FIAP ON

MODELING

SALVO PELO DICIONÁRIO

ANDRÉ DE FREITAS DAVID

08

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

1 SALVO PELO DICIONÁRIO

1.1 Cada macaco no seu galho!

Quanto mais avançamos no estudo dos conceitos básicos de programação, mais percebemos o poder que ganhamos quando aprendemos a manipular dados corretamente.

O poder dos nossos computadores modernos, com muitos gigabytes de memória RAM e processadores ultrarrápidos, porém, pode levar o programador a algumas escolhas preguiçosas e perigosas.

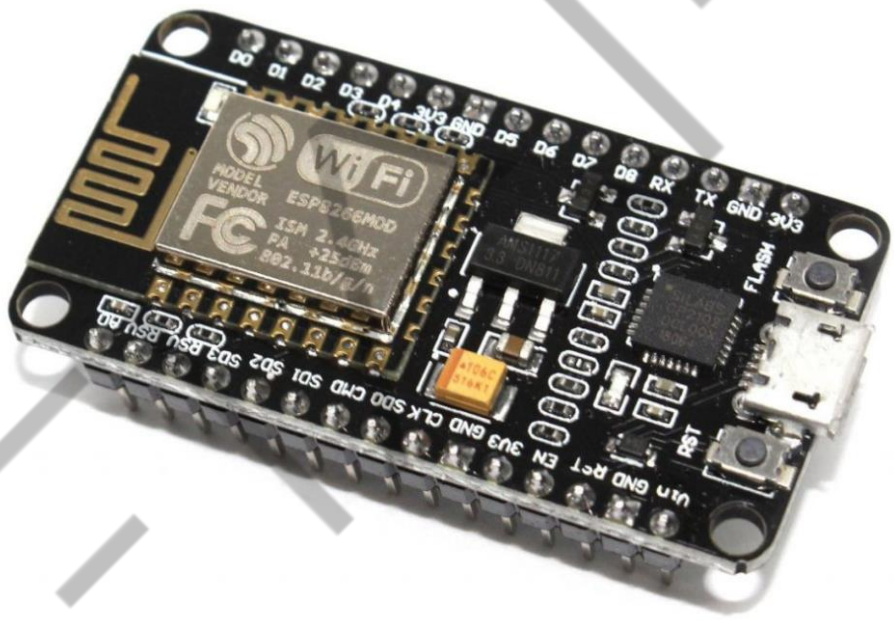

Figura 1 – Placa NodeMCU ESP-8266, que suporta micropython e tem cerca de 12kb de memória livre para variáveis
Fonte: Wikipedia (2020)

Será que as estruturas de dados que aprendemos até agora (variáveis e listas) são o suficiente para criar os *melhores* programas em todos os cenários? Ou será que o desenvolvedor de um software que vai ser executado em uma Raspberry PI ou no rádio de um carro deve ter alguma preocupação especial?

Neste capítulo, descobriremos que ainda existe um vasto mundo para desvendarmos e que o que vai diferenciar um desenvolvedor de ponta de um desenvolvedor “gastão” é lembrar para que serve cada estrutura.

Avise seus amigos e coloque um recado na sua rede social: os dias de confiar apenas nas listas acabaram!

1.2 Por que as listas não são o suficiente?

Geralmente, quando aprendemos uma nova estrutura de programação, nos perguntamos: “Ué, mas por que eu não posso continuar usando a mesma estrutura que eu já conheço?”.

Durante nosso estudo de estruturas para conjuntos de dados, você vai perceber que, com mais ou menos trabalho, tudo poderia ser resolvido com listas. Mas será que isso é adequado? Quanto espaço uma lista ocupa?

