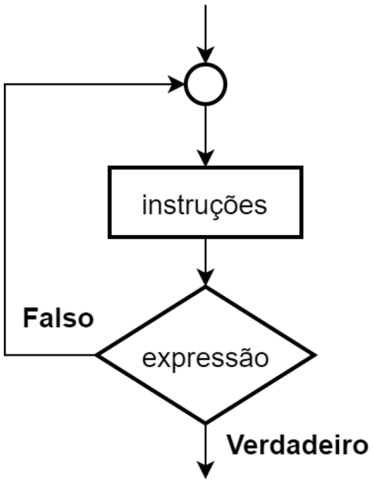
Codificação 8.3: Exemplo de funcionamento do comando `continue`.

No exemplo da calculadora simplificada, feito na Codificação 8.2, ao invés de encerrar o programa ao identificar uma divisão por zero, seria mais interessante informar o erro e reiniciar a calculadora. Uma forma simples de obter este resultado é simplesmente trocar o comando `break` por `continue`. Faça a modificação e teste!

Em geral, principalmente em programas mais simples, `break` e `continue` podem ser substituídos por uma adequação da condição do laço e por uma melhor organização de seu bloco de código. Portanto, evite o uso generalizado destes recursos, pois podem tornar o código mais confuso, uma vez que criam desvios do fluxo de execução que podem não ser óbvios à primeira vista, consequentemente dificultando a leitura e manutenção do código. Existem, no entanto, situações pertinentes para uso desses comandos, como a evidenciada em nosso próximo tópico.

8.4. Simulação da estrutura de repetição *repeat…until*

Python tem duas estruturas de repetição, `while` e `for` (ainda não abordamos). Porém, há problemas que são solucionados de forma mais elegante com uma terceira estrutura de repetição, conhecida como *repeat…until* ou, em português, *repita…até que*. Essa estrutura obriga que o bloco de instruções do laço seja executado pelo menos uma vez, pois, diferente do `while`, a condição para repetição está no fim do laço e não no início. Logo, para que o fluxo de execução atinja a condição deste laço, é necessário que primeiramente passe por seu bloco de código. A Figura 8.3 ilustra esse funcionamento.



O laço **`repeat…until`** possui uma demarcação inicial, para indicar seu início, seguida por um bloco de instruções e, por fim, sua expressão de decisão, ou seja, a condição do *loop*, que também indica seu fim.

Após a primeira execução do bloco de instruções, a condição é avaliada, caso resulte em **falso** o fluxo de execução é direcionado para a demarcação inicial, caso resulte em **verdadeiro**, o laço é encerrado.

Figura 8.3: Representação do laço *repeat…until*. Fonte: Elaborado pelo autor.

O funcionamento do *repeat…until* justifica o nome da estrutura, indicando que o laço *"repetirá as instruções até que sua condição resulte em verdadeiro"*. Como já mencionado, o Python não possui esse laço, mas podemos simulá-lo facilmente com a combinação dos comandos `while`, `if` e `break`, conforme a Codificação 8.4.

```
while True:
    <bloco de instruções>
    if <condição>: break
```

Codificação 8.4: Modelo de estrutura simulando o laço *repeat…until*.

Observe que na Codificação 8.4 definimos a condição do `while` como `True`, logo sempre será verdadeira. A princípio, poderíamos imaginar que isso tornaria o *loop* infinito, porém, no final do bloco de instruções, temos um `if` com uma condição que se resultar em verdadeiro executará o comando `break` e, como consequência, encerrará o laço. Veja na Codificação 8.5 um exemplo de solução com essa estrutura para o problema a seguir: *crie uma função que receba como argumento um número natural e exiba seus dígitos na ordem do último para o primeiro.*

```
def invertido(n):
    while True:
        print(n % 10, end='')
        n = n // 10
        if n == 0: break
```

Codificação 8.5: Exemplo de aplicação do laço *repeat…until*.

VAMOS PRATICAR!

Refaça a solução da Codificação 8.5, porém usando a estrutura de repetição `while` tradicional, sem usar `if` e `break`. Observe no Python Tutor o fluxo de execução de ambas as soluções. Por fim, discuta com seus colegas sobre a legibilidade de ambas versões e quais são as possíveis vantagens e desvantagens de cada abordagem.

8.5. Validação de dados de entrada

Quando os programas solicitam entradas ao usuário, é comum que ocorram inconsistências entre os dados esperados e aquilo que é inserido, erros cometidos pelo usuário propositalmente ou não. Quando é fornecido um dado inesperado, o programa pode falhar, encerrando inesperadamente ou produzindo resultado inconsistente. Para minimizar esse tipo de ocorrência, é útil realizar *validações de dados de entrada*.

