



# LÓGICA DE PROGRAMAÇÃO E ALGORITMOS

AULA 6



Prof. Vinicius Pozzobon Borin



## CONVERSA INICIAL

Ao longo desta aula, você irá aprender a trabalhar com variáveis compostas em Python. Iremos aprender os três principais tipos existentes na linguagem: as tuplas, as listas e os dicionários.

Iremos investigar em linguagem Python as características e diferenças de cada uma, assim como criar e manipular todas elas. No último tema desta aula, iremos aprender mais alguns recursos de manipulação de *strings* para consolidar o seu conhecimento.

Assim anteriormente, todos os exemplos apresentados neste material poderão ser praticados concomitantemente em um *Jupyter Notebook*, como o *Google Colab*, e não requer a instalação de nenhum *software* de interpretação para a linguagem Python em sua máquina.

Ao longo do material, você encontrará alguns exercícios resolvidos. Esses exercícios estão colocados em linguagem Python.

## TEMAS 1 – TUPLAS

Vamos iniciar esta aula retomando um conceito apresentado anteriormente por meio de uma analogia que variáveis são como gavetas em uma estante, em que cada gaveta contém um nome. E dentro de cada gaveta iremos armazenar um único dado.

Porém, imagine que agora você quer armazenar itens para uma mesma finalidade nas gavetas. Para esta situação, seria interessante que todas as gavetas pudessem atender pelo mesmo nome (nome da variável). Se todas as gavetas terão o mesmo nome, como podemos identificar qual gaveta será aberta? A resposta é: por meio de uma numeração, como um índice. Deste modo, podemos dizer: “abra a gaveta 3”, ou “abra a gaveta 5”.

Vejamos um segundo exemplo lúdico. O nossa mascote, o Lenhadorzinho, estava com dificuldades de transportar seus itens ao sair em sua aventura. Ele só conseguia transportar um item por vez em suas mãos. Sendo assim, ele resolveu comprar uma mochila para expandir seu espaço. Agora, dentro do seu inventário, cabem quatro itens: ele decide carregar seu machado, uma camisa e comida (*bacon* e *abacate*). Portanto, os itens dentro de sua mochila são:



Figura 1 – Mochila do Lenhadorzinho



Créditos: deymos; Shanvood; Batshevs; Inspiring/Shutterstock.

Observe que antes da existência da mochila. Somente um item podia ser transportado por vez. Isso seria o equivalente a termos uma variável capaz de armazenar na memória um único dado por vez, chamamos de **variável simples**. A nossa mochila seria o que chamamos de **variável composta**. Neste tipo de variável, temos diversos espaços iguais na memória capazes de armazenar dados e todos eles atendem pelo mesmo nome. Neste caso, se quisermos acessar o machado, escrevemos `mochila[0]`. O *bacon*? Colocamos na `mochila[2]`.

A variável apresentada no exemplo é uma estrutura de dados. **Uma estrutura de dados é um conjunto de dados organizados e uma maneira específica na memória do programa. A maneira como os dados estão organizados na memória, como podem ser buscados, manipulados e acessados são o que definem e diferenciam as estruturas de dados.**

Deste modo, em linguagem Python, temos majoritariamente três tipos de variáveis que armazenam diversos valores: tuplas, listas e dicionários. Todas apresentam características organizacionais distintas e iremos investigar todas elas ao longo desta aula.

### 1.1 Construindo e manipulando tuplas

Vamos agora de fato à primeira estrutura. A tupla é o tipo de variável composta em Python mais simples. Ela tem como característica primária ser imutável. Nunca esqueça dessa informação, pois uma tupla, uma vez criada, não pode mais ser alterada ao longo da execução do programa. Repetindo, **a tupla é imutável**.

**A tupla é uma estrutura estática de dados**, isso significa que uma vez alocada na memória, não pode mais ter seu endereçamento alterado. Apesar de



conter suas particularidades, a tupla se assemelha um pouco às estruturas de vetor/array, para quem já conhece linguagens de programação como C ou Java. A representação de uma tupla em Python se dá da maneira a seguir. Colocamos os dados que desejamos cadastrar na tupla separados por vírgula e dentro de parênteses. Se mandarmos fazer o *print* da tupla, todos os dados serão impressos junto dos parênteses.

```
1 mochila = ('Machado', 'Camisa', 'Bacon', 'Abacate')
2 print(mochila)

('Machado', 'Camisa', 'Bacon', 'Abacate')
```

Podemos manipular e fatiar a tupla da mesma maneira que aprendemos com *strings*, anteriormente. A seguir, você encontra um resumo com algumas possibilidades de fatiamento e manipulação de índices em tuplas.

```
1 print(mochila[0]) #print do Elemento 1 - Índice 0
2 print(mochila[2]) #print do Elemento 3 - Índice 2
3 print(mochila[0:2]) #print dos Elementos 1 e 2 - Índice 0 e 1
4 print(mochila[2:]) # print dos elementos a partir do índice 2
5 print(mochila[-1]) #print do último

Machado
Bacon
('Machado', 'Camisa')
('Bacon', 'Abacate')
Abacate
```

Lembrando que, como uma tupla é imutável, isso significa que não podemos realizar atribuições com tuplas. Veja que, se tentarmos substituir o valor *bacon* por *ovos*, ocorre um erro: *tuple object does not support item assignment* (objeto tupla não suporta atribuição).

```
1 mochila[2] = 'Ovos'

-----
TypeError                                Traceback (most recent call last)
<ipython-input-21-10e5e4bdcd85> in <module>()
----> 1 mochila[2] = 'Ovos'

TypeError: 'tuple' object does not support item assignment

SEARCH STACK OVERFLOW
```

Podemos fazer uma varredura pelos elementos da tupla utilizando um laço de repetição. Como a tupla contém um tamanho finito, vamos trabalhar com o laço *for*. Para tuplas, o *for* é capaz de trabalhar mesmo sem a função *range*. Podemos somente fazer o laço a seguir que imprimiremos todos os elementos da tupla na tela.



```
1 for item in mochila:
2     print('Na minha mochila tem: {}'.format(item))
```

Na minha mochila tem: Machado  
Na minha mochila tem: Camisa  
Na minha mochila tem: Bacon  
Na minha mochila tem: Abacate

A variável *item* está funcionando como um iterador neste caso e irá passar por todos os elementos da tupla realizando um *print* de cada um. Caso prefira, podemos também criar a varredura na tupla da maneira convencional, com *range*, mas, para isso, precisaremos primeiro encontrar o tamanho da tupla usando a função *len* e, em seguida, fazer o laço para esse tamanho. Veja a seguir.

```
1 tam = len(mochila)
2 for i in range(0, tam, 1):
3     print('Na minha mochila tem: {}'.format(mochila[i]))
```

Na minha mochila tem: Machado  
Na minha mochila tem: Camisa  
Na minha mochila tem: Bacon  
Na minha mochila tem: Abacate

Nesta segunda possibilidade, precisamos explicitar o índice de cada item da *mochila*, caso contrário, o *print* não é realizado corretamente.

