Aula 4 - Condicionais e Loops

Karl Jan Clinckspoor

2 de julho de 2018

Sumário

1 Introdução

Condicionais e loops são os fundamentos de qualquer programa. São eles que permitem que decisões sejam tomadas durante a execução de problema, e permitem a mesma tarefa ser realizada várias vezes seguidas. Por isso, enriquecem vastamente a complexidade e utilidade de scripts.

2 Condicionais

2.1 if

if testa uma expressão *booleana*, ou seja, alguma coisa que pode ser avaliada em *True* ou *False*. Já vimos nas aulas anteriores expressões ou funções que retornavam esses valores. A sintaxe de *if* é um pouco mais complexa do que o já visto anteriormente.

```
In [1]: cond = True
        if cond:
            print('Oi')

Oi
```

Note que depois da condição, foi colocado ':', e o próximo comando foi *indentado*. Isso indica que o comando *print* está dentro do bloco de código do condicional. Por exemplo:

```
In [2]: cond = False
        if cond:
            print('Oi')
        print('Tchau')

        if not cond:
            print('Oi denovo')
        print('Tchau blumenau')

Tchau
Oi denovo
Tchau blumenau
```

Veja que o comando *print('Oi')* não foi executado, porém *print('Tchau')* foi, apesar de um estar logo em seguida do outro. Isso ocorreu devido à indentação. Note também a palavra **not**, que inverte um condicional. Como *cond* foi definido como sendo *False*, *not False* é True. Logo, o programa executou o comando.

É possível colocar condicionais de complexidade arbitrária. Porém, é bom você não se perder nas tramas dos condicionais.

2.2 Juntando condições - *and*, *or*, *not*

Como já visto, a palavra chave **not** afeta um condicional. Esses operadores são:

- **and**, retorna True somente se ambos os termos forem True
- **or**, retorna True se um dos dois termos for True
- **not**, inverte um condicional

Eles podem ser utilizados para testar mais de uma condição de uma vez.

```
In [3]: a = 5
        b = 7
        c = 12

        if (b < c) and (b > a):
            print('Sucesso')
```

```
Sucesso
```

É de boa prática agrupar as expressões entre parênteses para eliminar ambiguidade e problemas de interpretação tanto do autor quanto do interpretador.

2.3 elif, else

Além de testes singulares com if, podem ser feitos testes sequenciais. Somente é possível existir um *elif* ou *else* se houver um *if* precedente, e eles estão "ligados".

```
In [4]: b = 7

        if b < 5:
            print('Possibilidade 1')
        elif b > 5:
            print('Possibilidade 2')
        else:
            print('Ocorre somente se b == 5')
```

```
Possibilidade 2
```

Esses testes são necessariamente testados um a um. Assim que o primeiro teste for verdadeiro, o interpretador "sai"do bloco de código e continua. Veja a diferença entre utilizar dois if e um if, seguido de um elif.

```
In [6]: b = 7

        if b == 7:
            print('Possibilidade 1')
        elif b > 4:
            print('Possibilidade 2')
```

```
Possibilidade 1
```

```
In [7]: if b == 7:
            print('Possibilidade 1')
        if b > 4:
            print('Possibilidade 2')
```

```
Possibilidade 1
Possibilidade 2
```

2.4 Aninhamento

É possível também realizar um teste dentro de outro, aninhados. Os níveis de indentação se somam.

```
In [10]: a = 3
         b = 5

         if b > a:
             if a > 2:
                 print('Sucesso')
             else:
                 print('Falha...')
         else:
             print('Falha')
```

```
Sucesso
```

Note que este caso é quase equivalente a:

```
if (b > a) and (a > 2)
```

O detalhe é que há duas condições *else*, o que não seria possível no caso não aninhado. Porém, o caso não aninhado é muito mais fácil de se compreender, então leve isso sempre em conta.