Geralmente, validações de entrada são feitas com *loops*, solicitando um valor para o usuário repetidamente, até que seja inserido algum válido de acordo com o esperado pelo algoritmo que é estipulado pelo programador. Existem várias abordagens, uma das mais simples é com a estrutura *repeat…until*. Veja na Codificação 8.6 a solução para o seguinte problema: *"crie um programa que receba como entrada um número natural `n` no intervalo `[0..100]`, e exiba a soma dos `n` primeiros naturais positivos. A soma deverá ser feita por uma função, que a devolverá como resposta. Note que o programa deverá validar a entrada e só chamará a função quando `n` for válido"*.

```
def somatorio(n):
    soma = 0
    natural = 1
    while natural <= n:
        soma += natural
        natural += 1
    return soma

while True:
    n = int(input('Natural entre 0 e 100: '))
    if 0 <= n <= 100: break

print(f'A soma dos {n} primeiros naturais é {somatorio(n)}.')
```

Codificação 8.6: Programa com validação de dados de entrada usando *repeat…until*.

Note que na Codificação 8.6 a chamada à função `somatorio` só ocorrerá caso o fluxo de execução ultrapasse a validação de entrada feita com o laço `while` (usado para simular a estrutura *repeat…until*). Portanto, o programa impede a ocorrência de um erro na função `somatorio` caso seja inserido um valor negativo na entrada, pois se isso ocorrer, será solicitado outro valor de entrada. A Codificação 8.7 exibe uma versão alternativa, em que a estrutura de repetição *repeat…until* é trocada por um laço `while` convencional, note que apenas o trecho alterado está escrito.

```
...
n = int(input('Natural entre 0 e 100: '))

while n < 0 or n > 100:
    n = int(input('Número inválido, digite novamente: '))
...
```

Codificação 8.7: Programa com validação de dados de entrada usando `while` tradicional.

Uma vantagem desta abordagem é que podemos personalizar a mensagem exibida quando o usuário digita um valor incorreto, no entanto, temos a desvantagem de precisar inserir duas vezes a instrução que pede o dado ao usuário, algo que pode ser evitado quando a personalização não é necessária.

Na Codificação 8.7, a condição do `while` é complementar à Codificação 8.6, pois na estrutura *repeat..until* a estratégia é *"repita algo até que a condição seja atendida"*, e a condição é o número ser válido. Já na estrutura tradicional do `while`, a estratégia é *"repita enquanto a condição for atendida"*, logo o laço só deve ser repetido enquanto o número for **inválido**, que é condição complementar à de número válido.

Para obter a condição complementar à utilizada na Codificação 8.6, podemos simplesmente negá-la com o operador `not`, ou simplificá-la a negação, como a seguir:

`not (0 <= n <= 100)` ⇒ `not (0 <= n and n <= 100)`
⇒ `not(0 <= n) or not(n <= 100)` ⇒ `0 > n or n > 100`
⇒ `n < 0 or n > 100`

Todas as expressões do acima são equivalentes, então podemos escolher qualquer uma como condição para o laço de validação que o resultado será exatamente o mesmo. Consequentemente, podemos escolher aquela que fizer mais sentido para nós.

Por fim, a Codificação 8.8 exibe uma terceira versão em que é utilizada uma *flag* booleana. Isso pode ser útil em situações em que temos mais de uma condição que pode encerrar o laço ou caso seja necessário, em algum momento posterior, verificar quais restrições foram obedecidas pelo usuário, em especial se o programa precisar da validação de múltiplas entradas.

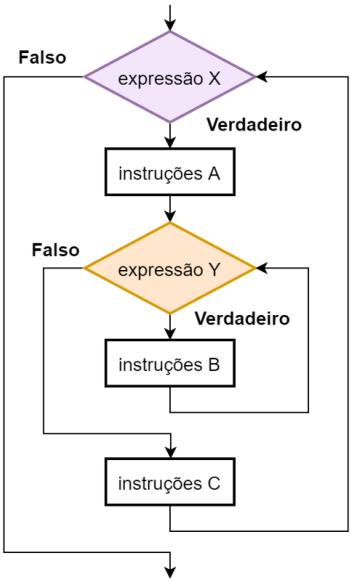
```
...
n_valido = False

while not n_valido:
    n = int(input('Natural entre 0 e 100: '))
    if 0 <= n <= 100:
        n_valido = True
...
```

Codificação 8.8: Programa com validação de dados de entrada usando *flag* booleana.

