PYTHON DEVELOPMENT

COLEÇÕES

4

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

COLEÇÕES

1 ESTRUTURAS PARA DAR E VENDER

1.1 Por que as variáveis não são o suficiente?

Desde o início deste curso, você já aprendeu a lidar com diferentes tipos de variáveis com base nos tipos de dados que eram necessários armazenar em cada situação. Porém, assim como o mundo para James Bond, para nós as variáveis não são o bastante.

Imagine um *e-commerce* com diversos produtos de diversas categorias prontos para encher os carrinhos de todos os clientes, em seguida, faça a si mesmo esta pergunta: todos os clientes comprarão a mesma quantidade de produtos? Não, certo? Se uma lista de produtos muda de cliente para cliente, quantas variáveis você precisaria criar para resolver esse problema?

Se a sua cabeça deu um nó, não se preocupe! Provavelmente a resposta ainda não apareceu porque falta aprender quais são e como funcionam as estruturas de dados que o Python tem para trabalhar com coleções, ou seja, as estruturas que o Python usa para trabalhar com conjuntos de dados.

Apesar de ainda não conhecer as estruturas que iremos explorar neste capítulo, basta ter em mente a inviabilidade de criar uma variável para cada um dos itens que o cliente adicionar ao carrinho ou, ainda, a dificuldade de se manipular essa quantidade tão grande de variáveis individuais.

Agora que já entendemos o problema, vamos às soluções! Prepare-se para conhecer um dos superpoderes do Python.

1.2 Uma lista de tarefas

Uma das formas que o Python oferece para lidarmos com coleções de dados são as *lists*. Uma lista nada mais é do que uma estrutura que suporta diversos dados simultaneamente, ela é dinâmica (pode crescer ou diminuir de tamanho) e heterogênea (suporta dados de diferentes tipos).

Agora que vamos trabalhar com estruturas que vão além das variáveis que já conhecíamos, precisaremos ficar muito atentos aos símbolos que representam essas estruturas. No caso das listas, este símbolo é o [] (abrir e fechar colchetes).

Veja, no script do Código-fonte “Criação de listas em Python”, duas formas de criar uma lista em Python:

```
    lista1 = [] #criação de uma lista vazia

    lista2 = [1, 7, 8, 9] #criação de uma lista com números
inteiros

    lista3 = [1, 73.5, "texto", True] #criação de uma lista
com dados de diferentes tipos

    print(f"A lista 1 é do tipo {type(lista1)}")
    print(f"A lista 2 é do tipo {type(lista2)}")
    print(f"A lista 3 é do tipo {type(lista3)}")
```

Código-fonte 1 – Criação de listas em Python
Fonte: Elaborado pelo autor (2022)

É possível notar que mesmo que a lista 1 não tenha nenhum elemento armazenado, a lista 2 tenha apenas elementos do tipo int e a lista 3 tenha elementos de diferentes tipos, as três são do tipo *list*. E isso é ótimo, porque nos permite aplicar qualquer um dos métodos das listas, independente de seu conteúdo.

Mas o que acontece se tentarmos exibir essas estruturas?

```
    lista1 = [] #criação de uma lista vazia

    lista2 = [1, 7, 8, 9] #criação de uma lista com números
inteiros

    lista3 = [1, 73.5, "texto", True] #criação de uma lista
com dados de diferentes tipos

    print(f"A lista 1 é do tipo {lista1}")
```

```
    print(lista1)
    print(f"A lista 2 é do tipo {lista2}")
    print(lista2)
    print(f"A lista 3 é do tipo {lista3}")
    print(lista3)
```

Código-fonte 2 – Exibição de listas
Fonte: Elaborado pelo autor (2022)

Quando printamos uma lista, o Python a exibe de uma forma muito parecida com a que vemos no código-fonte, ou seja: sinal de colchetes para iniciar, elementos separados por vírgula, sinal de colchetes para encerrar. Por mais que essa visualização seja útil e compreensível, não é a mais amigável para os usuários.

Para a nossa sorte, a lista é uma estrutura iterável do Python e, se ela é iterável, podemos utilizar um loop for para percorrer todos os seus elementos. Veja o script abaixo:

```
    cavaleiros_jedi = ["Obi-Wan Kenobi", "Mace Windu",
"Mestre Yoda", "Luke Skywalker", "Anakin Skywalker"]

    for jedi in cavaleiros_jedi:
        print(f"{jedi} é um dos cavaleiros da ordem Jedi")
```

Código-fonte 3 – Percorrendo uma lista com o loop for Python
Fonte: Elaborado pelo autor (2022)

Ao utilizar a lista como o intervalo do nosso loop for, foi possível percorrê-la e capturar cada um de seus elementos. Mas e se você precisar de um elemento específico ou de um intervalo de elementos? Para isso, podemos usar os índices!

Os índices são números inteiros iniciados em 0, que indicam a posição de um determinado elemento dentro da estrutura de dados. No nosso script anterior, por exemplo, o elemento “Obi-Wan Kenobi” tem índice 0, enquanto o elemento “Luke Skywalker” tem índice 3.

Para recuperar um elemento com base no seu índice, devemos utilizar o nome da lista seguida do índice indicado entre colchetes. Já para recuperarmos um intervalo de elementos, devemos indicar, entre colchetes, a posição inicial seguida de dois pontos seguida da posição final. Observe algumas formas de trabalhar com índices no script a seguir:

```
    cavaleiros_jedi = ["Obi-Wan Kenobi", "Mace Windu", "Mestre Yoda", "Luke Skywalker", "Anakin Skywalker"]

    print(cavaleiros_jedi[0]) #Exibe o primeiro elemento da lista
    print(cavaleiros_jedi[1]) #Exibe o segundo elemento da lista
    print(cavaleiros_jedi[-1]) #Exibe o último elemento da lista

    print(cavaleiros_jedi[1:3]) #Exibe os elementos que estão entre os índices 1 e 3, onde o 3 não está incluído

    print(cavaleiros_jedi[:3]) #Exibe os elementos que estão entre o início da lista e o índice 3, onde o 3 não está incluído
    print(cavaleiros_jedi[2:]) #Exibe os elementos que estão entre o índice 2 e o final da lista
```

Código-fonte 4 – Seleção de elementos de uma lista com base nos índices
Fonte: Elaborado pelo autor (2022)

Outra grande vantagem que as listas nos apresentam são os métodos que podem ser utilizados. Para simplificar o conceito dentro da abordagem deste curso, podemos imaginar os métodos como ações que uma determinada estrutura é capaz de realizar. No caso das listas, as principais ações são:

Método	Efeito
lista.append(elemento)	Adiciona um elemento no final da lista.
lista.index(elemento)	Retorna o índice de um determinado elemento.
lista.insert(posicao, item)	Insere um elemento em uma posição específica, fazendo com que os elementos posteriores a ele recebam novos índices.
lista.pop()	Remove o último elemento de uma lista.
lista.pop(posicao)	Remove o elemento que está em uma posição específica da lista.
lista.count("item")	Retorna o número de vezes que um elemento aparece na lista.
lista.sort()	Ordena a lista em ordem crescente.
lista.sort(reverse=True)	Ordena a lista em ordem decrescente.
lista.reverse()	Inverte a ordem dos elementos de uma lista.
lista.remove(item)	Remove um item da lista.