2.5 Exercício

Construa um código que atribui um conceito a uma nota numérica, seguindo a seguinte regra draconiana:

```
nota > 90: 'A'
80 < nota <= 90: 'B'
70 < nota <= 80: 'C'
60 < nota <= 70: 'D'
50 < nota <= 60: 'E'
 0 < nota <= 50: 'F'
```

Caso deseje ver a resposta, execute a célula abaixo.

```
In [21]: # Digite seu código aqui
         nota = 90
         conceito = ''
```

```
In [ ]: %load ./respostas/Conceitos.py
```

2.6 Outras coisas interpretadas como sendo verdadeiras e falsas

Por simplicidade, o Python interpreta o seguinte como sendo False:

- List/dict/tuple vazios
- Strings vazias
- 0
- 0.0

```
In [29]: if '':
             print('Oi')
         if []:
             print('Oi')
         if 0:
             print('Oi')
         if 0.0:
             print('Oi')
         if tuple():
             print('Oi')
         if {}:
             print('Oi')
```

2.7 *in*

É possível testar também se algum membro está presente numa lista, utilizando a palavra chave **in**.

```
In [30]: a = [1, 2, 3, 4]
         if 3 in a:
             print('Está sim')
```

```
Está sim
```

3 Loops

Loops executam um bloco de código (lembre-se, um bloco é qualquer coisa indentada) repetidas vezes, sempre checando por uma condição de saída. Por vezes, a condição está implícita, por exemplo, se o final de uma lista foi atingido, não é necessário, ou possível, continuar, e por vezes é explícita.

Suponha o seguinte caso. Há uma lista de nomes de arquivos de experimentos, e você possui um código que consegue carregar e tratar os dados daquele experimento. Ao invés de ter que informar qual o experimento você quer tratar, você pode criar uma lista com os nomes dos experimentos, colocar um loop que fornece o nome, um a um, para a sua função, e depois ela trata os dados.

3.1 *while*

O tipo mais simples de loop é o **while** loop. A sintaxe é:

```
while {condição}:
    {código}
```

Veja este exemplo bastante simples.

```
In [5]: lst = []

        while len(lst) < 6:
            lst.append('l')
        print(lst)
```

```
['l', 'l', 'l', 'l', 'l', 'l']
```

Em cada iteração do loop, a condição *len(lst) < 6* é testada. Isso somente se torna falso quando o comprimento é igual a 6, pois começa em zero. Então, na quinta iteração, testa-se que 5 < 6 (True), mais um 'l' é adicionado à lista, e depois a condição é checada.

3.2 Loops infinitos

Uma nota de cuidado. Loops *while* necessariamente precisam de alguma maneira para sair. Caso isso não ocorra, acontece algo chamado de loop infinito. Para sair de um loop infinito, é necessário parar o interpretador, com um *Keyboard Interrupt*. Isso é feito apertando o botão de parar na barra de um Jupyter Notebook, ou apertando CTRL+C no console. Por exemplo, o seguinte loop é infinito.

```
while True:
    print('Dachshund')
```

3.3 *for*

De maneira oposta, os loops do tipo **for** são mais resistentes a ocorrerem infinitamente, pois necessitam de uma condição de parada explícita. Ainda assim, eles também podem correr infinitamente, então é bom sempre tomar cuidado, e saber como sair de um loop infinito. A sintaxe é a seguinte.

```
for {item} in {objeto}:
    {código}
```

O nome da variável é arbitrário. Pode ser *item*, *i*, *índice*, *arquivo*, *João*, é de sua escolha. Porém, é conveniente nomeá-la algo relacionado com o objeto a ser iterado.