De início, atribui-se `False` a *flag* booleana para forçar que o fluxo de execução acesse o bloco de instruções do laço, que pode ser lido como "*repita enquanto a flag não for válida*". Uma vez no laço, recebe-se a entrada e verifica-se as condições da validação e, caso elas sejam atendidas, altera-se a *flag* para `True`. Observe que esta solução é semelhante à do laço *repeat…until*, pois a condição no `if` é a mesma: "*o número é válido?*". Porém, ao invés de `break`, alteramos o valor da *flag*, e o laço será interrompido na próxima verificação à condição do `while` (com `break`, o laço é interrompido instantaneamente).

8.6. Estruturas de repetição aninhadas

Há problemas complexos que exigem soluções contendo estruturas de repetição aninhadas, ou seja, um *loop* dentro do bloco de código de outro *loop*. Veja na Figura 8.4 um trecho de fluxograma em que há um laço aninhado e a correspondência em Python.



A cada rodada do *loop* mais externo (expressão X) o *loop* mais interno (expressão Y) executa todas suas rodadas. Dizemos que o *loop* mais interno está aninhado em relação ao mais externo.

```
...
while expressão X:
    instruções A
    while expressão Y:
        instruções B
    instruções C
...
```

Figura 8.4: Laço aninhado em fluxograma e em Python. Fonte: Elaborado pelo autor.

Para compreender a necessidade do uso de laços aninhados, começaremos por um problema bastante simples que evoluirá seu nível de complexidade à medida em que surgirem mais exigências no enunciado. Criaremos um relógio!

De início, a função que simulará um relógio é dada pelo enunciado: *"Crie uma função que exibe todos os segundos de um minuto. Lembre-se que o primeiro segundo é 0 e o último é 59."*. A Codificação 8.9 é uma solução válida para esse enunciado.

```
def relogio():
    s = 0
    while s < 60:
        print(s)
        s += 1
```

Codificação 8.9: Primeira versão da função que simula um relógio.

A Codificação 8.9 é válida, mas a forma como os segundos são exibidos ainda não se assemelha à de um relógio, então vamos para o próximo enunciado: *"Altere a função para que os segundos sejam exibidos no formato de um relógio digital, ou seja, hh:mm:ss. Use a função* `sleep` *da biblioteca* `time` *para aguardar um segundo entre as exibições"*. A Codificação 8.10 é uma solução válida para essa atualização do problema.

```
from time import sleep

def relogio():
    s = 0
    while s < 60:
        print(f'00:00:{s:02}')
        sleep(1)
        s += 1
```

Codificação 8.10: Segunda versão da função que simula um relógio.

Agora ampliaremos nosso relógio conforme o enunciado: *"Altere a função para que a cada 60 segundos o mostrador de minutos seja incrementado e o de segundos zerado. Repita o procedimento até completar 00:59:59"*. Precisaremos usar um *loop* aninhado, pois a cada um minuto executaremos o *loop* de segundos completamente, lembre-se que temos 60 minutos! A Codificação 8.11 uma solução válida[1].

```
def relogio():
    m = 0
    while m < 60:
        s = 0
        while s < 60:
            print(f'00:{m:02}:{s:02}')
            sleep(1)
            s += 1
        m += 1
```

Codificação 8.11: Terceira versão da função que simula um relógio.

Vamos ao próximo enunciado: *"Altere a função para que a cada 60 minutos o mostrador de horas seja incrementado e o de minutos zerado. Repita o procedimento até completar 23:59:59"*. Agora precisaremos de mais um *loop* para as horas, veja que com isso teremos três níveis de repetição! A Codificação 8.12 é uma solução válida.

E, em nossa última atualização da função, seguiremos o enunciado: *"Altere a função para que a cada 24 horas os três mostradores sejam zerados e o relógio reinicie a contagem."*. Novamente, precisaremos de mais um *loop* em que todos os anteriores estarão contidos, porém agora será um laço infinito para que o relógio nunca pare. A Codificação 8.13 é uma solução válida para esse novo enunciado.

[1] Dica: para testar o código, altere o tempo de espera na função `sleep` para um valor pequeno, como 1 centésimo ou 1 milésimo de segundo: `sleep(0.01)` ou `sleep(0.001)`.

```
def relogio():
    h = 0
    while h < 24:
        m = 0
        while m < 60:
            s = 0
            while s < 60:
                print(f'{h:02}:{m:02}:{s:02}')
                sleep(1)
                s += 1
            m += 1
        h += 1
```

Codificação 8.12: Quarta versão da função que simula um relógio.

```
def relogio():
    while True:
        h = 0
        while h < 24:
            m = 0
            while m < 60:
                s = 0
                while s < 60:
                    print(f'{h:02}:{m:02}:{s:02}')
                    sleep(1)
                    s += 1
                m += 1
            h += 1
```

Codificação 8.13: Quinta versão da função que simula um relógio.