Quadro 1 – Métodos da lista em Python
Fonte: Elaborado pelo autor (2022)

Para testarmos todos esses métodos, vamos criar o script seguinte:

```
    cavaleiros_jedi = ["Obi-Wan Kenobi", "Mace Windu",
"Mestre Yoda", "Luke Skywalker", "Anakin Skywalker"]

    print(f"A lista original é \n{cavaleiros_jedi}")

    cavaleiros_jedi.append("Mestre Yoda")
    print(f"Após a adição do elemento Mestre Yoda utilizando
o append, a lista é \n{cavaleiros_jedi}")

    print(f"O índice do elemento Mace Windu é
{cavaleiros_jedi.index('Mace Windu')}")

    cavaleiros_jedi.insert(1, "Rey")
    print(f"Após a adição do elemento Rey na posição 1
utilizando o insert, a lista é \n{cavaleiros_jedi}")

    print(f"A contagem do elemento Mestre Yoda é
{cavaleiros_jedi.count('Mestre Yoda')}")

    cavaleiros_jedi.pop()
    print(f"Após o uso do pop, a lista é
\n{cavaleiros_jedi}")

    cavaleiros_jedi.pop(0)
    print(f"Após o uso do pop com o índice 0, a lista é
\n{cavaleiros_jedi}")

    cavaleiros_jedi.reverse()
    print(f"Após o uso do reverse, a lista é
\n{cavaleiros_jedi}")

    cavaleiros_jedi.sort()
    print(f"Após o uso do sort, a lista é
\n{cavaleiros_jedi}")

    cavaleiros_jedi.sort()
    print(f"Após o uso do sort(reverse=True), a lista é
\n{cavaleiros_jedi}")
```

Código-fonte 5 – Teste dos métodos da lista em Python
Fonte: Elaborado pelo autor (2022)

Para colocarmos a mão na massa pensando em um caso real, que tal criarmos um programa onde o usuário informa quantas calorias consumiu ao longo do dia e, ao final, exibimos todas as calorias informadas, o total e a média aritmética?

```
    calorias = []

    resposta = ""

    while resposta.upper() != "NÃO":
        caloria = int(input("Por favor, informe a quantidade
de calorias consumida nesta refeição: "))
        calorias.append(caloria)
        resposta = input("Digite SIM para informar mais uma
caloria ou NÃO para encerrar a digitação")

    total = 0

    print("As calorias informadas para este dia foram: ")
    for caloria in calorias:
        total = total + caloria
        print(caloria)

    media = total / len(calorias)
    print(f"Ao longo do dia foram consumidas {total}
calorias, com uma média de {media:.2f} calorias por refeição")
```

Código-fonte 6 – Programa do consumo de calorias
Fonte: Elaborado pelo autor (2022)

As listas são estruturas extremamente úteis e flexíveis e, justamente por isso, cobram um preço em termos de consumo de recursos do computador. Que tal conhecermos uma estrutura semelhante, mas com menos uso de espaço em memória RAM?

1.3 Trabalhando em tuplas

Se você se equivocasse na hora de criar uma lista e utilizasse parênteses ao invés de colchetes, descobriria que seu objeto teria um outro tipo e seria, na verdade, uma *tuple* ou tupla.

Apesar de parecer que a diferença de uma *tuple* para uma *list* é apenas o símbolo utilizado na sua criação, a questão central está na mutabilidade. Tente executar o script abaixo:

```
tupla = ()

tupla.append("TESTE")

print(tupla)
```

Código-fonte 7 – Tentativa de modificar a tupla
Fonte: Elaborado pelo autor (2022)

Certamente você se deparou com um erro de execução na linha “tupla.append (“TESTE”)” e isso acontece por um detalhe muito simples: as tuplas não possuem método append().

Então você pode se perguntar: como incluir novos elementos na tupla? Pode ser a pergunta que está na sua mente agora, e a resposta é que você não poderá incluir, pois as tuplas são estruturas imutáveis, ou seja, depois de criadas não é possível modificar seu conteúdo. Veja:

```
categorias_jedi = ("Youngling", "Padawan", "Cavaleiro", "Mestre")

print(categorias_jedi) #exibindo toda a tupla

print(categorias_jedi[0]) #exibindo o primeiro elemento

print(categorias_jedi[-1]) #exibindo o último elemento

#iterando pela tupla
for categoria in categorias_jedi:
    print(categoria)

print(len(categorias_jedi)) #exibindo o tamanho da tupla
```

Código-fonte 8 – Tentativa de modificar a tupla
Fonte: Elaborado pelo autor (2022)

É possível recuperar um ou mais elementos da tupla, iterá-la e até recuperar seu tamanho utilizando a função *len()*, mas nenhuma operação de alteração do conteúdo está disponível. Para que, então, as tuplas podem ser úteis?

Utilizamos tuplas sempre que precisamos armazenar conjuntos de dados que não sofrerão alteração, mas, ainda assim, dependem de “habilidades” de iteração. Por não sofrerem alterações, elas também podem ser utilizadas como chaves em dicionários (como veremos um pouco mais adiante) e como argumentos em funções.

Mas para não ficarmos apenas na teoria do que as tuplas podem fazer, veja que interessante é o recurso de “desempacotar” uma tupla:

```
    tupla = ("André", 88, 1.65)

    print(f"A tupla é {tupla}")

    nome, peso, altura = tupla

    print(f"Após desempacotar a tupla, a variável nome contém {nome},  a variável peso contém {peso}, a variável altura contém {altura}")
```

Código-fonte 9 – Desempacotando valores de uma tupla
Fonte: Elaborado pelo autor (2022)

O exemplo anterior nos mostra que se tivermos uma tupla do lado direito do sinal de igual (ou seja, fornecendo os valores) e uma quantidade de variáveis compatível com a quantidade de elementos da tupla do lado esquerdo (ou seja, recebendo valores), o Python será capaz de desempacotar esses valores e armazenar corretamente em cada uma das variáveis.