Para responder a essas perguntas, vamos fazer um pequeno teste: usando o módulo *sys* do Python (que é instalado por padrão), poderemos verificar o tamanho de qualquer objeto por meio do método *getsizeof()*. Dê uma olhada no script de exemplo, no Código-fonte “Exemplo de uso do método getsizeof() do módulo sys”, que cria uma variável de texto, uma variável inteira e outra para números decimais, exibindo seus tamanhos:

```
     #primeiro importamos o módulo sys
     import sys

     #depois criamos algumas variáveis de exemplo
     nome = "Bruce Wayne"
     idade = 30
     peso = 92.3

     #Vamos exibir em uma mensagem o nome da variável, o tipo
(type) e o tamanho (getsizeof)
     print("A variável nome é do tipo {} e tem {}
bytes".format(type(nome), sys.getsizeof(nome)))
     print("A variável idade é do tipo {} e tem {}
bytes".format(type(idade), sys.getsizeof(idade)))
     print("A variável peso é do tipo {} e tem {}
bytes".format(type(peso), sys.getsizeof(peso)))
```

Código-fonte 1 – Exemplo de uso do método getsizeof() do módulo sys
Fonte: Elaborado pelo autor (2020)

Esse exemplo deixa claro como usar o método *getsizeof* para verificar o tamanho de um objeto. Vamos, agora, aplicar a mesma lógica utilizando uma lista?

```
      #primeiro importamos o módulo sys
      import sys

      #agora vamos criar uma lista vazia
      lista = []

      #E verificar o tamanho
      print("O objeto lista é do tipo {} e tem {}
bytes".format(type(lista), sys.getsizeof(lista)))
```

Código-fonte 2 – Verificando o tamanho de uma lista com o método *getsizeof()*
Fonte: Elaborado pelo autor (2020)

O resultado mostrado na tela é que uma lista vazia ocupa 56 bytes na memória RAM do computador. Pode parecer pouco, mas lembre-se: essa lista está vazia!

Então, por que ela ocupa tanto espaço? A resposta é muito simples, e, se juntarmos algumas evidências, podemos chegar a ela rapidamente.

Manipulando uma lista, nós podemos incluir novos elementos a qualquer momento, reordenar a lista e remover qualquer elemento, certo? Todas essas possibilidades têm um custo que se traduz em espaço em memória RAM.

Ei, não precisa se desesperar. As listas ocupam espaço, mas seu uso vale muito a pena, sempre que quisermos usar todos os recursos que elas oferecem. Só que, em todas as situações nas quais não queremos usar esses recursos, as listas são apenas um custo a mais no nosso projeto.

É por isso que existem outras estruturas mais adequadas do que a lista, cada uma voltada para uma situação específica.

Vamos estudar a primeira delas?

1.3 Trabalho em tupla

Passado o trauma inicial de perceber que o professor escreveu “tupla” em vez de “dupla”, vem um segundo trauma: isso foi intencional!

As tuplas ou tuples são estruturas da linguagem Python parecidíssimas com as listas, mas com a característica especial de serem imutáveis. Isso quer dizer que todos os dados que armazenarmos dentro de uma tupla permanecerão idênticos e na mesma posição ao longo de toda a execução do programa.

Figura 2 – Enquanto as listas parecem com o Sr. Fantástico e podem se esticar, as tuplas são como o Coisa e permanecem estáticas
Fonte: Marvel (2020)

Para termos uma ideia do quão parecidas as tuplas e as listas são, a criação das duas estruturas só difere no símbolo usado. Enquanto as listas são criadas **com colchetes**, as tuplas são criadas **com parênteses**.

O script no Código-fonte “Criação de tupla em linguagem Python” cria uma tupla com as categorias dos guerreiros Jedi no universo de Star Wars.

```
#criação de uma tupla com as categorias dos Jedi
categorias = ("Youngling", "Padawan", "Knight", "Master")
```

Código-fonte 3 – Criação de tupla em linguagem Python
Fonte: Elaborado pelo autor (2020)

Se pararmos para pensar um pouco, faz todo sentido que armazenemos esses dados em uma tupla em vez de em uma lista. Afinal de contas, essas categorias não mudarão durante a execução do programa. Se estivéssemos falando de *nomes* de membros da ordem Jedi, armazenar em uma lista faria mais sentido, pois esses dados são mutáveis.

