Aula 4 - Condicionais e Loops

Karl Jan Clinckspoor

15 de julho de 2018

Sumário

1 Revisão do conteúdo anterior

1. Funções.
    1. Há funções internas ao Python (print, int, str, list)

2. Há métodos de objetos, que podem ser chamados com a notação do ponto (`'string'.upper()`)
3. Podem requerer argumentos, separados por vírgula, e podem também receber argumenos opcionais de palavra chave (*keyword argument*), que tem valores padrão.
4. Retornam zero (`None`), um ou mais valores
5. Aninhamento de funções (chamar uma função dentro de outra) é possível.

2. Operações matemáticas básicas.

    1. Soma, subtração, multiplicação, divisão (`+`, `-`, `*`, `/`)
    2. Resto, divisão sem decimal, potênciação (`%`, `//`, `**`)
    3. Algumas podem ser utilizadas em outros objetos não numéricos, desde que definidas (concatenação de strings, listas)

3. Variáveis

    1. Tipos (strings, integers, floats, listas)
    2. Atribuição de valores e *unpacking* (a, b = 1, 2)
    3. Utilização de variáveis para deixar o código mais abstrato.
    4. Conversão de tipo (int->str, str->float, etc)
    5. Comparação de valores: retorna um booleano (True, False)
        1. `==`
        2. `!=`
        3. `\>`
        4. `<`
        5. `<=`
        6. `\>=`
        7. `a < b < c` também é válido e testa se b está entre a e c

4. Estruturas de dados

    1. Tipos: Listas, dicionários, tuplas
        1. Listas: sequências ordenadas de objetos arbitrários (strings, números, outras listas). Mutáveis.
        2. Dicionários: relações entre *chames* e *valores*. As chaves devem ser valores *hasháveis*, como strings, números, mas não listas, por exemplo. Idealmente, as chaves são imutáveis. Os dicionários, por sua vez, são mutáveis.
        3. Tuplas: sequências ordenadas de objetos arbitrários. Imutáveis.
    2. Declaração:
        1. Listas: itens entre colchetes, separados por vírgula. `[1, 2, 3, 4]`
        2. Dicionários: chave:valor, separados por vírgula, entre chaves. `{1:'a', 2:'b', 3:'c'}`
        3. Tuplas: items entre parênteses, separados por vírgula.
    3. Indexação:
        1. Listas e tuplas: índice entre colchetes após o nome da lista. `lista1[0]`.
            1. Índexação começa do zero. Então o primeiro elemento da lista1 é `lista1[0]`. O segundo, `lista1[1]`.
            2. Índices podem ser negativos, o que significa que a contagem começa da direita para a esquerda. Ou seja, o índice -1 indica o último item de uma lista.

2. Dicionários: chave entre colchetes após o nome. `dict1['chave']`.

4. *Slicing*:
    1. Obtenção de regiões de listas/tuplas utilizando uma notação 'início:fim(não incluso):passo'
    2. Os índices e os passos podem ser números positivos ou negativos.
    3. Por convenção, o passo é 1, o início é 0 e o final é a lista inteira. Então é possível abreviar os *slices*.
    4. Exemplos:
        1. `lista1[1:3]` pega os elementos do segundo ao terceiro, mas não o quarto (índice 3), de 1 a 1.
        2. `lista1[:3]` pega os elementos do primeiro ao terceiro, de 1 a 1.
        3. `lista1[::2]` pega todos os elementos, de 2 a 2.
        4. `lista1[-1::-2]` pega todos os elementos do último ao primeiro, indo de 2 a 2, no sentido contrário.
5. Métodos internos:
    1. Listas:
        1. `append`: adiciona o valor do argumento no final da lista, retorna `None`.
        2. `pop`: retorna o último elemento e remove-o.
        3. `reverse`: inverte a ordem de uma lista, retorna `None`.
        4. `sort`: organiza uma lista por um critério. Por padrão, é por número/alfabético. Retorna `None`.
    2. Dicionários:
        1. `keys`: retorna uma iterável (tipo uma lista) com as chaves do dicionário.
        2. `values`: retorna um iterável com os valores do dicionário
        3. `items`: retorna um iterável com uma tupla (chave, valor) de um dicionário.
6. Funções que podem ser aplicadas a estruturas de dados:
    1. `len`: informa o comprimento de uma lista (número de itens) ou o número de chaves num dicionário.
    2. `max`: informa o maior número numa lista ou chave do dicionário
    3. `min`: informa o menor número numa lista ou chave do dicionário

5. Strings
    1. Sequências imutáveis de caracteres. Podem ser transformados em listas.
    2. Podem ser concatenadas pelo operador +.
    3. *Escape sequences* são caracteres precedidos por um , que indicam um uso especial do caracter.
        1. `\n`: nova linha.
        2. `\t`: tab
        3. `\r`: retorna ao começo da linha
    4. *raw* strings: `r'string'`
        1. Ignora qualquer *escape sequence*.
    5. Métodos de formatação (junção de strings com outros valores)
        1. Padrão da linguagem C: %
        2. Método interno .format(itens)
            1. Substitui cada {} com seu valor correspondente nos argumentos.