É importante destacar que também é possível desempacotar listas, mas o fato de serem mutáveis acrescenta um componente de possível instabilidade no script, uma vez que a estrutura pode ter mais ou menos itens do que o esperado (por poder ser modificada).

Para compreendermos que as tuplas e listas não são inimigas, que tal fazermos um script que utilize ambas as estruturas em conjunto? Vamos criar uma lista que contenha tuplas que representarão as posições X e Y de inimigos no mapa de um jogo qualquer.

```
    inimigos = [(10, 15), (30, 30), (15, 25), (7, 10)]

    for x, y in inimigos:
        print(f"O inimigo está na posição X={x} e Y={y}")
```

Código-fonte 10 – Criação e exibição de lista de tuplas
Fonte: Elaborado pelo autor (2022)

A cada volta do nosso loop for, passamos por um dos elementos da lista “inimigos”, e como cada um desses elementos é uma tupla, podemos desempacotá-los.

Para melhorar ainda mais nosso exemplo, vamos pedir que o usuário digite valores para x e y e, se estiverem na lista, vamos remover o inimigo correspondente:

```
    inimigos = [(10, 15), (30, 30), (15, 25), (7, 10)]

    for x, y in inimigos:
        print(f"O inimigo está na posição X={x} e Y={y}")

    x = int(input("Digite a posição X do inimigo que deseja
acertar "))
    y = int(input("Digite a posição Y do inimigo que deseja
acertar "))

    if (x, y) in inimigos:
        print("ACERTOU!!")
        inimigos.remove((x, y))
    else:
        print("Não foi encontrado nenhum inimigo na posição
indicada ")

    print(f"Os inimigos ainda existentes são: {inimigos}")
```

Código-fonte 11 – Remoção de tuplas de uma lista
Fonte: Elaborado pelo autor (2022)

Apesar de a tupla ter aspecto de lista, pudemos ver que a forma de utilizar (e os cenários onde é possível fazer isso) são bem distintas entre si. Mas se você ainda não se cansou dessa “cara”, vamos conhecer mais uma estrutura muito semelhante: o set!

1.4 Pintando o *set*

As listas são mutáveis e apresentam uma série de métodos para a validação dos dados, já as tuplas são imutáveis e, por essa razão, não apresentam métodos de alteração do conteúdo. Independente disso, não existem limitações nessas estruturas quanto ao seu conteúdo.

Como estamos aprendendo, as limitações que as estruturas nos apresentam são, na verdade, características que podem acabar solucionando outros problemas para nós. E é isso que acontece com os *sets* ou *conjuntos*.

Os sets são estruturas do Python que permitem armazenar diversos valores assim como as listas, mas não permite repetições. Dessa forma, eles se tornam candidatos perfeitos para todos os cenários em que precisamos trabalhar com conjuntos de valores únicos.

Por representarem conjuntos, os sets contêm muitos métodos que podem ser utilizados. Veja o Quadro “Métodos do set em Python” para ter uma ideia dos “superpoderes” que essa estrutura nos apresenta:

Método	Efeito
set.add(elemento)	Adiciona um elemento ao set caso ele ainda não exista.
set.update(estrutura)	Adiciona a um set elementos de outro set ou outra estrutura iterável.
set.difference_update(set)	Remove de um set os elementos que também existem em outro set.
set.intersection_update(set)	Remove de um set os elementos que não existem em outro set.
set.symmetric_difference_update(set)	Atualiza o set com os elementos que não se repetem em ambos.
set.difference(set1, set2)	Retorna um set com os elementos que não se repetem em dois sets.
set.symmetric_difference(set1, set2)	Retorna um set com os elementos que não se repetem em dois sets.
set.union(set1, set2, ...)	Retorna um set contendo os elementos dos sets indicados
set.intersection(set1, set2, ...)	Retorna um set com os elementos que existem em todos os sets indicados
set.isdisjoint(set2)	Retorna verdadeiro se os dois sets contêm elementos diferentes e falso se algum dos elementos for igual.
set.issubset(set2)	Retorna verdadeiro se o set está contido em outro.
set.issuperset(set2)	Retorna verdadeiro se um set contém outro set.
set.clear()	Remove todos os elementos do set.
set.copy()	Retorna uma cópia do set.
set.pop()	Remove um elemento aleatório do set.
set.remove()	Remove um elemento do set e retorna um erro caso o elemento não exista.
set.discard()	Remove um elemento do set e não retorna um erro caso o elemento não exista.

Quadro 2 – Métodos do set em Python
Fonte: Elaborado pelo autor (2022)

Com tantos métodos assim, vamos começar devagar para entendermos o efeito de cada grupo deles. Para iniciar, vamos aprender a criar um set, utilizando a sintaxe abaixo:

```
    conjunto = set() #cria um set vazio
    conjunto = set(estrutura iterável) #cria um com o conteúdo de uma estrutura iterável
    conjunto = {item1, item2, item3} #cria um set com os itens indicados
```

Código-fonte 12 – Sintaxe de criação de um set
Fonte: Elaborado pelo autor (2022)

Agora que entendemos que existem três formas de criar um novo set, vamos utilizar os métodos que incluem novos itens:

```
    conjunto = set() #cria um set vazio

    conjunto.add("Cebolinha")
    conjunto.add("Cascão")
    conjunto.add("Mônica")
    conjunto.add("Cebolinha")
    print(f"O conteúdo do set que foi recebendo elementos com o método add() é \n{conjunto}") #Note que não existe repetição de elementos

    lista = ["Cebolinha", "Cascão", "Mônica", "Magali", "Cebolinha"]
    novo_conjunto = set(lista)
    print(f"\nPodemos criar um set a partir de uma lista de um outro set ou de qualquer estrutura iterável. A lista é \n{lista}")
    print(f"O conteúdo do set construído a partir da lista é \n{novo_conjunto}") #Note que não existe repetição de elementos
```

Código-fonte 13 – Métodos add() e set() do set
Fonte: Elaborado pelo autor (2022)