Passada a nossa reflexão em uma galáxia muito, muito distante, vamos explorar e exibição de todos os itens da tupla, que pode ser feita da mesma forma que em uma lista.

```
    #criação de uma tupla com as categorias dos Jedi
    categorias = ("Youngling", "Padawan", "Knight", "Master")

    #exibindo a tupla inteira
    print(categorias)
```

Código-fonte 4 – Exibição da tupla em linguagem Python
Fonte: Elaborado pelo autor (2020)

Lembra que os índices são aqueles números inteiros, iniciando em 0, que marcam a posição dos itens dentro de uma lista? Eles também podem ser usados nas tuplas, seguindo exatamente a mesma lógica que utilizávamos nas listas:

```
    #criação de uma tupla com as categorias dos Jedi
    categorias = ("Youngling", "Padawan", "Knight", "Master")

    #exibindo a tupla inteira -> ('Youngling', 'Padawan',
'Knight', 'Master')
    print(categorias)

    #exibindo o segundo item da tupla -> Padawan
    print(categorias[1])

    #usando um índice negativo para exibir o último item da
tupla -> Master
    print(categorias[-1])
```

Código-fonte 5 – Exibição da tupla em linguagem Python
Fonte: Elaborado pelo autor (2020)

“Só falta agora dizer que podemos usar os loops para exibir cada um dos itens da lis... digo, *tupla*, professor?!”. Pois é, você já é um programador tão avançado que está certinho por ter pensado nisso. Observe o Código-fonte “Exibição dos itens da tupla usando loop”.

```
    #criação de uma tupla com as categorias dos Jedi
    categorias = ("Youngling", "Padawan", "Knight", "Master")

    #exibindo cada item da tupla usando um loop
    for categoria in categorias:
        print(categoria)
```

Código-fonte 6 – Exibição dos itens da tupla usando loop
Fonte: Elaborado pelo autor (2020)

Então, qual é a grande vantagem de usar tuplas (ou listas), se a única diferença entre elas é a *mutabilidade* do conteúdo? A vantagem é exatamente essa.

Como a tupla é imutável, ela ocupa menos espaço na memória RAM, portanto, *custa* menos processamento. Vamos provar isso usando o bom e velho *getsizeof*?

No script do Código-fonte “Script para exibir o tamanho em bytes de uma lista e uma tupla”, criaremos uma lista vazia e uma tupla vazia, exibindo seus tamanhos.

```
     #importando o módulo sys para conseguirmos usar o
getsizeof
     import sys

     #criando uma lista e uma tupla vazias, respectivamente
     lista_vazia = []
     tupla_vazia = ()

     #printando o tipo e tamanho de cada estrutura
     print("O objeto lista_vazia é do tipo {} e ocupa {} bytes
na                          memória".format(type(lista_vazia),
sys.getsizeof(lista_vazia)))
     print("O objeto tupla_vazia é do tipo {} e ocupa {} bytes
na                          memória".format(type(tupla_vazia),
sys.getsizeof(tupla_vazia)))
```

Código-fonte 7 – Script para exibir o tamanho em bytes de uma lista e uma tupla
Fonte: Elaborado pelo autor (2020)

Se você executou o script em uma máquina rodando um sistema operacional de 64bits, deve ter obtido como resultado que a lista ocupa *56 bytes*, enquanto a tupla ocupa *40 bytes.*

Parece uma diferença muito pequena quando pensamos em um computador com 8gb de memória RAM, mas, e quando transportamos essa realidade para um *microcontrolador*?

Se tomarmos como exemplo a placa NodeMCU ESP-8266, que é programável em micropython, temos cerca de 12kb de memória disponível para as variáveis. Definitivamente, não podemos esbanjar, como fazemos com nosso desktop, não é?

Se a economia de espaço e processamento não for argumento suficiente para convencer você a analisar bem quando é prudente substituir uma lista por uma tupla, podemos pensar em termos de boas práticas de programação.