Outro exemplo de aplicação de estruturas de repetição aninhadas é o problema a seguir[2]: "*crie uma função que receba como parâmetro um número natural* `n` *e desenhe um tabuleiro* `n x n`*, no qual as posições escuras sejam desenhadas com dois caracteres* `\u2588`*, dado pelo código Unicode 9608, e as posições claras com dois espaços. Em Python os índices começam em zero, então podemos descobrir quais quadrados devemos colorir com base na paridade da soma dos índices, isto é, quando a soma for par, temos um quadrado escuro, e quando a soma for ímpar, temos um quadrado claro*". A solução pode ser vista na Codificação 8.14, e a Figura 8.5 ilustra um exemplo de execução da função `tabuleiro` de 5 por 5 casas.

[2] Exemplo baseado em Pereira (2021).

```
def tabuleiro(n):
    linha = 0
    while linha < n:
        coluna = 0
        while coluna < n:
            if (linha+coluna) % 2 == 0:
                print(2 * chr(9608), end='')
            else:
                print(2 * ' ', end='')
            coluna += 1
        print()
        linha += 1
```

Codificação 8.14: Função que desenha um tabuleiro de tamanho personalizado.

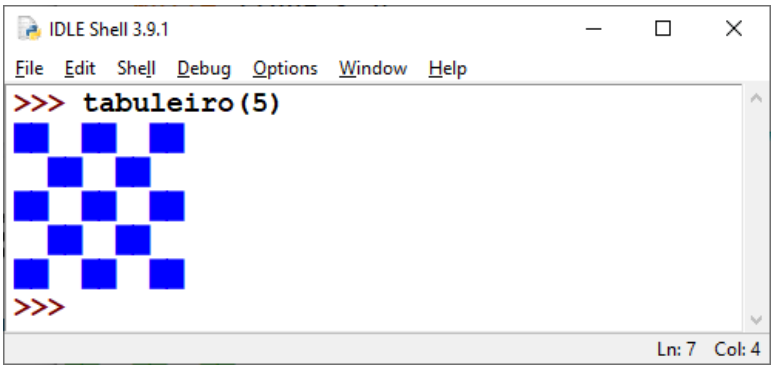


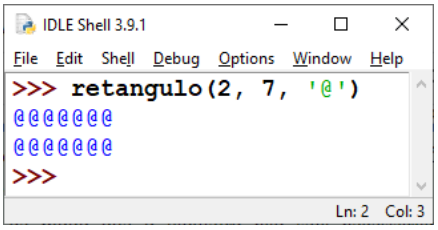
Figura 8.5: Exemplo de execução da função `tabuleiro`. Fonte: Elaborado pelo autor.

VAMOS PRATICAR!

1) Crie uma nova versão da função `relogio` em que o horário inicial seja 23:59:59 e a função exiba a contagem regressiva até 00:00:00.

2) Refaça a função tabuleiro de modo que o tabuleiro não seja necessariamente quadrado, ou seja, o número de linhas poderá ser diferente do número de colunas. Note que será necessário incluir mais um parâmetro na função.

3) Crie um programa que chame uma função com três parâmetros. O primeiro parâmetro é um natural `linhas`, o segundo é um natural `colunas` e o terceiro um caractere `s`. A função deve exibir um retângulo de tamanho `linhas` x `colunas` composto apenas por `s`, conforme exemplo à direita.



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.

PEREIRA, S. L. Fundamentos I. *In*: **Análise de Algoritmos**. 2021. Disponível em: <https://www.ime.usp.br/~slago/aa-02.ppsx>. Acesso em: 18 mar. 2021.

STURTZ, J. Python "while" Loops (Indefinite Iteration). **Real Python**, 2020. Disponível em: <https://realpython.com/python-while-loop/>. Acesso em: 12 fev. 2021.

TAGLIAFERRI, L. **How To Use Break, Continue, and Pass Statements when Working with Loops in Python 3**. 2017. Disponível em: <https://www.digitalocean.com/community/tutorials/how-to-use-break-continue-and-pass-statements-when-working-with-loops-in-python-3>. Acesso em: 14 fev. 2021.