Agora, imagine que vamos fazer um aprimoramento na mochila do Lenhadorzinho, inserindo mais dois espaços para itens. Se nossa variável da mochila contém quatro espaços e é uma tupla, ela não pode ter seu tamanho alterado. Porém, podemos criar uma nova variável de tupla que irá conter a mochila antiga mais a atualização de tamanho. Também podemos juntar ambas tuplas por meio de concatenação, assim como já aprendemos com *strings*. Veja:

```
1 mochila = ('Machado', 'Camisa', 'Bacon', 'Abacate')
2 upgrade = ('Queijo', 'Canivete')
3 mochila_grande = mochila + upgrade
4
5 print(mochila)
6 print(upgrade)
7 print(mochila_grande)
```

('Machado', 'Camisa', 'Bacon', 'Abacate')  
('Queijo', 'Canivete')  
('Machado', 'Camisa', 'Bacon', 'Abacate', 'Queijo', 'Canivete')

Temos uma tupla agora de seis dados, ou seja, uma mochila com seis espaços. Observe que, como fizemos uma concatenação, estamos juntando



duas tuplas, isso significa que a ordem como fazemos a junção faz toda a diferença. Veja no exemplo a seguir.

```
1  mochila_grande_invertida = upgrade + mochila
2  print(mochila_grande)
3  print(mochila_grande_invertida)

('Machado', 'Camisa', 'Bacon', 'Abacate', 'Queijo', 'Canivete')
('Queijo', 'Canivete', 'Machado', 'Camisa', 'Bacon', 'Abacate')
```

## 1.2 Desempacotamento de parâmetros em funções

Podemos empregar tuplas para realizar um procedimento bastante poderoso em Python, o de desempacotar um parâmetro em uma função.

Suponha que você quer realizar o somatório de diversos valores, porém, você não sabe quantos valores serão somados. Pode ser que sejam somente 2, ou então 10, ou mesmo 100 números. Como criar uma função capaz de receber um número tão variável de parâmetros? Vejamos no exemplo a seguir o recurso do desempacotamento.

```
1  def soma(*num):
2      soma = 0
3      print('Tupla: {}'.format(num))
4      for i in num:
5          soma += i
6      return soma
7
8  #Programa principal
9  print('Resultado: {}'.format(soma(1,2)))
10 print('Resultado: {}'.format(soma(1,2,3,4,5,6,7,8,9)))

Tupla: (1, 2)
Resultado: 3

Tupla: (1, 2, 3, 4, 5, 6, 7, 8, 9)
Resultado: 45
```

No código apresentado, note que criamos uma função *soma* com um parâmetro. Porém, colocamos um **asterisco** antes do nome da variável. O símbolo de asterisco em Python indica que queremos desempacotar a variável *num*. Observe que no programa principal realizamos duas chamadas distintas da função *soma()*. Uma com dois valores e outra com nove valores. Todos esses valores serão armazenados em uma mesma variável dentro da função, e uma tupla será criada com os dados recebidos.

Note que dentro da função foi realizado um *print*, mostrando que os dados ali colocados são uma tupla (os parênteses na saída indicam que é uma tupla). Dentro da função, fazemos um laço da maneira como apresenta nesta aula,



andando por todos os elementos da tupla e somando-os, retornando o resultado final do somatório.

## 1.3 Exercícios

Vamos praticar um pouco o conceito de tuplas e o de desempacotamento de funções.

### 1.3.1 Exercício

Visite o *site* que contém o *ranking* das linguagens de programação mais amadas no ano de 2020: <<https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved>>.

Crie uma tupla com as 10 primeiras linguagens de programação mais amadas. Em seguida, faça:

- Apresente na tela o *ranking* numerado com uma palavra em cada linha.
- Apresente na tela somente o top 3 de linguagens.
- Apresenta na tela as últimas 5 do *ranking*.
- Em que posição está a linguagem Python? Mostre na tela.

## Python

```
1 #Exercício 1
2 linguagens = ('Rust','TypeScript','Python','Kotlin','Go',
3              'Julia','Dart','C#','Swift','JavaScript',)
4 print('Top 10 linguagens de programação mais amadas de 2020:')
5 print('(segundo o Stack Overflow)')
6 for i in range(0, len(linguagens), 1):
7     print(i+1, ' - ', linguagens[i])
8
9 print('\nTOP 3:',linguagens[:3])
10 print('Últimos 5:',linguagens[-5:])
11 i = 0
12 while (linguagens[i] != 'Python'):
13     i += 1
14 print('Encontramos Python na {} posição!'.format(i+1))
```

Top 10 linguagens de programação mais amadas de 2020:  
(segundo o Stack Overflow)

1 - Rust  
2 - TypeScript  
3 - Python  
4 - Kotlin  
5 - Go  
6 - Julia  
7 - Dart  
8 - C#  
9 - Swift  
10 - JavaScript

TOP 3: ('Rust', 'TypeScript', 'Python')  
Últimos 5: ('Julia', 'Dart', 'C#', 'Swift', 'JavaScript')  
Encontramos Python na 3 posição!



### 1.3.2 Exercício

Escreva uma função que contenha dois parâmetros. Essa função recebe como parâmetro uma *string* com uma mensagem a ser impressa na tela e outro parâmetro como sendo uma quantidade arbitrária de números empacotados. Dentro da função, encontre o maior entre todos os números recebidos e escreva na tela, dentro da função, a mensagem e o maior valor.

#### Python

```
1  #Exercício 2
2  def maior(msg, *num):
3      maior = 0
4      for i in range (0, len(num), 1):
5          if (num[i] > maior):
6              maior = num[i]
7          print(msg, maior)
8
9  #Programa principal
10 maior('Maior: ',8,6,4,78,56,12,9)
```

Maior: 78

## TEMA 2 – LISTAS

Uma tupla é imutável, certo? Porém, existe uma maneira de criarmos uma variável composta em Python que seja possível alterar seus dados, a tupla.

Exemplificando, imagine que o Lenhadorzinho em sua aventura comeu o *bacon* e encontrou no caminho uma laranja. Ele deseja substituir a laranja pelo espaço que estava o bacon na sua mochila. Podemos fazer isso com uma estrutura do tipo lista. Escrevemos uma lista utilizando colchetes. Observe, a seguir, a diferença de uma lista para tupla em Python.

```
1  mochila = ('Machado', 'Camisa', 'Bacon', 'Abacate')
2  print('Tupla: ', mochila)
3  mochila = ['Machado', 'Camisa', 'Bacon', 'Abacate']
4  print('Lista: ', mochila)
```

Tupla: ('Machado', 'Camisa', 'Bacon', 'Abacate')

Lista: ['Machado', 'Camisa', 'Bacon', 'Abacate']

Uma tupla sempre é escrita com parênteses, já a lista, sempre com colchetes. Agora, caso queira fazer a substituição de *bacon* por laranja, a lista permitirá (lembre-se que na tupla isso resultava em um erro). Veja.





```
1 mochila[2] = 'Laranja'
2 print('Lista: ', mochila)

Lista: ['Machado', 'Camisa', 'Laranja', 'Abacate']
```

## 2.1 Manipulando listas

As possibilidades de manipulação de listas em Python são bastante vastas. Podemos realizar todas as manipulações já aprendidas com tuplas, mas vamos além.