1.3.1 Quando a tupla é uma boa prática

Imagine que você esteja fazendo um software para o DETRAN do seu estado. Com você é um estudante atento e já aprendeu a modularizar seus programas, cria um módulo apenas para lidar com as carteiras de habilitação.

```
    #método para validar a categoria informada
    def validar_categoria(categoria_usuario):
        #verificando se a categoria digitada pelo usuário está
presente na lista
        if categoria_usuario.upper() in categorias:
            #se estiver, é exibida essa mensagem
            print("Categoria válida!")
        else:
            #se não estiver, é exibida essa mensagem
            print("Categoria inválida!")

    #lista com categorias de habilitação do DETRAN
    categorias=["A", "B", "C", "D", "E"]
```

Código-fonte 8 – Módulo “habilitacoes” que usa uma list com um método que valida a categoria
Fonte: Elaborado pelo autor (2020)

Ao fazer isso, você disponibiliza o seu módulo para que outros programadores da equipe utilizem. Sabemos que a utilização correta do módulo seria parecida com o Código-fonte “Script para utilizar o módulo anterior para validar uma categoria”.

```
    #importando o módulo que lida com as habilitacoes
    import habilitacoes

    #pedindo que o usuário a categoria
    categoria_digitada = input("Digite a categoria de
habilitação")

    #utilizando o método validar_categoria para verificar se
o que foi digitado é válido
    habilitacoes.validar_categoria(categoria_digitada)
```

Código-fonte 9 – Script para utilizar o módulo anterior para validar uma categoria
Fonte: Elaborado pelo autor (2020)

Parece que tudo está perfeito no cenário que criamos, não é verdade? Afinal de contas, nosso programa está modularizado e consegue validar corretamente as categorias A, B, C, D e E.

O que o nosso cenário imaginário esconde, porém, é que um programador mal-intencionado pode alterar a lista de categorias sem sequer abrir o código do módulo “habilitações”. Veja o Código-fonte “Sript para utilizar o módulo ‘habilitacoes’ e incluir uma categoria falsa em tempo de execução”.

```
    #importando o módulo que lida com as habilitacoes
    import habilitacoes

    #pedindo que o usuário a categoria
    categoria_digitada = input("Digite a categoria de
habilitação")
```

```
    #incluindo uma nova categoria falsa em tempo de execução
    habilitacoes.categorias.append("ESPECIAL")

    #utilizando o método validar_categoria para verificar se
o que foi digitado é válido
    habilitacoes.validar_categoria(categoria_digitada)
```

Código-fonte 10 – Script para utilizar o módulo “habilitacoes” e incluir uma categoria falsa em tempo de execução
Fonte: Elaborado pelo autor (2020)

Com a *malandragem* do código anterior, o usuário poderá digitar a categoria falsa “ESPECIAL” e validá-la. Afinal de contas, o objeto *categorias* presente dentro do módulo *habilitacoes* é do tipo *list*, e as listas são mutáveis.

Tudo isso pode ser prevenido utilizando uma tupla. Vamos ver, no Código-fonte “Módulo ‘habilitações’ que usa uma tupla com um método que valida a categoria”, como ficará o código do nosso módulo utilizando uma tupla e não uma lista.

```
    #importando o módulo que lida com as habilitacoes
    import habilitacoes

    #pedindo que o usuário a categoria
    categoria_digitada = input("Digite a categoria de
habilitação")

    #incluindo uma nova categoria falsa em tempo de execução
    habilitacoes.categorias.append("ESPECIAL")

    #utilizando o método validar_categoria para verificar se
o que foi digitado é válido
    habilitacoes.validar_categoria(categoria_digitada)
```

Código-fonte 11 – Módulo “habilitacoes” que usa uma tupla com um método que valida a categoria
Fonte: Elaborado pelo autor (2020)

Com essa simples alteração, se o programador tentar incluir a categoria falsa na hora de utilizar o módulo, obterá o seguinte erro indicado na Figura “Erro ao tentar incluir uma categoria falsa dentro de uma tupla”.

```
Traceback (most recent call last):
  File "C:/Users/andre/PycharmProjects/capitulos/detran.py", line 8, in <module>
    habilitacoes.categorias.append("ESPECIAL")
AttributeError: 'tuple' object has no attribute 'append'
```

Figura 3 – Erro ao tentar incluir uma categoria falsa dentro de uma tupla
Fonte: Elaborado pelo autor (2020)

Portanto, já sabemos: toda vez que tivermos um conjunto de dados imutáveis dentro do nosso script, uma tupla faz muito mais sentido do que uma lista!

Existe, porém, um mundo ainda maior para explorarmos com nossos grandes volumes de dados!