Existem ainda métodos que podem atualizar os elementos de um set com base nos elementos presentes em um segundo set, como: .difference_update(), .intersection_update() e .symmetric_difference_update():

```
    conjunto1 = {"Mega Drive", "Super Nintendo",
"Playstation"}
    conjunto2 = {"Playstation", "Nintendo 64", "Sega Saturn",
"Dreamcast"}

    print(f"O primeiro set contém {conjunto1}")
    print(f"O segundo set contém {conjunto2}")
    conjunto1.difference_update(conjunto2)
    print(f"O primeiro set foi atualizado com o método
.difference_update() e agora contém: \n{conjunto1}")


    print("\nRECONSTRUINDO OS SETS")
    conjunto1 = {"Mega Drive", "Super Nintendo",
"Playstation"}
    conjunto2 = {"Playstation", "Nintendo 64", "Sega Saturn",
"Dreamcast"}

    print(f"O primeiro set contém {conjunto1}")
    print(f"O segundo set contém {conjunto2}")
    conjunto1.intersection_update(conjunto2)
    print(f"O primeiro set foi atualizado com o método
.intersection_update() e agora contém: \n{conjunto1}")

    print("\nRECONSTRUINDO OS SETS")
    conjunto1 = {"Mega Drive", "Super Nintendo",
"Playstation"}
    conjunto2 = {"Playstation", "Nintendo 64", "Sega Saturn",
"Dreamcast"}

    print(f"O primeiro set contém {conjunto1}")
    print(f"O segundo set contém {conjunto2}")
    conjunto1.symmetric_difference_update(conjunto2)
    print(f"O primeiro set foi atualizado com o método
.symmetric_difference_update() e agora contém: \n{conjunto1}")
```

Código-fonte 14 – Métodos difference_update(), intersection_update() e symmetric_difference_update() do set
Fonte: Elaborado pelo autor (2022)

É possível que, ao invés de alterar um dos sets, você queira apenas *retornar* um novo set contendo os elementos que resultaram das comparações. Para isso, podemos utilizar os métodos .difference(), .symmetric_difference(), .union(), .intersection():

```
    conjunto1 = {"Mega Drive", "Super Nintendo",
"Playstation"}
    conjunto2 = {"Playstation", "Nintendo 64", "Sega Saturn",
"Dreamcast"}

    print(f"\nO primeiro set contém {conjunto1}")
    print(f"O segundo set contém {conjunto2}")
    novo_conjunto = set.difference(conjunto1, conjunto2)
    print(f"O novo set foi gerado a partir do método
.difference() usando os sets 1 e 2 e contém:
\n{novo_conjunto}")


    print(f"\nO primeiro set contém {conjunto1}")
    print(f"O segundo set contém {conjunto2}")
    novo_conjunto = set.symmetric_difference(conjunto1,
conjunto2)
    print(f"O novo set foi gerado a partir do método
.symmetric_difference() usando os sets 1 e 2 e contém:
\n{novo_conjunto}")

    print(f"\nO primeiro set contém {conjunto1}")
    print(f"O segundo set contém {conjunto2}")
    novo_conjunto = set.union(conjunto1, conjunto2)
    print(f"O novo set foi gerado a partir do método .union()
usando os sets 1 e 2 e contém: \n{novo_conjunto}")

    print(f"\nO primeiro set contém {conjunto1}")
    print(f"O segundo set contém {conjunto2}")
    novo_conjunto = set.intersection(conjunto1, conjunto2)
    print(f"O novo set foi gerado a partir do método
.intersection() usando os sets 1 e 2 e contém:
\n{novo_conjunto}")
```

Código-fonte 15 – Métodos difference(), .symmetric_difference(), .union(), .intersection() do set
Fonte: Elaborado pelo autor (2022)

Ainda podemos utilizar métodos que retornam valores booleanos, como .isdisjoint(), issubset(), issuperset():

```
    conjunto1 = {"Mega Drive", "Super Nintendo",
"Playstation"}
    conjunto2 = {"Mega Drive", "Super
Nintendo","Playstation", "Nintendo 64", "Sega Saturn",
"Dreamcast"}

    print(f"\nO primeiro set contém {conjunto1}")
    print(f"O segundo set contém {conjunto2}")
    print(f"Ao comparar o set 1 com o set 2 utilizando o
método isdisjoint() obtivemos
\n{conjunto1.isdisjoint(conjunto2)}, pois existem elementos do
set 1 que se repetem no set 2")

    print(f"\nO primeiro set contém {conjunto1}")
    print(f"O segundo set contém {conjunto2}")
    print(f"Ao comparar o set 1 com o set 2 utilizando o
método issubset() obtivemos \n{conjunto1.issubset(conjunto2)},
pois o conjunto de elementos do set 1 está contido no set 2")

    print(f"\nO primeiro set contém {conjunto1}")
    print(f"O segundo set contém {conjunto2}")
    print(f"Ao comparar o set 1 com o set 2 utilizando o
método issuperset(), obtivemos
\n{conjunto1.issuperset(conjunto2)}, pois o conjunto de
elementos do set 2 não está contido no set 1")
```

Código-fonte 16 – Métodos .isdisjoint(), issubset(), issuperset() do set
Fonte: Elaborado pelo autor (2022)

Por fim, vamos testar os últimos métodos disponíveis para o tipo set: .clear(), .copy(), .pop(), .remove() e .discard().

```
    conjunto = {"Mega Drive", "Super Nintendo","Playstation",
"Nintendo 64", "Sega Saturn", "Dreamcast"}
    copia = conjunto.copy()

    print(f"O conjunto original contém {conjunto}")
    print(f"Ao gerar uma cópia utilizando o copy(), obtivemos
{copia}")

    copia.clear()
    print(f"\nAo utilizar o método clear na cópia, ela ficou
assim: \n{copia}")
    print(f"Mas o conjunto original permanece inalterado:
\n{conjunto}")

    conjunto.pop()
```

```
    print(f"\nAo utilizar o método pop() no conjunto, ele ficou assim: \n{conjunto}")


    conjunto = {"Mega Drive", "Super Nintendo","Playstation", "Nintendo 64", "Sega Saturn", "Dreamcast"}
    print(f"\nO conjunto foi recriado e agora contém \n{conjunto}")
    conjunto.remove("Mega Drive")
    print(f"Ao utilizar o método remove() com o item Mega Drive, o conjunto ficou assim \n{conjunto}, mas retornaria um erro caso o item não estivesse no conjunto")

    conjunto = {"Mega Drive", "Super Nintendo","Playstation", "Nintendo 64", "Sega Saturn", "Dreamcast"}
    print(f"\nO conjunto foi recriado e agora contém \n{conjunto}")
    conjunto.discard("Mega Drive")
    print(f"Ao utilizar o método discard() com o item Mega Drive o conjunto ficou assim \n{conjunto}, e não retornaria um erro caso o item não estivesse no conjunto")
```

Código-fonte 17 – Métodos . .clear(), .copy(), .pop(), .remove() e .discard() do set
Fonte: Elaborado pelo autor (2022)

Agora você já sabe: quando precisar de uma estrutura dinâmica que suporte repetições, deve usar a list; quando precisar que ela não suporte repetições de elementos, é melhor usar o set; e, por fim, quando precisar de uma estrutura imutável, a tuple será a solução. Mas e quando precisarmos associar os nossos dados a determinados valores? É aí que entra o dicionário!