Começamos aprendendo a inserir um elemento no final da lista. Nossa mochila, atualmente, contém quatro dados (índices 0 até 3). Se tentarmos adicionar um quinto item, fazendo `mochila[4] = "Ovos"`, não irá funcionar, pois não destinamos um espaço de memória para um quinto dado. Não podemos adicionar elementos desta maneira na lista, então precisamos utilizar um método chamado de *append*, que irá se encarregar de alocar a memória no programa extra. Veja como utilizar a seguir.

```
1 mochila.append('Ovos')
2 print('Lista: ', mochila)

Lista: ['Machado', 'Camisa', 'Laranja', 'Abacate', 'Ovos']
```

Desta maneira, adicionamos mais um elemento no final da nossa lista, totalizando cinco. O *append* só funciona para inserção no final, mas é também possível inserir um dado em qualquer posição da lista, para isso, é necessário utilizar o método *insert*, passando como parâmetro o índice em que se deseja inserir. Assim, todos os elementos existentes serão deslocados para a direita e o novo elemento inserido no respectivo índice. Vamos inserir um *canivete* no índice de valor um. Veja que a *camisa* que estava no índice um, agora está no dois, e assim por diante.

```
1 mochila.insert(1, 'Canivete')
2 print('Lista: ', mochila)

Lista: ['Machado', 'Canivete', 'Camisa', 'Laranja', 'Abacate', 'Ovos']
```

Por fim, podemos também remover um elemento qualquer da nossa lista. Existem algumas maneiras distintas de fazer isso, vejamos duas delas. Na primeira, indicamos o índice (*del*) do dado a ser removido e na outra indicamos



o nome específico do dado (*remove*). O uso de cada uma dependerá da aplicação desejada.

```
1 del mochila[1]
2 print('Lista: ', mochila)
3 mochila.remove('Ovos')
4 print('Lista: ', mochila)

Lista: ['Machado', 'Camisa', 'Laranja', 'Abacate', 'Ovos']
Lista: ['Machado', 'Camisa', 'Laranja', 'Abacate']
```

### Saiba mais

**Você observou que nos exemplos anteriores nos referimos ao *append*, *insert*, *remove* etc., por método em vez de função? Por que essa alteração de nomenclatura agora?**

O que acontece é que uma lista em Python é, na verdade, um objeto de uma classe chamada de *list*, dentro da linguagem. Os conceitos de objetos e classe são outro paradigma de programação, que chamamos de programação orientada a objetos (POO), em que programamos classes e objetos, em vez de somente manipular funções.

É uma maneira distinta de se programar e que não será abordada nesta disciplina, pois é um assunto extenso. Se você estiver cursando um curso na área da computação, certamente terá uma outra disciplina somente voltada para POO.

Para esta nossa disciplina, cabe a você somente compreender como aplicar esses métodos, que para nós serão simples funções. No paradigma de POO, acessamos uma função/métodos invocando o nome da variável, seguida de um ponto e, após o ponto, o nome da função. Por exemplo, como vimos:

- *Mochila.append('Ovos')*
- *Variável.função(parâmetro)*

A quantidade de métodos disponíveis para manipularmos os objetos de listas e mesmo tuplas e dicionários é bastante grande e podem ser encontrados aqui: <<https://docs.python.org/pt-br/3/tutorial/datastructures.html>>. Acesso em: 17 fev. 2021.

Nesta disciplina, iremos trabalhar somente com alguns deles. Fique à vontade para investigar os outros, caso deseje tornar-se um programador avançado em Python.



## 2.2 Cópia de listas

A linguagem Python apresenta uma característica bastante interessante quando falamos de cópia de listas. Vamos observar o código a seguir.

```
1  #mesma referência
2  x = [5, 7, 9, 11]
3  y = x
4  print(x)
5  print(y)
```

[5, 7, 9, 11]  
[5, 7, 9, 11]

Iniciamos o código criando uma lista, na linha 2. Na linha 3, estamos fazendo com que uma nova variável *y* receba a mesma lista criada *x*. Como já era de se esperar, ao realizarmos o *print* de ambas, o resultado impresso é idêntico. Agora, vamos alterar o valor do índice zero da lista *y* e realizar o *print* de ambas na tela.

```
1  y[0] = 2
2  print(x)
3  print(y)
```

[2, 7, 9, 11]  
[2, 7, 9, 11]

Se alteramos somente a variável *y*, por que o *print* das listas *x* e *y* saíram iguais? Nós alteramos *x* também? Como?

O que acontece aqui é que toda a lista é um objeto, se fizemos algo como *y = x* em um objeto, somente estamos criando um segundo objeto com a mesma referência do primeiro e, portanto, caso alteremos um, estamos alterando o outro também (como se existisse uma conexão invisível entre eles). É como se estivéssemos dando um apelido para a variável *x*, que agora atende por *y* também.

Deste modo, existe uma maneira de realmente criar uma cópia independente da lista *x* e da lista *y*? Sim. Existe. Veja a seguir como fazemos isso.



```
1  #cópia
2  x = [5, 7, 9, 11]
3  y = x[:]
4  print(x)
5  print(y)
```

[5, 7, 9, 11]  
[5, 7, 9, 11]

Note a alteração na linha 3. Em vez de uma simples atribuição de uma variável na outra, fazemos `y = x[:]`. Isso garante que uma cópia de `x` é colocada em `y`. O uso dos dois pontos dentro dos colchetes garante a cópia integral. Caso queira copiar somente uma parte, você pode realizar um fatiamento de `x`. Assim, qualquer manipulação feita com uma variável não afetará a outra, conforme vemos no `print` a seguir, em que agora temos resultados diferentes.

```
1  y[0] = 2
2  print(x)
3  print(y)
```

[5, 7, 9, 11]  
[2, 7, 9, 11]

### TEMA 3 – STRINGS E LISTAS DENTRO DE LISTAS

Vimos até então como acessar um dado individual dentro de uma variável composta por meio de seu índice. Isso vale para tuplas e listas. Porém, existem dados que são compostos na sua essência, como uma *string*. Neste caso, podemos ir além e acessar os índices individuais da *string* dentro um único índice de uma lista.

Ficou confuso? Vamos esclarecer. Voltamos ao nosso exemplo da mochila. Se quisermos realizar o `print` de um dos dados da lista, basta fazer como o indicado a seguir.

```
1  mochila = ['Machado', 'Camisa', 'Bacon', 'Abacate']
2  print(mochila[0])
```

Machado

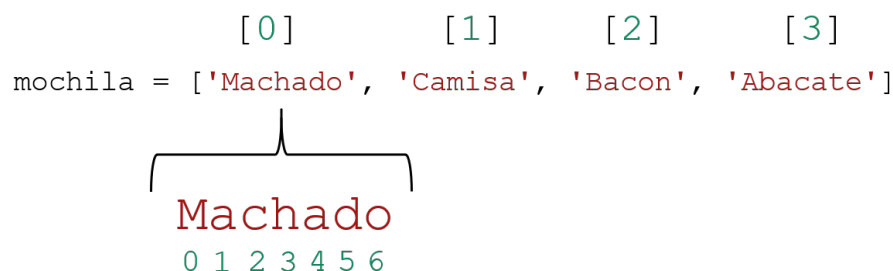
Nenhuma novidade até aí. Também já aprendemos que podemos fatiar uma *string* e acessar seus índices individualmente e aqui não será diferente. Neste caso, **temos uma dupla indexação. Ou seja, temos um índice referente a cada item da lista, e um segundo índice referente a cada caractere da**



**string**. Assim, podemos acessar não só cada dado dentro da lista, mas também cada caractere das *strings* de um índice da lista.

Na Figura 2, ilustramos a indexação dupla de listas com *strings*. Note que foram colocados os índices de cada item da lista acima deles. Porém, dentro de um item, temos uma *string* que também pode ser acessada individualmente. Por exemplo, no índice zero, temos o **Machado**. Ele contém sete caracteres, os quais podemos acessar individualmente.