1.3.2 Principais métodos e funções

Com as tuplas, podemos utilizar alguns métodos, como:

- count() – retorna a quantidade de vezes que um valor aparece na tupla.
- index() – procura na tupla um valor e retorna o número do índice da primeira ocorrência desse valor.

Vamos colocar os dois métodos em um único script para compreender seu uso?

```
#criação da tupla
tupla = (1, 7, 7, 19, 3, 2, 11, 15, 6, 1, 5)

#exibição da tupla
print(f"A tupla foi criada assim: {tupla}")

#contagem de elementos
contagem = tupla.count(7)
print(f"Nessa tupla o número {7} aparece {contagem} vezes")

#índice em que encontrou o valor
indice = tupla.index(11)
print(f"O número {11} foi encontrado no índice: {indice}")
```

Código-fonte 12 – Métodos que podem ser utilizados com a classe tupla no Python
Fonte: Elaborado pelo autor (2022)

Além dos métodos apresentados, podemos utilizar também algumas funções embutidas no Python. A seguir, temos algumas funções utilizadas com as tuplas:

- all() – retorna True se todos os elementos de um objeto iterável (lista, tupla, dicionário, set etc) forem True ou também se o objeto estiver vazio.
- any() – retorna False se todos os elementos de um objeto iterável forem False ou 0 (zero). Também retorna False se o objeto estiver vazio.
- enumerate() – recebe uma coleção e a retorna como um objeto enumerado.
- len() – retorna a quantidade de elementos em uma coleção.
- max() – retorna o maior valor entre os elementos de um objeto.

- min() – retorna o menor valor entre os elementos de um objeto.
- sum() – realiza a soma dos elementos presentes em um objeto.
- tuple() – converte um objeto em uma tupla.

Agora, vamos testar essas funções com as tuplas!

```
#criação e exibição da tupla
tupla = (1, 7, 7, 19, 3, 2, 11, 15, 6, 1, 5)
print(f"A tupla foi criada assim: {tupla}")

#utilizando enumerate para mostrar uma sequência
for numero, valor in enumerate(tupla):
    print(f"No índice {numero} temos: {valor}")

#mostrando a quantidade de itens na tupla
print(f"Quantidade: {len(tupla)}")

#mostrando o valor mínimo na tupla
print(f"Mínimo: {min(tupla)}")

#mostrando o valor máximo na tupla
print(f"Máximo: {max(tupla)}")

#mostrando a soma de todos os valores da tupla
print(f"Máximo: {sum(tupla)}")

#utilizando tuple() para a conversão de uma lista para uma
tupla
lista = [True,False]
print(f"Lista: {lista}")
tupla2 = tuple(lista)
print(f"Tupla: {tupla2}")

#criando a tupla3 com os valores 1 (True) e 0 (False)
tupla3 = (1,0)

#função all
print(f"Testando a tupla2 com all: {all(tupla2)}")
print(f"Testando a tupla3 com all: {all(tupla3)}")

#função any
print(f"Testando a tupla2 com any: {any(tupla2)}")
print(f"Testando a tupla3 com any: {any(tupla3)}")
```

Código-fonte 13 – Principais funções que podem ser utilizadas com as tuplas em Python
Fonte: Elaborado pelo autor (2022)

2 O QUE ESTÁ NO DICIONÁRIO?

Neste capítulo, já viajamos tanto pelo espaço (seja ele da memória RAM, seja de uma galáxia muito, muito distante) que faz sentido continuarmos com o mesmo exemplo.

Já sabemos quando devemos usar as lists e quando devemos usar as tuples, mas será que já sabemos quando ambas são insuficientes?

Se nós quisermos saber qual personagem do universo Star Wars pertence a qual categoria, poderíamos usar duas listas, certo? Uma para os personagens e outra para as categorias, associando o mesmo índice às duas.