Veja um exemplo.

```
In [19]: numeros = [1, 2, 3, 4, 5]
         quadrados = []
```

```
        for numero in numeros:
            quadrados.append(numero ** 2)

        print(numeros, quadrados)
```

```
[1, 2, 3, 4, 5] [1, 4, 9, 16, 25]
```

Esse loop vai de elemento em elemento de uma lista, eleva o elemento ao quadrado (mantendo a lista original intacta), e compara ambos. Não é necessário que a iteração ocorra somente numa lista. A única condição é que o objeto onde o *for* loop for ocorrer seja um iterável (iterable). Todas as estruturas de dados vistas na aula passada são iteráveis. Inclusive, strings são iteráveis. Veja o seguinte exemplo com um dicionário.

```
In [17]: frutas = {1: 'banana', 2:'maçã'}

         for key in frutas:
             print(key)
```

```
1
2
```

Lembrando que um dicionário associa um objeto a outro objeto, neste caso, um número a uma fruta (string), quando rodamos um loop pelo dicionário, estamos, na verdade, pegando somente as chaves (keys), não os itens associados. Para pegar os itens, é necessário usar o método de dicionário **.values()**.

```
In [15]: for value in d.values():
             print(value)
```

```
banana
maçã
```

Se você quiser tanto a chave quanto o valor, pode ser utilizado o método **.items()**.

```
In [18]: for item in d.items():
             print(item)
```

```
(1, 'banana')
(2, 'maçã')
```

3.4 Duas (ou mais) variáveis

Se você se lembra, é possível declarar duas variáveis de uma só vez separando-as por vírgulas. Veja que cada item do loop anterior, uma tuple, possui dois componentes. Então, é possível utilizar duas variáveis para o loop for.

```
In [20]: for key, value in d.items():
             print(key, value)

1 banana
2 maçã
```

Caso você tenha duas listas separadas e deseja realizar um loop simultamenamente por ambas, é possível utilizar o **zip**. Caso uma das listas seja mais comprida que a outra, o loop termina assim que a lista mais curta for "esgotada".

```
In [22]: bandeja1 = ['arroz', 'bife', 'alface']
         bandeja2 = ['feijão', 'cebola', 'tomate']

         for a, b in zip(bandeja1, bandeja2):
             print(f'{a} e {b}')

arroz e feijão
bife e cebola
alface e tomate
```

3.5 *enumerate*

Caso você deseje saber qual é o índice da iteração, isso é, quantas vezes o loop se repetiu, é possível utilizar **enumerate**, que retorna a posição e o item da iteraçao.

```
In [28]: alunos = ['José', 'João', 'Carlos']

         for i, aluno in enumerate(alunos):
             print(f'{i + 1}) {aluno}')

1) José
2) João
3) Carlos
```

3.6 *range*

O **range** é utilizado para controlar os loops utilizando uma faixa de números. Esses números não são gerados automaticamente, e sim a cada iteração. A sintaxe é até que semelhante com as sintaxe de seccionamento (lst[1:4:1], *slicing*). Veja o exemplo.

```
In [32]: for numero in range(11):
             if numero % 2:
                 print(f'{numero} é ímpar!')

1 é ímpar!
3 é ímpar!
```

```
5 é ímpar!
7 é ímpar!
9 é ímpar!
```

Da mesma maneira que um *slice*, o range **não** inclui o valor de parada. Há 3 maneiras básicas de se utilizar um range:

```
range(fim)
range(começo, fim)
range(começo, fim, passo)
```

Um range, com um len, podem ser utilizados para acessar os elementos de uma lista.

```
In [36]: componentes = ['Bário', 'Fosfato', 'Bicarbonato', 'Cloreto', 'Ferro']

         for i in range(len(componentes)):
             print(componentes[i])
```

```
Bário
Fosfato
Bicarbonato
Cloreto
Ferro
```

Como pode ser visto, esse método é claramente inferior ao método de um loop for sobre os elementos da lista, sem ter que fazer uma indexação. Há casos onde isso é estritamente necessário, mas são raros.