1.5 Olha aí no dicionário!

Quando estávamos na escola era frequente o pedido das professoras e professores para que buscássemos uma palavra que ainda não estava no nosso vocabulário dentro do dicionário, e esse ato tão simples revela algo importante sobre a estrutura do Python que possui o mesmo nome.

O dicionário de língua portuguesa funciona de maneira simples: existem apenas palavras únicas e com base nessas palavras encontraremos um ou mais significados.

Na programação, esse mecanismo é chamado de *chave-valor*, onde a chave é um identificador único e o valor é o dado que estamos associando a essa chave.

Esse conceito é tão poderoso que é utilizado para trafegar informações pela web (com o formato JSON, por exemplo), para criar bancos de dados não-relacionais (como o Amazon DynamoDB) e também em estruturas de dados, como é o caso do *dict* do Python.

A sintaxe da estrutura de um dicionário é a seguinte:

```
dicionario = {
    chave1:valor1,
    chave2:valor2,
    chave3:valor3
}
```

Código-fonte 18 – Sintaxe do dicionário em Python
Fonte: Elaborado pelo autor (2022)

Uma das grandes vantagens dessa estrutura é que podemos recuperar um determinado valor sem ter que percorrer o dicionário inteiro, bastando saber a sua chave. Para compreendermos melhor, vamos criar um dicionário com alguns elementos e exibi-lo:

```
dicionario = {
    "nome": "Star Wars - Episódio IV - Uma nova esperança",
    "diretor": "George Lucas",
    "lançamento": 1977,
    "bilheteria": 775000000.00
}

print(dicionario)
print(f"O valor da chave nome é {dicionario['nome']}")
```

Código-fonte 19 – Criação e exibição de um dicionário
Fonte: Elaborado pelo autor (2022)

O exemplo anterior nos revela algumas informações

- O dicionário é uma estrutura heterogênea, ou seja, suporta valores de diferentes tipos.
- Se printarmos o dicionário, veremos a estrutura de forma muito semelhante ao que vemos no código.

- É possível acessar o valor de uma chave indicando essa chave entre colchetes ao lado do nome do dicionário.

Se quisemos incluir uma nova chave e um novo valor no dicionário, podemos utilizar uma notação semelhante a recuperar valores. Basta indicarmos a nova chave entre colchetes ao lado do nome do dicionário, seguido do sinal de igual e do novo valor. Veja:

```
    dicionario = {
        "nome": "Star Wars - Episódio IV - Uma nova
esperança",
        "diretor": "George Lucas",
        "lançamento": 1977,
        "bilheteria": 775000000.00

    }

    print(f"O dicionário completo é {dicionario}")
    print(f"O valor da chave nome é {dicionario['nome']}")

    dicionario["gênero"] = "Space opera"
    print(f"O dicionário completo é {dicionario}")
```

Código-fonte 20 – Criação de nova chave no dicionário
Fonte: Elaborado pelo autor (2022)

Por mais que seja simples fazer a manipulação do dicionário através dos colchetes, a notação pode gerar alguns problemas. Veja o que ocorre quando tentamos visualizar uma chave que não está contida no dicionário:

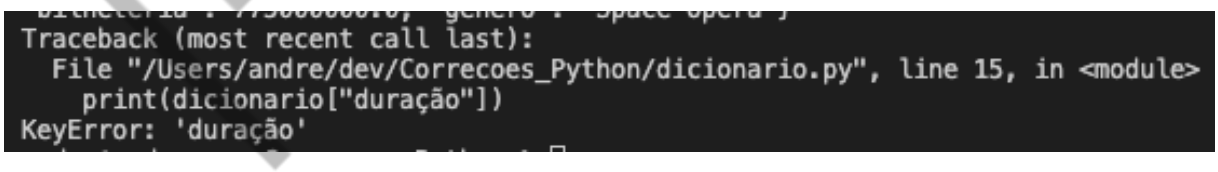

Figura 1 – Exceção causada ao tentar acessar uma chave inexistente
Fonte: Elaborado pelo autor (2022)

Para a nossa sorte, assim como ocorria na lista, o dicionário é repleto de métodos que facilitam a manipulação dos seus dados. Os principais são:

Método	Efeito
dicionario.get(chave)	Retorna o valor associado a uma chave específica.
dicionario.values()	Retorna uma lista com os valores do dicionário.
dicionario.keys()	Retorna uma lista de chaves do dicionário.
dicionario.items()	Retorna uma lista de tuplas contendo chave e valor.
dict.fromkeys(chaves, valor)	Retorna um dicionário com base nas chaves especificadas. Pode ou não atribuir um valor fornecido.
dicionario.setdefault(chave, valor)	Retorna o valor de uma determinada chave. Se a chave não existir, é criada com valor None ou o valor informado.
dicionario.update(chave, valor)	Atualiza o dicionário com base na chave e no valor fornecidos.
dicionario.copy()	Retorna uma cópia do dicionário.
dicionario.pop(chave)	Remove um item do dicionário com base na chave.
dicionario.popitem()	Remove a última chave e valor inserida no dicionário.
dicionario.clear()	Remove todos os elementos do dicionário.