Algo parecido com o Código-fonte “Usando duas listas para associar dados”.

```
     #Criando as duas listas
     personagens=[]
     categorias=[]

     #Executando um loop 10 vezes
     for x in range(10):
         #pedindo que o usuário informe um nome e colocando na
lista de personagens
         personagens.append(input("Informe    o    nome    do
personagem: "))
         #pedindo que o usuário informe a categoria e colocando
na lista de categorias
         categorias.append(input("Informe   a   categoria   do
personagem: "))
     #Executando um loop 10 vezes
     for indice in range(10):
         #exibindo a cada volta o que está contido em um índice
de personagens e categorias
         print("O      personagem      {}      é      um(a)
{}".format(personagens[indice], categorias[indice]))
```

Código-fonte 14 – Usando duas listas para associar dados
Fonte: Elaborado pelo autor (2020)

Se você se lembrar do início do capítulo, vai se lembrar de que o *lado sombrio* é tentador e, nesse caso, o lado sombrio é usar listas para tudo.

Apesar de resolver o problema para o usuário, as listas estão nos forçando a fazer controles manuais de tudo! Veja: é o programador que está fazendo o controle

de associar que o que está contido em um mesmo índice nas duas listas representa a mesma informação.

Se algum desavisado tentar exibir o personagem do índice 3 com a categoria do índice 1, vai acabar achando que o Darth Vader é um balconista de alguma cafeteria intergaláctica.

Quem vai solucionar o nosso problema, agora, são os dicionários de dados!

Os **dicionários de dados** são estruturas em Python que funcionam seguindo a lógica de *chave* e *valor*. Isso quer dizer que poderemos associar dois dados, fazendo com que um desses dados seja uma *chave única* que identificará um valor.

Ficou confuso? Vamos dar uma olhada nos dados da Figura “Personagens de Star Wars e suas categorias”.

Personagem	Categoria
Yoda	Mestre Jedi
Mace Windu	Mestre Jedi
Anakin Skywalker	Cavaleiro Jedi
R2-D2	Dróide
Dex	Balconista

Figura 4 – Personagens de Star Wars e suas categorias
Fonte: Elaborado pelo autor (2020)

Se olharmos para a categoria, não conseguimos obter apenas um personagem, certo? Mas, se olharmos para um personagem, conseguimos saber exatamente sua categoria!

Partindo desse raciocínio, descobrimos que os personagens são as nossas chaves, e as categorias são os nossos valores.

Agora que a lógica está posta, vamos criar um dicionário em Python?

2.1 A criação de um dicionário

Para aplicarmos a lógica de *key* e *value* que os *dictionaries* em Python exigem, precisaremos criar essa estrutura. O símbolo que será usado agora é o símbolo de chaves.

A criação de um dicionário vazio, portanto, fica como no Código-fonte “Criação de um dicionário vazio em Python”.

```
    #criando um dicionário vazio
    dicionario = {}

    #exibindo o tipo do dicionário
    print("O objeto dicionario é do tipo
{}".format(type(dicionario)))
```

Código-fonte 15 – Criação de um dicionário vazio em Python
Fonte: Elaborado pelo autor (2020)

Se quisermos criar um dicionário que contenha dados, precisamos organizá-los da seguinte forma: escreveremos o dado chave, colocaremos o símbolo de dois pontos e, só então, escreveremos o dado valor.

Pegando como exemplo nossa lista de personagens de Star Wars, teremos o dicionário indicado no Código-fonte “Criação de um dicionário com dados em Python”.

```
    #criando um dicionário com dados
    dicionario = {"Yoda":"Mestre Jedi", "Mace Windu": "Mestre
Jedi", "Anakin Skywalker":"Cavaleiro Jedi", "R2-D2":"Dróide",
"Dex":"Balconista"}
```

Código-fonte 16 – Criação de um dicionário com dados em Python
Fonte: Elaborado pelo autor (2020)

Talvez a mera *criação* de um dicionário não seja o suficiente para mostrarmos seu poder. Então, vamos partir para a exibição?

2.2 Exibindo o conteúdo de um dicionário

A grande característica de um dicionário é a associação entre uma chave e um valor, e essa característica se revela justamente no momento de exibir um conteúdo.