3.7 *break* e *continue*

Em alguns casos, é necessário impor condições que fazem um loop terminar precocemente, ou que um loop ignore parte, ou todo, o código e prossiga para o próximo item. Para isso, são utilizadas as palavras chave **break** e **continue**.

```
In [44]: numero = 1
         quadrados = []

         while True:
             quadrados.append(numero ** 2)
             if numero == 5:
                 break  # Caso isso não exista, você irá consumir toda sua memória criando número
             numero += 1  # Equivalente a escrever numero = numero + 1

         print(quadrados)
```

```
[1, 4, 9, 16, 25]
```

Veja como isso é equivalente ao seguinte loop:

```
In [43]: quadrados = []
         for num in range(1, 6):
             quadrados.append(num ** 2)
         print(quadrados)
```

```
[1, 4, 9, 16, 25]
```

Um continue é utilizado para pular uma etapa do loop. Por exemplo, se há algo indesejado naquele item de iteração e não se deseja fazer nada com ele. Suponha que tenhamos o início do trecho de um livro, e desejamos remover todas as palavras que contém a letra 'e':

```
In [49]: texto = ['Um', 'belo', 'dia', 'de', 'verão', 'João', 'comeu', 'feijão']  # Pontuação fo
         sm_sgunda_vogal = []

         for palavra in texto:
             if 'e' in palavra:
                 continue
             sm_sgunda_vogal.append(palavra)

         print(sm_sgunda_vogal)
```

```
['Um', 'dia', 'João']
```

Veja que sem o continue, todas as palavras seriam adicionadas.

3.8 Loops aninhados (*nested loops*)

Assim como é possível criar condicionais aninhados, é possível criar loops aninhados. Isso é relativamente comum, mas um tanto difícil de se entender no começo. Porém, sempre tenha em mente que o loop interno será terminado antes do próximo loop externo continuar. Suponha então o seguinte:

```
for i in range(5):
    for j in range(5):
        expressão
```

Quantas vezes a expressão será executada? Vamos testar.

```
In [63]: counter_ext = 0
         counter_int = 0

         for i in range(5):
             counter_ext += 1
             for j in range(5):
                 counter_int += 1

         print(f'Interno: {counter_int}\nExterno: {counter_ext}')
```

```
Interno: 25
Externo: 5
```

Veja que os loops internos e externos possuem o mesmo número de passos, pois sua sintaxe é igual. Porém, veja que o loop interno ocorreu 25 vezes, ao contrário do externo, que ocorreu 5 vezes. Isso é porque, a cada ciclo do loop externo, 5 ciclos do interno ocorreram. Então 5 * 5 = 25.

A nomenclatura de (i, j) utilizada aqui não foi coincidência. Lembre-se de matrizes. Uma maneira de se operar por uma matriz inteira é ir de linha em linha, e em cada linha operar em todas as colunas.

3.9 List comprehension

List comprehension é uma característica bastante marcante do Python. Ao invés de ter que gerar um loop for para gerar uma lista, é possível realizar tudo em uma única linha. A sintaxe é um pouco complicada de se entender de início, mas entender como uma *list comprehension* funciona é algo muito desejado. Veja como gerar uma lista de números e seus quadrados, como strings, em uma única linha. Depois, como realizar o exercício anterior mais sucintamente.

A sintaxe é:

```
\[{operação com iterador(es)} for {iterador(es)} in {iteráveis} [if {condição1}] [for {outro i
```