Figura 2 – *Strings* com listas



Fonte: Borin, 2021.

A lógica por trás do acesso individual de cada caractere está no fato de que devemos abrir e fechar colchetes duas vezes, em vez de somente uma. A primeira vez representa o índice dentro da lista, e a segunda, o índice dentro da *string*. Se quisermos acessar o primeiro caractere de **Machado**, fazemos `mochila[0][0]`. Já `mochila[2][1]`, por exemplo, nos dá o terceiro item (*bacon*), segundo caractere (*a*).

```
1 mochila = ['Machado', 'Camisa', 'Bacon', 'Abacate']
2 print(mochila[0][0])
3 print(mochila[2][1])
```

M

a

Já aprendemos a realizar o *print* de uma variável composta empregando um laço de repetição (seção 1). Porém, existe uma maneira alternativa de realizar isso, que é passando por ambos os índices. Para fazermos uma varredura dupla de índices, colocamos dois laços de repetição aninhados. Veja a seguir.



```
1  mochila = ['Machado', 'Camisa', 'Bacon', 'Abacate']
2  for item in mochila:
3      for letra in item:
4          print(letra, end='')
5      print()

Machado
Camisa
Bacon
Abacate
```

O primeiro laço de repetição anda pelos itens da lista, já o segundo, ainda dentro do primeiro, faz a varredura caractere por caractere daquele item. Assim, só avançamos ao próximo item quando terminado o atual.

O *print* da linha 4 contém o parâmetro *end=""* que impede que o *print* pule de linha a cada caractere impresso. E o *print* da linha 5 está vazio propositalmente, servindo apenas para dar uma quebra de linha após uma palavra e outra.

Outra maneira de realizar essa varredura dupla, assim como aprendemos lá em tuplas, é com o código a seguir.

```
1  mochila = ['Machado', 'Camisa', 'Bacon', 'Abacate']
2  for i in range(0, len(mochila), 1):
3      for j in range(0, len(mochila[i]), 1):
4          print(mochila[i][j], end='')
5      print()

Machado
Camisa
Bacon
Abacate
```

Lembrando que a impressão da lista com dois laços resulta, neste caso, no mesmo resultado apresentado com um só laço de repetição que aprendemos lá nas tuplas e que valem também para listas. Apesar disso, esta maneira alternativa que você acabou de aprender não é desprovida de utilidade, podendo ter alguma aplicação dependendo do algoritmo a ser resolvido. Por exemplo, imagine que você precisa verificar caractere por caractere das palavras da lista procurando um determinado padrão. Essa varredura dupla poderá vir a ser bastante útil.

Agora, imagine uma seguinte situação em que você deve realizar o cadastro de uma lista de compras em um sistema. Cada produto comprado deverá ser registrado, com seu nome, quantidade e valor unitário.

Para resolver esse problema, talvez a solução mais simples que venha na sua mente seja a de criar três listas simples distintas, uma para cada tipo de



dado. Porém, o Python permite irmos além, pois podemos criar uma lista única em que dentro de cada índice da lista teremos outra, uma lista contendo os campos de nome, quantidade e valor. Este é mais um caso de indexação dupla.

Vejamos, a seguir, como implementar isso em Python. Vamos criar uma lista vazia e ir adicionando um produto por vez nela. O laço de repetição de cadastro permite somente três itens para fins de simplificação do exercício.

```
1 item = []
2 mercado = []
3
4 for i in range(3):
5     item.append(input('Digite o nome do item:'))
6     item.append(int(input('Digite a quantidade:')))
7     item.append(float(input('Digite o valor:')))
8     mercado.append(item[:])
9     item.clear()
10 print(mercado)
```

↳ Digite o nome do item:Cebola  
Digite a quantidade:2  
Digite o valor:0.99  
Digite o nome do item:Tomate  
Digite a quantidade:5  
Digite o valor:0.89  
Digite o nome do item:Saco de Arroz  
Digite a quantidade:1  
Digite o valor:5  
[['Cebola', 2, 0.99], ['Tomate', 5, 0.89], ['Saco de Arroz', 1, 5.0]]

Na linha 1 e na linha 2, criamos duas listas vazias. A lista *item* é uma variável temporária que servirá para inserir a lista dentro da lista *mercado*. Dentro do laço de repetição, povoamos com leituras de dados via teclado, a lista *item*, usando *append*. Colocamos nela o nome de um produto, sua quantidade e o valor unitário.

Em seguida, na linha 8, inserimos uma cópia dela dentro da lista *mercado* (não esqueça do [:]). Na linha 9, limpamos a lista *item* para que na próxima leitura de dados ela não contenha as informações que já inserimos no *loop* anterior. O *print* realizado nos mostra na tela que temos uma lista externa e, dentro dela, diversas pequenas listas separadas por vírgulas. Cada lista menor conterá uma trinca de valores heterogêneos: uma *string*, um *int* e um *float*.

Uma maneira alternativa de solucionarmos o mesmo problema é, em vez de criarmos duas listas, uma temporária menor e outra principal, criamos somente a principal e variáveis simples para cada campo da lista. Veja a seguir como ficaria esse código.



```
1 mercado = []
2
3 for i in range(3):
4     nome = input('Digite o nome do item:')
5     qtd = int(input('Digite a quantidade:'))
6     valor = float(input('Digite o valor:'))
7     mercado.append([nome, qtd, valor])
8     print(mercado)

Digite o nome do item:Cebola
Digite a quantidade:2
Digite o valor:0.99
Digite o nome do item:Tomate
Digite a quantidade:5
Digite o valor:0.89
Digite o nome do item:Saco de Arroz
Digite a quantidade:1
Digite o valor:5
[['Cebola', 2, 0.99], ['Tomate', 5, 0.89], ['Saco de Arroz', 1, 5.0]]
```

A saída resultante é idêntica à anterior. A diferença é que declaramos três variáveis *nome*, *qtd* e *valor* e inserimos ela dentro da lista *mercado* no formato de uma lista *[nome, qtd, valor]*.

Podemos acessar individualmente um dado somente da lista *mercado*? Certamente. Vejamos o que os exemplos a seguir retornam.

```
1 # Qual o nome do primeiro produto?
2 print(mercado[0][0])
3 # Quanto custa um tomate?
4 print(mercado[1][2])
5 # Quantos sacos de arroz foram comprados?
6 print(mercado[2][1])

Cebola
0.89
1
```

## Saiba mais

### Para pensar!

Podemos acessar somente um caractere dentro de uma lista com listas?

Por exemplo, como você acessaria a letra *C* da palavra *cebola*?

Dica: você terá uma *string* dentro de uma lista que está dentro de outra lista. Portanto, uma indexação tripla!

Finalizando o assunto de manipulação de listas de listas, podemos pegar a lista de compras do mercado e realizar um *print* dela na tela, informando em colunas cada um dos dados bem como o valor total do pedido. O resultado em código é apresentado a seguir.





```
1 soma = 0
2 print('Lista de compras:')
3 print('-' * 20)
4 print('item | quantidade | valor unitário | total do item')
5 for item in mercado:
6     print('{} | {} | {} | {}'.format(item[0], item[1], item[2], item[1] * item[2]))
7     soma += item[1] * item[2]
8 print('-' * 20)
9 print('Total a ser pago: {}'.format(soma))
```

Lista de compras:

item	quantidade	valor unitário	total do item
Cebola	2	0.99	1.98
Tomate	5	0.89	4.45
Saco de Arroz	1	5.0	5.0

Total a ser pago: 11.43

## 3.1 Exercícios

Vamos praticar um pouco o conceito de listas. Lembrando que manipulações vistas com tuplas continuam sendo válidas também.