Quadro 3 – Métodos das tuplas em Python
Fonte: Elaborado pelo autor (2022)

Como são muitos métodos, vamos criar exemplos separados para compreendê-los por grupos, sendo os primeiros: .values(), .keys() e .items() os métodos que retornam estruturas iteráveis.

```
    dicionario = {
        "nome": "Star Wars - Episódio IV - Uma nova esperança",
        "diretor": "George Lucas",
        "lançamento": 1977,
        "bilheteria": 775000000.00

    }

    print(f"O método .keys() retorna \n{dicionario.keys()}")
    print("Se percorrermos a lista retornada com um loop for teremos: ")
    for chave in dicionario.keys():
        print(chave)

    print(f"\nO método .values() retorna \n{dicionario.values()}")
    print("Se percorrermos a lista retornada com um loop for teremos: ")
```

```
    for valor in dicionario.values():
        print(valor)

    print(f"\nO método .items() retorna \n{dicionario.items()}")
    print("Como foi retornada uma lista de tuplas e as tuplas podem ser desempacotadas, teremos: ")
    for chave, valor in dicionario.items():
        print(f"{chave} -- {valor}")
```

Código-fonte 21 – Testando os métodos .keys(), .values() e .items()
Fonte: Elaborado pelo autor (2022)

Viu como foi bom ter estudado as tuplas e as listas antes? O dicionário nos retorna alguns dados no formato dessas estruturas e podemos aplicar as mesmas técnicas de iteração que utilizávamos com elas.

Agora é hora de testarmos os métodos que retornam valores, como .get() e .setdefault(), e que retornam um dicionário inteiro, como o .copy():

```
    dicionario = {
        "nome": "Star Wars - Episódio IV - Uma nova esperança",
        "diretor": "George Lucas",
        "lançamento": 1977,
        "bilheteria": 775000000.00

    }

    print(f"O método .get() para a chave diretor retorna \n{dicionario.get('diretor')}")
    print(f"O método .get() para a chave publico (que não existe) retorna \n{dicionario.get('publico')}")

    print(f"\nO método .setdefault() para a chave diretor retorna \n{dicionario.setdefault('diretor')}")
    dicionario.setdefault("publico")
    print(f"O método .setdefault() para a chave publico (que não existe) cria a chave e coloca o valor None. Veja como ficou nosso dicionário depois de utilizar este método: \n{dicionario}")
    dicionario.setdefault("custo", 11000000.0)
    print(f"Caso utilizemos o método .setdefault() para a chave custo (que não existe) e também passarmos um valor como argumento, a chave será criada com este valor. Veja como ficou nosso dicionário depois de utilizar este método: \n{dicionario}")
```

```
    print(f"\nO método .copy() retorna a seguinte cópia do dicionário que, se for alterada, não impacta no dicionário original: {dicionario.copy()}")
```

Código-fonte 22 – Testando os métodos .get(), .setdefault() e .copy()
Fonte: Elaborado pelo autor (2022)

É possível perceber que o método .setdefault() é muito útil para as situações em que precisamos verificar o valor de uma chave e criá-la caso não exista.

Quando a nossa intenção, porém, é apenas a de modificar o dicionário, podemos utilizar os métodos .update(), .pop(), .popitem() e .clear():

```
    dicionario = {
        "nome": "Star Wars - Episódio IV - Uma nova esperança",
        "diretor": "George Lucas",
        "lançamento": 1977,
        "bilheteria": 775000000.00

    }

    dicionario.update({"custo": 50.0})
    print(f"O método .update() com a chave custo (que não existia) e o valor 50.0 nos deixa com o seguinte dicionário \n{dicionario}")
    dicionario.update({"custo": 11000000.0})
    print(f"O método .update() com a chave custo (que já existe após a mudança anterior) e o valor 11000000.0 nos deixa com o seguinte dicionário \n{dicionario}")


    dicionario.pop("diretor")
    print(f"\nO método .pop() para a chave diretor nos deixa com o seguinte dicionário \n{dicionario}")

    dicionario.popitem()
    print(f"\nO método .popitem() nos deixa com o seguinte dicionário \n{dicionario}")

    dicionario.clear()
    print(f"\nO método .clear() nos deixa com o seguinte dicionário \n{dicionario}")
```

Código-fonte 23 – Testando os métodos .update(), .pop(), .popitem() e .clear()
Fonte: Elaborado pelo autor (2022)

O único método do qual ainda não falamos é o método .fromkeys(), que é usado para gerar um dicionário com um conjunto de chaves e o mesmo valor para todas. Esse método não é usado para extrair dados de um dicionário existente, mas, sim, para gerar um novo. Veja:

```
    notas = dict.fromkeys(("Matemática", "Física", "Química"))

    print(f"O dicionário gerado com o método fromkeys para a tupla de chaves (Matemática, Física, Química) e sem valor foi: \n{notas}")

    notas = dict.fromkeys(("Matemática", "Física", "Química"), [])

    print(f"\nO dicionário gerado com o método fromkeys para a tupla de chaves (Matemática, Física, Química) e valor zero foi: \n{notas}")
```

Código-fonte 24 – Testando o método .fromkeys()
Fonte: Elaborado pelo autor (2022)

O dicionário é uma estrutura tão poderosa e repleta de possibilidades que vale a pena criarmos um script pensando em um caso real. Vamos criar uma ficha de cadastro que permita ao usuário informar os campos que deseja incluir e os valores para esses campos. Para facilitar o uso, criaremos uma estrutura de menus com o loop while:

```
    op = 0
    ficha = {} #criação do dicionário vazio
    while op != 4:
        print("\nFICHA CADASTRAL")
        print("1 - Incluir informações na ficha")
        print("2 - Recuperar informação da ficha")
        print("3 - Exibir a ficha completa ")
        print("4 - Sair")
        op = int(input("Informe a opção desejada: "))

        if op == 1:
            chave = input("\nEm qual campo deseja inserir dados? ")
            valor = input(f"Qual é o dado que deseja incluir no campo {chave}? ")

            ficha.update({chave:valor})
            #ficha[chave] = valor #esta notação também é muito utilizada e gera o mesmo efeito da linha acima
```

```
        elif op == 2:
            print(f"\nOs campos disponíveis na ficha são:
{ficha.keys()}")
            chave = input("Você deseja obter dados de qual
campo? ")
            if chave in ficha.keys():
                print(f"{chave} -> {ficha.get(chave)}")
                #print(f"{chave} -> {ficha[chave]}") #como o
if já garante que a chave existe, poderíamos utilizar esta
notação sem correr riscos
            else:
                print("O campo informado não existe na ficha
cadastral.")
        elif op == 3:
            print("\n---FICHA---")
            for campo, dado in ficha.items():
                print(f"{campo.upper()} -> {dado}")
            print("---------------------------")
        elif op == 4:
            print("Saindo do sistema de ficha cadastral")
            break
        else:
            print("Por favor, escolha uma opção válida")
```

Código-fonte 25 – Programa de ficha cadastral
Fonte: Elaborado pelo autor (2022)

Já notou que o dicionário foi uma ótima adição ao nosso cinto de utilidades, certo? Então aperte esse mesmo cinto, porque vamos entrar no mundo das *collections*!