A sintaxe pode parer complexa, mas isso é porque há várias partes opcionais. Quanto mais partes opcionais foram colocadas, mais complicado vai ser entender a operação, e melhor seria a criação de um loop. Veja os exemplos a seguir, indo do mais simples para o mais complexo.

```
In [54]: numeros = [i for i in range(1,6)]
         quadrados_0 = [i**2 for i in range(1, 6)]
         quadrados_1 = [f'{i}:{i**2}' for i in range(1, 6)]  # Os iteradores podem ser utilizado
         print('Números:', numeros)
         print('Quadrados:', quadrados_0)
         print('Juntos:', quadrados_1)
```

```
Números: [1, 2, 3, 4, 5]
Quadrados: [1, 4, 9, 16, 25]
Juntos: ['1:1', '2:4', '3:9', '4:16', '5:25']
```

```
In [55]: texto = ['Um', 'belo', 'dia', 'de', 'verão', 'João', 'comeu', 'feijão']
         sm_sgunda_vogal = [i for i in texto if 'e' not in i]
         sm_sgunda_vogal
```

```
Out[55]: ['Um', 'dia', 'João']
```

```
In [64]: complexo1 = [i * j for i in range(1, 5) for j in range(5, 10)]  # Nested comprehension
         complexo2 = [i * j for i, j in zip(range(1,6), range(5,10))]    # Zip - não nested!
         print(complexo1)
         print(complexo2)
```

```
[5, 6, 7, 8, 9, 10, 12, 14, 16, 18, 15, 18, 21, 24, 27, 20, 24, 28, 32, 36]
[5, 12, 21, 32, 45]
```

4 Consolidação dos conceitos

Agora que um corpo suficientemente grande de conceitos foram aprendidos, podemos fazer um primeiro exercício realmente útil, que utiliza muito do que já foi visto. Abriremos um arquivo de dados, extrairemos o conteúdo desse arquivo, que possui duas colunas, em duas listas. Depois, calcularemos a média dos valores das duas colunas.

Essas tarefas são vastamente simplificadas pelos pacotes que serão aprendidos no futuro.

4.1 Etapa 1: Pensando no problema.

É sempre bom analisar concretamente o que será feito para depois abstrair os conceitos. Então, abra o arquivo 'Consolidação1.txt' no bloco de notas/notepad++/algum editor de texto. Veja que o arquivo consiste de duas colunas, nomeadas 'x' e 'y'. As colunas são separadas por um espaço.

Para transformar essa sequência de texto em duas listas de dados, seria necessário:

1. Ir de linha em linha
2. separar as duas colunas no espaço
3. Transformar cada valor em números (floats nesse caso)
4. Adicionar cada valor para sua lista correspondente.

4.2 Etapa 2: Abrindo o arquivo

A função **open** do Python cria algo chamado de *file handle*, ao invés de abrir o arquivo inteiro na memória. Um file handle é iterável, e retorna a cada iteração a linha atual. Porém, um file handle não pode ser indexado como uma lista. A sintaxe é:

```
open({nome do arquivo}, {leitura('r'), gravação('w'), adição('a')}, [codificação('utf-8')])
```

Não é necessário fornecer nada além do nome do arquivo. A função irá assumir que você está tentando abrir um arquivo no modo de leitura ('r'), e que a codificação do arquivo é 'utf-8'. Não se preocupe muito com codificação, pois a grande maioria dos arquivos hoje em dia são escritos em utf-8. Há algumas exceções, e nesses casos é necessário procurar qual é a codificação do arquivo para passar para a função. Eu, pessoalmente, só encontrei a codificação 'latin-1' em meu trabalho.

Não se esqueca sempre de fechar o file handle aberto! Isso é feito com **fhand.close()**

Veja como criar seu file handle:

```
In [70]: fhand = open('Consolidação1.txt', 'r')
```

Agora, vamos utilizar um loop para ir de linha em linha.

```
In [71]: for line in fhand:
             print(line, end='')
```

```
x y
0.0000 4.9332
0.3344 2.6304
...
99.3311 0.4917
99.6656 4.3795
100.0000 7.1438
```

Antes de realizarmos a separação de tudo, vamos treinar com uma linha aleatória. Depois colocaremos que soubermos que o código funciona, colocaremos no loop. Isso é possível porque todas as linhas são iguais. Em programação, sempre busque por padrões nos dados, pois esses padrões permitem a automatização.