3. `f-strings`: f'string {nome da variável}'
    1. Coloca pedaços de código dentro de chaves que são executados ao criar a string. Geralmente são colocados nomes de variáveis. Não é necessário conversão explícita de tipos.

6. Obtenção de ajuda:
    1. Função help.
    2. ?(nome da função): ?print
    3. Dentro dos parênteses na hora de chamar uma função, apertar Shift+Tab. 1x, 2x, 4x.

2 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. No final desta aula será apresentado um exercício que utiliza todos os conceitos utilizados até o presente momento.

3 Condicionais

3.1 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.

```
if condição (== True):
    ... (bloco de código indentado)
```

```
In [2]: cond = True
        if cond:
            print('Cond é True')
```

```
Cond é True
```

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. O bloco de código é composto por todas as linhas de nível igual de indentação após o `:`. Removendo ou adicionando identação quebra o bloco. Por exemplo, altere os valores de `entende` no código a seguir:

```
In [13]: print('D: Darmok and Jalad at Tanagra')

         entende = False
         if not entende:
             print("P: I don't understand! Who is Darmok?")
             print('D: Shaka, when the walls fell')
```

```
        print()
        print('P: Now I understand.')
        entende = True
        if entende:
            print('D: Sokath, his eyes uncovered')
            print('P: Temba, his arms wide')
```

```
D: Darmok and Jalad at Tanagra
P: I don't understand! Who is Darmok?
D: Shaka, when the walls fell

P: Now I understand.
D: Sokath, his eyes uncovered
P: Temba, his arms wide
```

Note também a palavra **not**, que inverte um condicional.

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

3.2 Juntando condições - *and, or, not, all, any*

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.
Há também as funções `all` e `any`.

- **all** retorna `True` se todos os valores forem `True`
- **any** retorna `True` se algum dos valores for `True`

```
In [16]: a = 5
         b = 7
         c = 12

         if (b < c) and (b > a):
             print('b é menor que c e maior que a')
         if all([True, True, True]):
             print('Tudo é verdadeiro')
         if any([True, True, False]):
             print('Pelo menos um é verdadeiro')
```

```
b é menor que c e maior que a
Tudo é verdadeiro
Pelo menos um é verdadeiro
```

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

3.3 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
```

No segundo caso, as duas condições foram testadas. Se há muitas condições que devem ser testadas sobre um termo só, que só pode obedecer uma dessas condições, é mais econômico utilizar vários `elif` do que vários `if`, pois no segundo caso todas as condições serão testadas, e no primeiro, somente até alguma delas for verdadeira.

3.4 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.

3.5 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
```

3.6 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 [17]: # Todos estes exemplos são falsos

         if '':
             print('Nada')
         if []:
             print('Nada')
         if 0:
             print('Nada')
         if 0.0:
             print('Nada')
         if tuple():
             print('Nada')
         if {}:
             print('Nada')
```

3.7 *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
```

4 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.

4.1 *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.

4.2 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('Para o infinito e além')
```

4.3 *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. A condição que o esse loop testa é se ainda há itens restantes no objeto.

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 [19]: 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 [20]: for value in frutas.values():
             print(value)

banana
maçã
```

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

```
In [21]: for item in frutas.items():
             print(item)

(1, 'banana')
(2, 'maçã')
```

4.4 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 [22]: for key, value in frutas.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
```

4.5 *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
```