1.6 O mundo das collections!

Além dos *containers* que fazem parte das estruturas nativas do Python, como tuple, list, dict e set, existem outros que estão presentes em módulos adicionais que estão voltados para usos mais específicos. O famoso pacote NumPy ou o Pandas, por exemplo, possuem conteiners específicos para computação científica e manipulação de dados.

Existe um módulo nativo do Python que se chama *collections* e, dentro dele, encontraremos algumas estruturas baseadas naquelas que já aprendemos neste capítulo (e que estudaremos agora): *defaultdict*, ordereddict, namedtuple e deque! Vamos a elas!

1.6.1 Defaultdict

O *defaultdict* é uma sub-classe do dicionário que nós já conhecemos. Sem entrar em detalhes do paradigma de programação orientado a objetos, basta saber que isto significa que o defaultdict tem praticamente as mesmas funcionalidades do dicionário tradicional.

E grande diferença entre o dict e o defaultdict é que a estrutura que vamos aprender agora não gera exceções quando uma chave não é encontrada. Ao invés disso ela sempre atribui um valor padrão para esses casos.

Para criar um defaultdict é preciso importá-lo do módulo collections e, depois, indicar quais serão os valores padrão assumidos. Veja o quadro a seguir:

```
    from collections import defaultdict

    dicionario = defaultdict(gerador do valor padrão)
```

Código-fonte 26 – Criação do defaultdict
Fonte: Elaborado pelo autor (2022)

A grande questão é que os valores padrão para o defaultdict costumam vir de funções (que estudaremos no próximo capítulo) ou de funções lambda (tópico avançado de Python). Por essa razão, criaremos o nosso defaultdict utilizando apenas uma lista, uma função de exemplo (que compreenderemos melhor quando estudarmos funções) e uma função lambda (cujo funcionamento não exploraremos neste curso básico).

O código a seguir reúne demonstrações de uso do defaultdict:

```
    #importação do defaultdict
    from collections import defaultdict

    #criação de um default dict com uma lista como valor
padrão
    dicionario_lista = defaultdict(list)
    dicionario_lista["PRODUTO"] = "Macbook Pro"
    dicionario_lista["MARCA"] = "Apple"
    print(f"Exibindo a chave PRODUTO do dicionario criado com
uma lista: {dicionario_lista['PRODUTO']}")
    print(f"Exibindo a chave PREÇO, que não existe no
dicionario criado com uma lista: {dicionario_lista['PREÇO']}")
```

```
    #Criação de função que retorna a frase "INEXISTENTE"
    def funcao_exemplo():
        return "INEXISTENTE"
    dicionario_funcao = defaultdict(funcao_exemplo)
    dicionario_funcao["PRODUTO"] = "Macbook Pro"
    dicionario_funcao["MARCA"] = "Apple"
    print(f"\nExibindo a chave PRODUTO do dicionario criado com uma função: {dicionario_funcao['PRODUTO']}")
    print(f"Exibindo a chave PREÇO, que não existe no dicionario criado com uma função: {dicionario_funcao['PREÇO']}")

    #Criação de dicionário com uma função lambda
    dicionario_lambda = defaultdict(lambda: "Não disponível")
    dicionario_lambda["PRODUTO"] = "Macbook Pro"
    dicionario_lambda["MARCA"] = "Apple"
    print(f"\nExibindo a chave PRODUTO do dicionario criado com uma função lamda: {dicionario_lambda['PRODUTO']}")
    print(f"Exibindo a chave PREÇO, que não existe no dicionario criado com uma função lambda: {dicionario_lambda['PREÇO']}")
```

Código-fonte 27 – Exemplos do defaultdict
Fonte: Elaborado pelo autor (2022)

Os métodos disponíveis para o defaultdict são os mesmos do dicionário padrão e, portanto, não há necessidade de retomá-los aqui.

1.6.2 OrderedDict

Assim como ocorreu com o *defaultdict*, o *OrderedDict* também é uma sub-classe do dicionário padrão e, portanto, conta com as mesmas facilidades e métodos do original.

A grande diferença aqui é que o *OrderedDict* se recorda da ordem em que as chaves foram inseridas mesmo que seu valor seja alterado. O único caso em que uma chave mudará de ordem é se for excluída e depois reinserida. Veja o exemplo a seguir:

```
    #importação do OrderedDict
    from collections import OrderedDict

    dicionario_ordenado = OrderedDict()

    print(f"O dicionario foi criado e ainda não contém nenhum
valor: \n{OrderedDict}. Adicionaremos os seguintes valores e
chaves: Nome:Iphone, Marca:Apple, Modelo:14 Pro Max")

    #Adicionando chaves e valores
    dicionario_ordenado["NOME"] = "Iphone"
    dicionario_ordenado["MARCA"] = "Apple"
    dicionario_ordenado["MODELO"] = "14 Pro Max"

    print("\nPercorrendo   o   dicionario   verificamos   as
seguintes chaves e valores: ")

    for chave, valor in dicionario_ordenado.items():
        print(f"{chave} --- {valor}")

    dicionario_ordenado["MARCA"] = "Maçã"

    print("\nAo alterar o valor da chave MARCA, percebemos
que a ordem se mantém ")

    for chave, valor in dicionario_ordenado.items():
        print(f"{chave} --- {valor}")

    dicionario_ordenado.pop("MARCA")

    print("\nAo removermos a chave MARCA e percorrermos o
dicionario, obtemos: ")

    for chave, valor in dicionario_ordenado.items():
        print(f"{chave} --- {valor}")

    dicionario_ordenado["MARCA"] = "Apple"
    print("\nAo reinserir a chave MARCA, notamos que passou a
ser colocada na última posição ")

    for chave, valor in dicionario_ordenado.items():
        print(f"{chave} --- {valor}")
```

Código-fonte 28 – Exemplos do OrderedDict
Fonte: Elaborado pelo autor (2022)

Mais uma vez não é necessário explorar seus métodos, uma vez que são os mesmos do *dict* e do *defaultdict.*

1.6.3 Namedtuple

Enquanto os itens presentes dentro de uma *tuple* tradicional podem ser acessados através dos seus índices numéricos, a *namedtuple* permite que criemos nomes para cada um dos dados que vamos inserir dentro da estrutura.