4.3 Etapa 3: Teste com uma linha

Lembre-se que strings possuem métodos internos. Um deles é o método **split**, que retorna uma lista com os elementos separados por o caracter fornecido. Neste caso, desejamos separar no espaço, então usamos um *split(' ')*.

```
In [81]: teste = '0.0000 4.9332\n'  # O \n foi adicionado porque é isso que separa uma linha da
         temp_x, temp_y = teste.split(' ')
         temp_y

Out[81]: '4.9332\n'
```

Conseguimos separar as duas colunas. Porém, os dados estão como strings (note as aspas e a presença do
n), e devemos converter para floats. Isso é feito com a função **float**.

```
In [82]: temp_x = float(temp_x)
         temp_y = float(temp_y)
         temp_y

Out[82]: 4.9332
```

Note que a função de conversão foi inteligente o suficiente para remover o *newline*, '
n'. Senão, seria necessário removê-lo antes da conversão.

Agora devemos adicionar isso em duas listas, que serão chamadas de x e y.

```
In [85]: x = []
         y = []

         x.append(temp_x)
         y.append(temp_y)
         print(x, y)

[0.0] [4.9332]
```

Agora podemos colocar esse código no loop. Note, porém, que a primeira linha do arquivo não tem números, mas sim caracteres! Veja o que acontece se tentamos aplicar um float nos caracteres.

```
In [86]: temp_x = float('x')


         ---------------------------------------------------------------------------

         ValueError                                Traceback (most recent call last)
```

```
    <ipython-input-86-4e2e5fc4dfc2> in <module>()
----> 1 temp_x = float('x')


ValueError: could not convert string to float: 'x'
```

Então nosso loop iria falhar logo na primeira linha. Para isso, devemos ignorar uma linha caso ela comece com 'x'. Idealmente, se ela contiver um caracter, a linha deve ser ignorada. Pense numa maneira de deixar isso bastante geral. Neste caso, algo mais simples será utilizado.

Tente criar o loop completo para a criação das listas.

4.4 Criação do loop

```
In [ ]: fhand = open('Consolidação1.txt', 'r')
        x = []
        y = []

        for line in fhand:
            pass   # Substitua o pass pelo seu código. Essa palavra chave não faz nada.

        fhand.close()
        print('x:', x, '\ny:', y)
```

```
In [ ]: %load ./respostas/Consolidação1.py
```

Agora que temos os valores de x e y, podemos estudá-los um pouco mais. Vemos que x é uma lista de números que começa do zero e vai até 100. len(x) informaria quantos numeros há nessa lista. y, por sua vez, contém tantos números quanto x, mas não parecem ter um padrão.

Como calcularíamos a média das duas listas? Isso é feito somando-se os números das duas listas, e depois dividindo essa soma pelo número de pontos. Felizmente, o Python possui a função **sum** que faz justamente o que o nome indica, ela soma tudo num iterável. Além disso, vamos achar o menor e o maior valor da lista y pelas funções **min** e **max**.

Como exercício, implemente loops para contar o número de itens numa lista, somá-los, e encontrar o valor mínimo e o máximo. Uma solução está no arquivo 'Exercicio_max_min_len_sum.py'.

```
In [97]: media_x = sum(x) / len(x)
         media_y = sum(y) / len(y)
         min_y = min(y)
         max_y = max(y)

         print(f'A média de x é {media_x}')
         print(f'A média de y é {media_y}')
         print(f'O mínimo de y é {min_y}')
         print(f'O máximo de y é {max_y}')
```

```
A média de x é 49.999999999999986
A média de y é 5.091745666666669
O mínimo de y é 0.0097
O máximo de y é 9.9589
```

Assim, vemos que y possui números praticamente entre 0 e 10, cuja média é 5. A média esperada de x é 50, mas há um pequeno erro decimal devido à incapacidade de computadores de se representar floats com precisão total. De fato, a lista y (podemos também descrevê-la como um vetor) possui números aleatórios entre 0 e 10, logo a média de 5.

Nas próximas aulas essas duas colunas aprenderemos a carregar esse arquivo facilmente, e a plotar essas informações.