### 3.1.1 Exercício

Escreva um algoritmo em Python que crie uma lista vazia e vá adicionando valores referentes às notas de um aluno nesta lista. Quando o usuário desejar parar de digitar notas (digitando um valor negativo, por exemplo), calcule a média das notas digitadas.

Além disso, faça um tratamento para a exceção de divisão por zero (*ZeroDivisionError*). Essa exceção poderá ocorrer caso o usuário não digite nenhuma nota válida.

## Python

```
1 #Exercício 1
2 try:
3     notas = []
4     x = float(input('Digite uma nota:'))
5     while x >= 0:
6         notas.append(x)
7         x = float(input('Digite uma nota:'))
8     soma = 0
9     for i in notas:
10         soma += i
11     media = soma/len(notas)
12     print(notas)
13     print('Média das notas digitadas: {}'.format(media))
14 except ZeroDivisionError:
15     print('Nenhum valor foi digitado para o cálculo. Encerrando...')
```



### 3.1.2 Exercício

O algoritmo mais simples de se buscar um dado em uma estrutura de dados é chamada de busca sequencial. A busca sequencial é uma varredura simples do primeiro ao último elemento da estrutura, verificando se o dado desejado se encontra presente.

Escreva uma função em Python que receba como parâmetro uma lista e um dado. Verifique se o dado está presente na lista e retorne da função o seu índice, caso ele esteja presente, caso contrário, retorne -1.

#### Python

```
1 #Exercício 2
2 def buscaSequencial(lista, dado):
3     x = 0
4     while x < len(lista):
5         if (lista[x] == dado):
6             return x
7         x += 1
8     return -1
9
10 #Programa principal
11 teste = [3,7,9,1,0,7,5,12]
12 #utilize o tipo de dado que quiser, basta alterar aqui
13 dado = int(input('Digite um valor inteiro: '))
14 res = buscaSequencial(teste, dado)
15 if res >= 0:
16     print('Posição onde o {} foi encontrado: {}'.format(dado, res + 1))
17 else:
18     print('Dado não localizado...')
```

### 3.1.3 Exercício

Escreva um algoritmo que leia o nome, altura e peso de pessoas e armazene as informações em uma lista. O programa deve ir cadastrando um número indeterminado de dados e armazenar dentro da lista também o IMC da pessoa. Ao final do programa, imprima a lista completa e também:

- O total de cadastros
- A pessoa com o maior IMC
- A pessoa com o menor IMC

O cálculo do IMC deve ser realizado empregando uma função *lambda* e é dado como:  $IMC = peso / (altura^2)$ .

Em que a massa é dada em quilograma e a altura em metros.

```

1  #Exercício 3
2  pessoas = []
3  imc = lambda peso, altura: peso / (altura * altura)
4
5  while True:
6      nome = input('Nome: ')
7      altura = float(input('Altura (m): '))
8      peso = int(input('Peso: '))
9      x = imc(peso, altura)
10     pessoas.append([nome, peso, altura, x])
11     res = input('Desejar fazer mais um cadastro? [S/N]')
12     if res in 'Nn':
13         break
14 print('Cadastros: ', pessoas)
15 print('Total de cadastros:', len(pessoas))
16 maior = 0
17 menor = 99
18 for i in range(0, len(pessoas), 1):
19     if pessoas[i][3] > maior:
20         maior = pessoas[i][3]
21     if pessoas[i][3] < menor:
22         menor = pessoas[i][3]
23 print('Maior IMC:', maior)
24 print('Menor IMC:', menor)

```

## TEMA 4 – DICIONÁRIOS

Vejamos agora a última estrutura de dados: o dicionário. É válido iniciar este tema lembrando que todas as manipulações que você aprendeu em tupla e em lista continuam valendo em dicionário. Iremos somente expandir o seu conhecimento com esta nova estrutura.

Nas tuplas, indicávamos elas por meio de parênteses. As listas, com colchetes. Já os dicionários serão criados com a abertura de chaves.

```

1  mochila = ('Machado', 'Camisa', 'Bacon', 'Abacate')
2  print('Tupla: ', mochila)
3  mochila = ['Machado', 'Camisa', 'Bacon', 'Abacate']
4  print('Lista: ', mochila)
5  mochila = {'Machado':1, 'Camisa':2, 'Bacon':3, 'Abacate':4}
6  print('Dicionário: ', mochila)

```

Tupla: ('Machado', 'Camisa', 'Bacon', 'Abacate')

Lista: ['Machado', 'Camisa', 'Bacon', 'Abacate']

Dicionário: {'Machado': 1, 'Camisa': 2, 'Bacon': 3, 'Abacate': 4}

Um dicionário contém no seu cerne uma estrutura de dados chamada de *hash*. Uma *hash* é um tipo de estrutura de dados que é capaz de inserir e buscar



dados utilizando chaves (também chamadas de palavras-chaves), e não os seus índices. A estrutura de dicionário no Python funciona de maneira semelhante ao *map/hashmap*, presente em linguagens como C++ e Java.

**Nos dicionários, chaves são utilizadas para referenciar os dados em vez de índices.** Deste modo, **as palavras-chaves dos dicionários irão funcionar como índices para acessarmos os dados.** Em vez de termos índices numéricos atrelados aos cadastros, podemos ter palavras que melhor atendam às necessidades da nossa aplicação. Vejamos um exemplo de criação de um dicionário em que temos três chaves: nome de um jogo de *videogame*, desenvolvedora do jogo e do ano de lançamento.

```
1 game = {'nome': 'Super Mario',  
2         'desenvolvedora': 'Nintendo',  
3         'ano': 1990}  
4 print(game)  
  
{'nome': 'Super Mario', 'desenvolvedora': 'Nintendo', 'ano': 1990}
```

No dicionário, cada dado inserido é composto de um par no formato *chave:dado*. Separamos a chave do seu respectivo valor por dois pontos. Portanto, em *'nome': 'Super Mario'*, a chave é a *string nome*. E seu respectivo valor é *Super Mario*. Assim como *desenvolvedora* e *ano* são chaves também. Quando realizamos o *print* do dicionário, estamos imprimindo sempre o par chave e valor associado.