A *namedtuple* pode ser criada com a seguinte sintaxe:

```
    nome_tipo = namedtuple(nome_do_tipo, _[nomes_dos_campos])
    novo_objeto = nome_tipo(valor_campo_1, valor_campo_2,
...)
```

Código-fonte 29 – Sintaxe da criação da namedtuple
Fonte: Elaborado pelo autor (2022)

Como é uma sintaxe de abordagem diferente das estruturas com as quais vínhamos trabalhando, vamos entender essa estrutura por etapas. Primeiro vamos criar e exibir uma *namedtuple* para produtos:

```
    from collections import namedtuple
    Produto = namedtuple("Produto", ["nome", "marca",
"preco"]) #Criação do novo tipo

    novo_produto = Produto("Ipad", "Apple", 2499.99)
    print(f"Criamos o objeto chamado novo_produto, usando
como tipo Produto. Ao exibirmos este objeto temos:
\n{novo_produto}")
```

Código-fonte 30 – Criação de um objeto com a namedtuple
Fonte: Elaborado pelo autor (2022)

O que nós fizemos no exemplo anterior foi criar um novo tipo chamado Produto. Esse novo tipo tem *campos* para nome, marca e preço. Ao criar um novo objeto que seja do tipo Produto, fornecemos valores para esses três campos.

Agora que criamos o novo objeto, que é do tipo Produto, percebemos que se parece um pouco com o dicionário, pois cada campo (que age como uma chave) possui o seu respectivo valor. Apesar disso, ele ainda é uma *tuple* e podemos nos beneficiar disso, como iterá-lo sem a necessidade de nenhum método especial ou desempacotar valores. Veja:

```
    from collections import namedtuple
    Produto = namedtuple("Produto", ["nome", "marca",
"preco"]) #Criação do novo tipo

    novo_produto = Produto("Ipad", "Apple", "2499.99")
    print(f"Criamos o objeto chamado novo_produto, usando
como tipo Produto. Ao exibirmos este objeto temos:
\n{novo_produto}")

    #iteração
    print("\nComo a namedtuple ainda é uma tuple, podemos
iterá-la: ")
    for valor in novo_produto:
        print(valor)

    #desempacotamento
    x, y, z = novo_produto
    print(f"\nTambém foi possível desempacotar os valores de
novo_produto nas variáveis X={x}, Y={y} e Z={z}")
```

Código-fonte 31 – iteração e desempacotamento da namedtuple
Fonte: Elaborado pelo autor (2022)

1.6.4 Deque

Se conhecemos duas novas estruturas baseadas em dicionário e uma baseada em tupla, é claro que não poderíamos encerrar sem conhecer uma nova estrutura baseada em lista!

A deque é uma estrutura que cumpre o mesmo objetivo das lists, mas com uma grande diferença: o desempenho! Enquanto o desempenho das operações de append e pop na lista piora de acordo com a quantidade de elementos, na deque permanece sempre constante.

A criação de uma deque é muito simples e atende à seguinte sintaxe:

```
    from collections import deque

    nova_deque = deque([elemento1, elemento2, elemento3])
```

Código-fonte 32 – Sintaxe de criação da deque
Fonte: Elaborado pelo autor (2022)

Seu uso também é idêntico ao da lista, com a adição de alguns outros métodos:

Método	Efeito
deque.appendleft(elemento)	Inclui um elemento no início da deque.
deque.popleft(elemento)	Remove um elemento do início da deque.
deque.extend(estrutura_iteravel)	Inclui, no fim da deque, os elementos de uma estrutura iterável.
deque.extendleft(estrutura_iteravel)	Inclui, no início do deque, os elementos de uma estrutura iterável.
deque.rotate(valor)	Movimenta os elementos da deque, de acordo com o valor passado como argumento.

Quadro 4 – Métodos da deque em Python
Fonte: Elaborado pelo autor (2022)

1.6 Encerrando com o mapa da mina

Ao longo deste capítulo, você aprendeu a lidar com uma série de estruturas que armazenam conjuntos de dados e todas elas eram iteráveis ou tinham métodos que retornavam uma estrutura iterável. É chegada a hora de aproveitarmos melhor este recurso através do map.

A função map permite executar uma mesma função para todos os elementos de uma estrutura iterável. Sua sintaxe é:

```
map(função, espaço_iteravel)
```

Código-fonte 33 – Sintaxe da função map
Fonte: Elaborado pelo autor (2022)

Como ainda não estudamos a criação de funções, vamos usar um exemplo simples: lembra-se da função len()? Ela é uma função nativa do Python que retorna o tamanho de uma estrutura (ou, no caso das strings, o tamanho do texto). Nós vamos aplicar a função len em uma lista de strings, utilizando o map para fazer isso.

Antes de exibirmos o resultado do nosso mapa, precisamos convertê-lo para lista para garantir que possa ser lido. Veja:

```
    lista = ["Python", "Java", "Fortran", "Cobol", "C++"]
    mapa = map(len, lista)
    print(f"Criamos a seguinte lista: \n{lista}")
    print(f"Se tentarmos exibir o mapa que foi gerado
aplicando a função len à lista, teremos apenas seu endereço de
```

```
memória: \n{mapa}")
    lista_do_mapa = list(mapa)
    print(f"Convertendo o mapa para o tipo list, obtemos o resultado da aplicação da função len a cada uma das palavras da lista original: \n {lista_do_mapa}")

    print(f"\nApesar de termos criado objetos para armazenarem o mapa e a lista convertida, o mesmo efeito poderia ser atingido escrevendo em um print list(map(len, lista)): \n{list(map(len, lista))}")
```

Código-fonte 34 – Sintaxe da função map
Fonte: Elaborado pelo autor (2022)

Este exemplo, apesar de simples, nos mostra o potencial que a função list apresenta para quando pudermos aplicar outras funções às estruturas que conhecemos. Por isso, prepare-se para entrar no incrível mundo da criação de funções em Python!

REFERÊNCIAS

LUTZ, M. **Learning Python.** Sebastopol: O'Reilly Media Inc, 2013.

PUGA, S.; RISSETI, G. Lógica de programação e estrutura de dados com aplicações em Java. São Paulo: Pearson Prentice Hall, 2009.