É possível combinar, por exemplo, funções que retornam vários itens, desde que os valores sejam separados corretamente. É um tanto confuso entender esse tipo de execução. Compare a junção de `items`, que retorna tuplas (`chave`,`valor`), e enumerate, que retorna uma tupla (`índice`,`valor`):

```
In [27]: for i, (key, value) in enumerate(frutas.items()):
             print(i, key, value)
```

```
0 1 banana
1 2 maçã
```

```
In [28]: # Deixando a existência de duas tuplas mais evidente:
         for (i, (key, value)) in enumerate(frutas.items()):
             print(i, key, value)
```

```
0 1 banana
1 2 maçã
```

```
In [29]: # Mostrando o que acontece quando os valores não são arranjados corretamente
         for i, key, value in enumerate(frutas.items()):
             print(i, key, value)


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

         ValueError                                Traceback (most recent call last)

         <ipython-input-29-810d5880832d> in <module>()
           1 # Mostrando o que acontece quando os valores não são arranjados corretamente
     ----> 2 for i, key, value in enumerate(frutas.items()):
           3     print(i, key, value)


         ValueError: not enough values to unpack (expected 3, got 2)
```

Veja que 3 valores eram esperados `i`, `key`, `value`, mas a função `enumerate(frutas.items())` retornou somente dois, isto é, o índice `i` e uma tupla (`key`, `value`)

4.6 *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.

4.7 *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 núme
             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 [31]: texto = ['Um', 'belo', 'dia', 'de', 'verão', 'João', 'comeu', 'feijão']  # Pontuação fo
         texto_sem_e = []

         for palavra in texto:
```

```
            if 'e' in palavra:
                continue
            texto_sem_e.append(palavra)

        print(texto_sem_e)
```

```
['Um', 'dia', 'João']
```

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

4.8 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.

4.9 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]
```

5 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.

5.1 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.

5.2 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.