Como todas as estruturas com variáveis compostas que já aprendemos nesta aula, podemos imprimir somente o valor referenciando o índice que desejarmos. Como o índice no dicionário é a chave e não um número, colocamos:

```
1 print(game['nome'])  
2 print(game['desenvolvedora'])  
3 print(game['ano'])  
  
Super Mario  
Nintendo  
1990
```

Existem alguns métodos que auxiliam neste processo de obter as informações de um dicionário. Se quisermos somente imprimir todos os valores do dicionário, invocamos o método/função *values*.



```
1 print(game.values())  
  
dict_values(['Super Mario', 'Nintendo', 1990])
```

Podemos realizar uma varredura pelos valores do dicionário com um laço, assim como fizemos em listas e tuplas:

```
1 for i in game.values():  
2     print(i)  
  
Super Mario  
Nintendo  
1990
```

É possível realizarmos o acesso somente das chaves de todo o dicionário utilizando método/função *keys* (que significa chave, em inglês). Note que foi impresso as três chaves: *nome*, *desenvolvedora*, *ano*.

```
1 print(game.keys())  
  
dict_keys(['nome', 'desenvolvedora', 'ano'])
```

Assim como para os valores, podemos andar por todas as chaves usando um laço de repetição:

```
1 for i in game.keys():  
2     print(i)  
  
nome  
desenvolvedora  
ano
```

Existe um terceiro método/função que acessa o par chave e valor, que é o *items*.

```
1 print(game.items())  
  
dict_items([('nome', 'Super Mario'), ('desenvolvedora', 'Nintendo'), ('ano', 1990)])
```

Também podemos varrer o dicionário acessando sempre a dupla de chave e valor. Para isso, fazemos um laço de repetição com duas variáveis. A primeira será para as chaves (*i*) e a segunda para os valores (*j*).



```
1 for i,j in game.items():
2     print('{} = {}'.format(i, j))

nome = Super Mario
desenvolvedora = Nintendo
ano = 1990
```

#### 4.1 Listas com dicionários

Aprendemos a criar um dicionário. Mas e se quisermos criar uma coleção de itens dentro do dicionário? Em vez de somente um nome de jogo de *videogame*, por exemplo, inserirmos vários. Podemos construir essa estrutura combinando uma lista com um dicionário em que cada índice da lista conterà um cadastro completo de um dicionário. Vejamos o exemplo a seguir.

```
1 games = []
2 game1 = {'nome': 'Super Mario',
3         'videogame': 'Super Nintendo',
4         'ano': 1990}
5 game2 = {'nome': 'Zelda Ocarina of Time',
6         'videogame': 'Nintendo 64',
7         'ano': 1998}
8 game3 = {'nome': 'Pokemon Yellow',
9         'videogame': 'Game Boy',
10        'ano': 1999}
11 games = [game1, game2, game3]
12 print(games)

[{'nome': 'Super Mario', 'videogame': 'Super Nintendo', 'ano': 1990},
{'nome': 'Zelda Ocarina of Time', 'videogame': 'Nintendo 64', 'ano': 1998},
{'nome': 'Pokemon Yellow', 'videogame': 'Game Boy', 'ano': 1999}]
```

Nele, criamos uma lista chamada *games*. A lista inicia vazia. Em seguida, estamos criando três variáveis de dicionário *game1*, *game2* e *game3*. Todos os dicionários contêm três chaves: *nome*, *desenvolvedora* e *ano*. Na linha 11, juntamos todos os dicionários dentro da lista *games*, criando uma coleção de jogos de *videogame*.

É claro que a maneira apresentada acima não é dinâmica o suficiente, pois só permite três cadastros. Podemos criar uma inserção via teclado dos dados empregando um laço de repetição. Para isso, criamos uma estrutura de lista e uma de dicionário vazias. A lógica aqui é povoar o dicionário com dados do teclado e em seguida colocar o dicionário dentro da lista fazendo um *append*. Veja a seguir o código.



```
1 game = {}
2 games = []
3 for i in range(3):
4     game['nome'] = input('Qual o nome do jogo?')
5     game['videogame'] = input('Para qual video-game ele foi lançado?')
6     game['ano'] = input('Qual o ano de lançamento?')
7     games.append(game.copy())
8     print('-' * 20)
9     for e in games:
10         for i,j in e.items():
11             print('O campo {} tem o valor {}'.format(i,j))
```

Dentro do laço de repetição, lemos os dados via teclado inserindo-os nas respectivas chaves *nome*, *videogame* e *ano*. Na linha 7, fazemos o *append* para a inserção de um novo elemento na lista. Neste momento, precisamos utilizando uma função/método chamado de *copy*, pois devido às características da estrutura de dicionário, não podemos fazer algo como *game.append(game[:])*, isso não irá funcionar. Nas linhas 8 até 11, fazemos o *print* de tudo que foi cadastrado utilizando dois laços de repetição. O laço da linha 9 serve para andar pela lista, e o laço interno da linha 10 serve para andar dentro de cada dicionário colocado em um índice da lista.

## 4.2 Dicionários com listas

Podemos fazer o processo contrário do que você acabou de aprender. Em vez de criar uma lista na qual, em cada índice, conterà um dicionário. Podemos criar um único dicionário e, para cada palavra-chave, teremos uma lista.

A seguir, você encontra o código exemplo. Compare bem a saída desse código com a saída dos exemplos de listas com dicionários. Observe bem onde ficam os colchetes e as chaves. Note que agora temos uma chave envolvendo toda a estrutura e dentro das chaves temos colchetes representando listas. A palavra-chave *nome* contém sua lista de nomes, assim como *videogame* e *ano*.

```
1 games = {'nome': ['Super Mario', 'Zelda Ocarina of Time', 'Pokemon Yellow'],
2          'videogame': ['Super Nintendo', 'Nintendo 64', 'Game Boy'],
3          'ano': [1990, 19998, 1999]}
4 print(games)

{'nome': ['Super Mario', 'Zelda Ocarina of Time', 'Pokemon Yellow'],
 'videogame': ['Super Nintendo', 'Nintendo 64', 'Game Boy'],
 'ano': [1990, 19998, 1999]}
```

Também podemos criar a inserção dos dados no dicionário com listas via teclado, assim como víamos anteriormente. Veja o código a seguir.



```
1 games = {'nome':[], 'videogame':[], 'ano':[]}  
2 for i in range(3):  
3     nome = input('Qual o nome do jogo?')  
4     videogame = input('Para qual video-game ele foi lançado?')  
5     ano = input('Qual o ano de lançamento?')  
6     games['nome'].append(nome)  
7     games['videogame'].append(videogame)  
8     games['ano'].append(ano)  
9     print('-' * 20)  
10    print(games)
```

Na linha 1, criamos a estrutura base do dicionário, já inserindo as palavras-chave e criando para cada uma delas uma lista completamente vazia. Em seguida, dentro do laço de repetição, fazemos a inserção dos dados em variáveis simples e, em seguida, fazemos um *append* na respectiva lista do dicionário.

### 4.3 Exercícios

Vamos praticar um pouco o conceito de dicionários e suas relações com listas.

#### 4.3.1 Exercício

Escreva um programa em Python que leia o nome de um aluno e três notas. Armazene em um dicionário o nome e a média aritmética da nota. Ainda, armazene no dicionário a situação do aluno:

- Média  $\geq 7$ , aprovado.
- Média  $< 7$  e  $\geq 5$ , em exame.
- Média  $< 5$ , reprovado.

Apresente tudo na tela ao final do programa em um formato organizado.



```
1 # Exercício 1
2 aluno = {}
3 aluno['nome'] = input('Qual o nome do aluno?')
4 n1 = float(input('Qual a primeira nota?'))
5 n2 = float(input('Qual a segunda nota?'))
6 n3 = float(input('Qual a terceira nota?'))
7 aluno['media'] = (n1 + n2 + n3) / 3
8 if aluno['media'] >= 7:
9     aluno['status'] = 'A'
10 elif aluno['media'] >= 5 and aluno['media'] < 7:
11     aluno['status'] = 'E'
12 else:
13     aluno['status'] = 'R'
14
15 for i, j in aluno.items():
16     print('{ } = {}'.format(i, j))
```

Qual o nome do aluno?Lenhadorzinho  
Qual a primeira nota?5.5  
Qual a segunda nota?8.9  
Qual a terceira nota?4.7  
nome = Lenhadorzinho  
media = 6.366666666666667  
status = E

### 4.3.2 Exercício

Crie um programa em Python para controle de estoque de produtos de um estabelecimento que vende produtos de hortifruti. Para o estoque, armazene tudo dentro de um dicionário contendo listas. A chave deverá ser o nome de cada produto e dentro de cada lista teremos o preço e a quantidade disponível no estoque. O estoque pode estar pré-cadastrado no sistema com quantos itens desejar.

Simule uma compra. Peça ao usuário para digitar o nome do produto e a quantidade que deseja até que ele decida encerrar a compra. Ao final, apresente tudo na tela em um formato organizado, mostrando o total a ser pago por produto e o total final do pedido.

Ainda, dê baixa no sistema descontado o que foi comprado do total. Imprima na tela o estoque restante.



```
1  # Exercício 2
2  loja = {'cenoura':[100, 0.99],
3          'brócolis':[50, 3.99],
4          'batata':[200, 0.49],
5          'cebola':[75, 1.10]}
6
7  pedido = []
8  while True:
9      item_nome = input('Digite o nome do item que deseja comprar: ')
10     item_qtd = int(input('Deseja comprar quantos?'))
11     pedido.append([item_nome, item_qtd])
12     res = input('Desejar adicionar outro item? [S/N]')
13     if res in 'Nn':
14         break
15     total = 0
16     print('\nVendas:')
17     for item in pedido:
18         produto = item[0]
19         qtd = item[1]
20         preco = loja[produto][1]
21         valor_produto = preco * qtd
22         print('{}: {} x {} = {}'.format(produto, qtd, preco, valor_produto))
23         loja[produto][0] -= qtd
24         total += valor_produto
25     print('Custo total: {}'.format(total))
26     print('Estoque:')
27     for i, j in loja.items():
28         print('Descrição:', i)
29         print('Quantidade:', j[0])
30         print('Preço: {}'.format(j[1]))
```

## TEMA 5 – TRABALHANDO COM MÉTODOS EM *STRINGS*

Você aprendeu, anteriormente, a manipular *strings*. Aprendeu a fatiar, concatenar, encontrar o tamanho etc. Porém, *strings* é um assunto bastante vasto e que apresenta muitas funcionalidades. Agora, que você aprendeu nesta aula o que são métodos e como usá-los, e também aprendeu o conceito de listas, podemos praticar um pouco mais com *strings*. Vamos encerrar nossa disciplina aprendendo mais algumas funcionalidades com *strings* para que você fique com um conhecimento sólido neste assunto.

O primeiro assunto que é importante comentar é que uma *string*, uma vez criada, é imutável, assim como uma tupla. Portanto, alterar um dado dentro dela é impossível. Veja a seguir o erro.



```
1 s1 = 'Algoritmos'
2 print(s1)
3 s1[0] = 'a'
```

Algoritmos

-----

TypeError Traceback (most recent call last)

<ipython-input-25-8c04c5f6db42> in <module>()

1 s1 = 'Algoritmos'

2 print(s1)

----> 3 s1[0] = 'a'

TypeError: 'str' object does not support item assignment

SEARCH STACK OVERFLOW

Apesar disso, agora que você aprendeu listas, podemos transformar uma *string* em uma lista, utilizando a função *list*, conforme a seguir. Se fizermos o *print* convencional, veremos que cada caractere será impresso separado por vírgula, caracterizando uma lista. Podemos fazer o *print* em um formato de texto caso fizermos como apresentado na linha 3.

```
1 s1 = list('Algoritmos')
2 print(s1) #print separado
3 print(''.join(s1)) #print agrupado
```

['A', 'l', 'g', 'o', 'r', 'i', 't', 'm', 'o', 's']

Algoritmos

## 5.1 Verificando caracteres

Podemos verificar se uma *string* qualquer é iniciada com um ou mais caracteres. Para fazermos essa verificação, utilizamos o método *startswith* (*início com*, em tradução livre). Fazemos:

```
1 s1 = 'Lógica de Programação e Algoritmos'
2 s1.startswith('Lógica')
```

True

No exemplo, estamos verificando se a palavra *lógica* está contida na *string* *s1*. Caso exista, o resultado aparece como *True*. Podemos testar o final da *string* com *endswith* (*termina com*, em tradução livre). Veja a seguir.

```
1 s1 = 'Lógica de Programação e Algoritmos'
2 s1.endswith('Algoritmos')
```

True



Um detalhe interessante é que os métodos *startswith* e *endswith* diferenciam caracteres maiúsculos de minúsculos. Ou seja, verificar se a palavra *algoritmos* está presente é diferente de verificar se essa palavra existe. Veja o que acontece a seguir.

```
1 s1 = 'Lógica de Programação e Algoritmos'
2 s1.endswith('algoritmos')

False
```

Veja que agora a palavra procurada não existe devido ao caractere minúsculo, retornando *False*. E se quisermos que o teste ignore se os caracteres são maiúsculos ou minúsculos, é possível? Sim. Podemos contornar isso convertendo todos os caracteres da *string* *s1* para minúsculo antes de realizar a procura, fazemos isso com o método *lower*. Veja a seguir.

```
1 s1 = 'Lógica de Programação e Algoritmos'
2 s1.lower().endswith('algoritmos')

True
```

Agora, como toda a *string* está minúscula, procurar por *algoritmos* irá retornar verdadeiro. Se o método *lower* converte toda a *string* para minúsculo, o método *upper* converte tudo para maiúsculo. Veja a seguir como fica *s1* com ambos os métodos.

```
1 s1 = 'Lógica de Programação e Algoritmos'
2 print(s1.upper())
3 print(s1.lower())

LÓGICA DE PROGRAMAÇÃO E ALGORITMOS
lógica de programação e algoritmos
```

## 5.2 Contando caracteres

Podemos contar a ocorrência de um caractere (ou vários) dentro de uma *string* com o método *count*. Vamos buscar pela letra *a* dentro da *string* *s1*, veja o resultado a seguir.

```
1 s1 = 'Lógica de Programação e Algoritmos'
2 s1.count('a')

3
```



Note que, novamente, se estamos buscando pela letra *a*, em minúsculo, a maiúscula não será contabilizada. Mas, se realizarmos um *lower* no *string* *s1*, antes de fazer a busca, iremos considerar todos os caracteres do texto. Veja a diferença no resultado:

```
1 s1 = 'Lógica de Programação e Algoritmos'
2 s1.lower().count('a')

4
```

Podemos buscar por um conjunto específico dentro de uma *string*, como uma palavra, por exemplo. Veja o exemplo a seguir.

```
1 s1 = 'Um mafagafinho, dois mafagafinhos, três mafagafinhos...'
2 s1.lower().count('mafagafinho')

3
```

### 5.3 Quebrando *strings*

É possível empregarmos um método que divida uma *string* em *substrings*, baseando-se em um caractere como divisor. O método responsável por isso é o *split*, o qual recebe como parâmetro qual caractere será usado para dividir.

```
1 s1 = 'Um mafagafinho, dois mafagafinhos, três mafagafinhos...'
2 s1.split(' ')

['Um', 'mafagafinho,', 'dois', 'mafagafinhos,', 'três', 'mafagafinhos...']
```

A *string* foi quebrada sempre onde apareceu um espaço, criando uma lista em que cada dado dela é uma palavra.