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
0.6689 0.2616
1.0033 9.6285
1.3378 9.8156
1.6722 3.4563
2.0067 4.2851
2.3411 2.9700
2.6756 2.6107
3.0100 6.9557
3.3445 0.4337
3.6789 3.1854
4.0134 9.9589
```

4.3478 5.7379
4.6823 1.1816
5.0167 9.2105
5.3512 7.4361
5.6856 5.3084
6.0201 3.9650
6.3545 5.4174
6.6890 2.5082
7.0234 9.0577
7.3579 7.6524
7.6923 4.4824
8.0268 8.7327
8.3612 3.7490
8.6957 6.5221
9.0301 0.9619
9.3645 0.2198
9.6990 2.8539
10.0334 7.0893
10.3679 9.4408
10.7023 2.6565
11.0368 9.7310
11.3712 9.8093
11.7057 5.8178
12.0401 6.7173
12.3746 8.7104
12.7090 4.7235
13.0435 6.7729
13.3779 3.3745
13.7124 9.7091
14.0468 7.8854
14.3813 8.1481
14.7157 3.7298
15.0502 4.6762
15.3846 4.9247
15.7191 9.3834
16.0535 7.6253
16.3880 8.6944
16.7224 1.1795
17.0569 4.5163
17.3913 2.9947
17.7258 4.7856
18.0602 9.8164
18.3946 1.4629
18.7291 2.1097
19.0635 4.1651
19.3980 4.3886
19.7324 8.1570
20.0669 4.6924

20.4013 6.0443
20.7358 8.1667
21.0702 1.1284
21.4047 0.8373
21.7391 7.2267
22.0736 1.5875
22.4080 6.8535
22.7425 1.5653
23.0769 0.2793
23.4114 6.2715
23.7458 9.0932
24.0803 0.6310
24.4147 0.0097
24.7492 6.0500
25.0836 9.6266
25.4181 1.9359
25.7525 2.8334
26.0870 9.2101
26.4214 3.9883
26.7559 3.7808
27.0903 7.1108
27.4247 7.2146
27.7592 1.3096
28.0936 3.3310
28.4281 2.6261
28.7625 3.7289
29.0970 3.7338
29.4314 7.3388
29.7659 2.7529
30.1003 6.1057
30.4348 1.3859
30.7692 3.6852
31.1037 0.4651
31.4381 8.5270
31.7726 2.9286
32.1070 9.0130
32.4415 9.2394
32.7759 7.1735
33.1104 2.9477
33.4448 4.5982
33.7793 6.8619
34.1137 5.0830
34.4482 6.4653
34.7826 1.6926
35.1171 8.9705
35.4515 2.2673
35.7860 6.8304
36.1204 0.2731

36.4548 5.0427
36.7893 8.5055
37.1237 8.6574
37.4582 9.3241
37.7926 4.8309
38.1271 4.0422
38.4615 6.7698
38.7960 0.9697
39.1304 8.5725
39.4649 5.5660
39.7993 3.2908
40.1338 4.2054
40.4682 2.1572
40.8027 9.3822
41.1371 9.3523
41.4716 1.9222
41.8060 6.4261
42.1405 1.7394
42.4749 8.0793
42.8094 5.6397
43.1438 4.9531
43.4783 7.3845
43.8127 7.3493
44.1472 7.7749
44.4816 2.4184
44.8161 1.2008
45.1505 9.1526
45.4849 2.2160
45.8194 2.8074
46.1538 4.0961
46.4883 9.3474
46.8227 5.4110
47.1572 2.1861
47.4916 1.0608
47.8261 8.6795
48.1605 0.8890
48.4950 2.3965
48.8294 4.7780
49.1639 6.3726
49.4983 4.7009
49.8328 0.0214
50.1672 2.7380
50.5017 6.5146
50.8361 7.7317
51.1706 5.0141
51.5050 5.2073
51.8395 2.4086
52.1739 5.6926

52.5084 8.8704
52.8428 4.1886
53.1773 0.3795
53.5117 8.1269
53.8462 7.7413
54.1806 0.0475
54.5151 0.7080
54.8495 3.8902
55.1839 7.0747
55.5184 7.9797
55.8528 5.7547
56.1873 8.4791
56.5217 9.0512
56.8562 1.8813
57.1906 3.7016
57.5251 0.4336
57.8595 0.0551
58.1940 9.1762
58.5284 4.3343
58.8629 1.2884
59.1973 5.2372
59.5318 9.4591
59.8662 3.6110
60.2007 8.6452
60.5351 7.0857
60.8696 7.3188
61.2040 7.5320
61.5385 9.2172
61.8729 1.1884
62.2074 6.6855
62.5418 3.0312
62.8763 2.3215
63.2107 8.0823
63.5452 9.7543
63.8796 9.9007
64.2140 9.4558
64.5485 3.1305
64.8829 0.0631
65.2174 0.8226
65.5518 0.0965
65.8863 1.9549
66.2207 4.4016
66.5552 5.3368
66.8896 9.2438
67.2241 9.8415
67.5585 9.2280
67.8930 3.8001
68.2274 2.7396

68.5619 6.0073
68.8963 1.5942
69.2308 4.2382
69.5652 6.0304
69.8997 1.0239
70.2341 6.9685
70.5686 9.2155
70.9030 6.1919
71.2375 6.0183
71.5719 1.2412
71.9064 4.6574
72.2408 7.1749
72.5753 5.3581
72.9097 5.3640
73.2441 2.3255
73.5786 6.9252
73.9130 4.4544
74.2475 5.9858
74.5819 7.9557
74.9164 3.8310
75.2508 2.2655
75.5853 5.1490
75.9197 1.6010
76.2542 3.0698
76.5886 6.3641
76.9231 2.5298
77.2575 4.9185
77.5920 6.8045
77.9264 3.0136
78.2609 0.0831
78.5953 6.0968
78.9298 6.4937
79.2642 7.0300
79.5987 6.1008
79.9331 8.8389
80.2676 4.1778
80.6020 2.0641
80.9365 9.3183
81.2709 8.3561
81.6054 3.2519
81.9398 5.9641
82.2742 1.8360
82.6087 4.8532
82.9431 0.8335
83.2776 0.7955
83.6120 6.6058
83.9465 5.7090
84.2809 7.9862

84.6154 9.5947
84.9498 3.5152
85.2843 8.6718
85.6187 9.9198
85.9532 7.0403
86.2876 3.2236
86.6221 0.9546
86.9565 7.7691
87.2910 7.9551
87.6254 9.2994
87.9599 1.1998
88.2943 9.0820
88.6288 5.0188
88.9632 1.0853
89.2977 2.1021
89.6321 5.0200
89.9666 2.5389
90.3010 9.5829
90.6355 4.6947
90.9699 5.0835
91.3043 6.0117
91.6388 1.1744
91.9732 4.7987
92.3077 3.2995
92.6421 2.8451
92.9766 6.4157
93.3110 8.7281
93.6455 5.2672
93.9799 2.7272
94.3144 3.1104
94.6488 9.7496
94.9833 2.4387
95.3177 7.8159
95.6522 2.7489
95.9866 5.5142
96.3211 2.5828
96.6555 2.9464
96.9900 8.8909
97.3244 5.7007
97.6589 9.0214
97.9933 3.9608
98.3278 1.0522
98.6622 6.1496
98.9967 2.4198
99.3311 0.4917
99.6656 4.3795
100.0000 7.1438

Se você tentar abrir o arquivo no bloco de notas, é possível que todos os valores estejam em uma única linha. Isso é devido à diferença de separador de linhas entre Windows e Linux, que utilizam \r\n e \n respectivamente.

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.

5.3 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), 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*, ". 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.

5.4 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.