### 5.4 Substituindo *strings*

É possível substituir um caractere ou mesmo uma palavra dentro de uma *string* com o método *replace*. O método recebe sempre dois parâmetros obrigatórios, o primeiro é a palavra a ser substituída, e o segundo, a nova palavra. Veja a seguir.

```
1 s1 = 'Um mafagafinho, dois mafagafinhos, três mafagafinhos...'
2 s1.replace('mafagafinho', 'gatinho')

'Um gatinho, dois gatinhos, três gatinhos...'
```



Podemos adicionar um terceiro parâmetro, opcional, que nos diz quantas vezes a substituição deve acontecer. Se indicar o valor um, somente uma vez a troca é feita:

```
1 s1 = 'Um mafagafinho, dois mafagafinhos, três mafagafinhos...'
2 s1.replace('mafagafinho', 'gatinho', 1)

'Um gatinho, dois mafagafinhos, três mafagafinhos...'
```

## 5.5 Validando tipos de dados

A linguagem Python contém inúmeras funções capazes de validar se os dados dentro de uma *string* atendem a um determinado critério. Por exemplo, o método *isalnum* testa se a *string* contém caracteres e números dentro dela, somente. Caso ela esteja vazia ou contenha caracteres especiais, retornará *False*. Veja a seguir duas *strings*. A primeira contém letra e espaços. A segunda, somente números.

```
1 s1 = 'Lógica de Programação e Algoritmos'
2 s2 = '42'
```

Ao fazermos *isalnum* nelas, a primeira irá retornar *False*, pois contém espaços. Já a segunda retorna verdadeiro devido à existência somente de números.

```
1 print(s1.isalnum())
2 print(s2.isalnum())

False
True
```

Existe também o método *isalpha*, que é mais restritivo que *isalnum*, retornando verdadeiro somente se na *string* existir somente caracteres regulares e com acentuação. Qualquer outra situação é *False*. Veja:

```
1 print(s1.isalpha())
2 print(s2.isalpha())

False
False
```



Ambas retornaram falso, pois na primeira *string* temos espaços e na segunda *string*, números. Caso removêssemos os espaços da primeira, irá resultar em verdadeiro, veja:

```
1 s1 = 'Lógica de Programação e Algoritmos'
2 print(s1.isalpha())

True
```

A quantidade de funções/métodos de validação de *strings* é bastante vasta, bem como a sua aplicabilidade. Qualquer programa desenvolvido com um caráter mais profissional deve se preocupar com a validação dos dados, portanto, nunca esqueça da existência dessas funções. Deixo a seguir uma tabela contendo todas as principais possibilidades e seu respectivo objetivo:

Tabela 1 – Relação de métodos para validação de dados em *strings*

Função/método	Retorna <i>true</i> para uma <i>string</i> com...
<b><i>isalnum</i></b>	Letras e números somente. Acentos são aceitos.
<b><i>isalpha</i></b>	Letras somente. Acentos são aceitos.
<b><i>isdigit</i></b>	Números somente.
<b><i>isnumeric</i></b>	Números somente. Aceita também caracteres matemáticos, como frações.
<b><i>isupper</i></b>	Caracteres maiúsculos somente.
<b><i>islower</i></b>	Caracteres minúsculos somente.
<b><i>isspace</i></b>	Espaços somente. Inclui TAB, quebra de linha, retorno etc.
<b><i>isprintable</i></b>	Caracteres possíveis de serem impressos na tela somente.

Fonte: Borin, 2021.

## 5.6 Resumo

A linguagem Python é vasta quando se trata de funções/métodos para uso em *strings*. A seguir, você encontra um resumo dos principais métodos mostrados neste Tema 5, bem como mais alguns que poderão vir a ser úteis para você.

Tabela 2 – Relação de métodos para uso com *strings*

Função/método	Objetivo
<b><i>startswith</i></b>	Verifica se caracteres existem no início da <i>string</i> .
<b><i>endswith</i></b>	Verifica se caracteres existem no final da <i>string</i> .
<b><i>lower</i></b>	Converte <i>string</i> para minúscula.
<b><i>upper</i></b>	Converte <i>string</i> para maiúscula.
<b><i>find</i></b>	Busca a primeira ocorrência de um padrão de caracteres em uma <i>string</i> .
<b><i>rfind</i></b>	Idêntico ao <i>find</i> , mas inicia a busca da direita para a esquerda.
<b><i>center</i></b>	Centraliza uma <i>string</i> .
<b><i>ljust, rjust</i></b>	Ajusta uma <i>string</i> com alinhamentos à esquerda, ou à direita, respectivamente.
<b><i>split</i></b>	Divide uma <i>string</i> .
<b><i>replace</i></b>	Substitui caracteres em uma <i>string</i> .



<b><i>lstrip, rstrip</i></b>	Remove espaços em branco à esquerda, ou à direita, respectivamente.
<b><i>strip</i></b>	Remove espaços em branco das extremidades.

Fonte: Borin, 2021.

## FINALIZANDO

Nesta aula, aprendemos a manipular as três principais estruturas de dados da linguagem Python: tuplas, listas e dicionários.

As tuplas são as mais simples e são imutáveis. As listas e os dicionários apresentam características mais dinâmicas e, portanto, são mais empregadas. O que diferencia listas de dicionários é que as primeiras são indexadas por valores inteiros numéricos, ao passo que os dicionários são indexados por chaves (palavras-chave).

Na representação em Python, cada uma delas é indicada por:

- Tuplas: ()
- Listas: []
- Dicionários: {}

Para finalizar, a seguir, você encontra um quadro comparativo entre as três estruturas, com diferenças e aplicações.

Tabela 3 – Resumo das estruturas de dados estudadas

	<b>Tuplas</b>	<b>Listas</b>	<b>Dicionários</b>
<b>Ordem dos elementos</b>	Fixa	Fixa	Mantida a partir do Python 3.7
<b>Tamanho</b>	Fixo	Variável	Variável
<b>Elementos repetidos</b>	Sim	Sim	Pode repetir valores, mas as chaves devem ser únicas
<b>Pesquisa</b>	Sequencial, índice numérico	Sequencial, índice numérico	Direta por chave
<b>Alterações</b>	Não	Sim	Sim
<b>Uso primário</b>	Sequências constantes	Sequências	Dados indexados por chave

Fonte: Elaborado com base em Nilo, p 137.

Aprendemos também que podemos realizar diferentes manipulações com essas estruturas invocando métodos, que são semelhantes a funções que aprendemos anteriormente.

Esta aula contém os conceitos mais avançados da disciplina, portanto, pratique e resolva todos os exercícios do seu material com bastante calma.





## REFERÊNCIAS

FORBELLONE, A. L. V. et al. **Lógica de programação**: a construção de algoritmos e estruturas de dados. 3. ed. São Paulo: Pearson, 2005.

MATTHES, E. **Curso intensivo de Python**: uma introdução prática baseada em projetos à programação. 1. ed. São Paulo: Novatec, 2015.

MENEZES, N. N. C. **Introdução à programação Python**: algoritmos e lógica de programação para iniciantes. 3. ed. São Paulo: Novatec, 2019.

PERKOVIC, L. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

PUGA, S.; RISSETI, G. **Lógica de programação e estrutura de dados**. 3. ed. São Paulo: Pearson, 2016.