Se escrevermos o Código-fonte “Exibição do valor de uma chave no dicionário em Python”, imediatamente será exibido o valor associado à chave “R2-D2”.

```
    #criando um dicionário com dados
    dicionario = {"Yoda":"Mestre Jedi", "Mace Windu": "Mestre
Jedi", "Anakin Skywalker":"Cavaleiro Jedi", "R2-D2":"Dróide",
"Dex":"Balconista"}

    #exibindo o valor associado a uma chave específica
    print(dicionario["R2-D2"])
```

Código-fonte 17 – Exibição do valor de uma chave no dicionário em Python
Fonte: Elaborado pelo autor (2020)

Podemos também exibir todos os valores presentes em um dicionário, por meio de um loop associado ao método *values*.

```
    #criando um dicionário com dados
    dicionario = {"Yoda":"Mestre Jedi", "Mace Windu": "Mestre
Jedi", "Anakin Skywalker":"Cavaleiro Jedi", "R2-D2":"Dróide",
"Dex":"Balconista"}

    #exibindo o valor associado a uma chave específica
    print(dicionario["R2-D2"])

    #exibindo todos os valores em um dicionário
    for valor in dicionario.values():
        print(valor)
```

Código-fonte 18 – Exibição de todos os valores em um dicionário
Fonte: Elaborado pelo autor (2020)

E, até mesmo, exibir todas as chaves, associando um loop ao método *Keys*.

```
    #criando um dicionário com dados
    dicionario = {"Yoda":"Mestre Jedi", "Mace Windu": "Mestre
Jedi", "Anakin Skywalker":"Cavaleiro Jedi", "R2-D2":"Dróide",
"Dex":"Balconista"}

    #exibindo todas as chaves em um dicionário
    for chave in dicionario.keys():
        print(chave)
```

Código-fonte 19 – Exibição de todas as chaves em um dicionário
Fonte: Elaborado pelo autor (2020)

Mas o que vai ser ainda mais poderoso é exibirmos tanto as chaves quanto os valores. Podemos fazer isso utilizando um loop for que utilize duas variáveis para navegar dentro do dicionário com o método *items*.

```
    #criando um dicionário com dados
    dicionario = {"Yoda":"Mestre Jedi", "Mace Windu": "Mestre
Jedi", "Anakin Skywalker":"Cavaleiro Jedi", "R2-D2":"Dróide",
"Dex":"Balconista"}

    #exibindo o dicionário completo
    for chave, valor in dicionario.items():
```

```
        print("O personagem {} é da categoria {}".format(chave, valor))
```

Código-fonte 20 – Exibição de todo o conteúdo do dicionário
Fonte: Elaborado pelo autor (2020)

Se compararmos nosso último script com a tentativa de associar os dados utilizando listas, vamos perceber o quão mais fácil o dicionário torna nossa vida.

É claro que, como todo programador, você deve estar se perguntando: mas como eu faço para o *usuário* digitar dados que serão armazenados no dicionário? É justamente o que vamos aprender agora.

2.3 Inserindo novos dados em um dicionário

Já aprendemos a criar dicionários com e sem dados e a exibir os itens armazenados neles. Chegou a hora de inserirmos novos dados durante a execução do script.

Essa tarefa pode ser facilmente cumprida com a estrutura **nome_do_dicionario[chave_a_ser_incluida]=valor_a_ser_incluido**.

O script Código-fonte “Inclusão de novos valores no dicionário” demonstra esse procedimento com dados escritos no código e com dados escritos pelo usuário.

```
    #criando um dicionário vazio
    dicionario = {}

    #incluindo dados no dicionário
    dicionario["André David"] = "Professor"

    #Pedindo para o usuário incluir dados no dicionário
    nova_chave = input("Informe o nome do colaborador da FIAP")
    novo_valor = input("Informe a função do colaborador da FIAP")
    dicionario[nova_chave] = novo_valor

    #exibindo o dicionário completo
    for chave, valor in dicionario.items():
        print("O colaborador {} desempenha a função de {}".format(chave, valor))
```

Código-fonte 21 – Inclusão de novos valores no dicionário
Fonte: Elaborado pelo autor (2020)

É importante notar, porém, que a mesma estrutura utilizada para incluir novos itens em um dicionário serve para **substituir** itens que já estavam lá. Portanto, se indicarmos uma chave pré-existente, seu valor será substituído. Veja o Código-fonte “Substituição de valores antigos em um dicionário”.

```
     #criando um dicionário vazio
     dicionario = {}

     #incluindo dados no dicionário
     dicionario["André David"] = "Professor"

     #alterando dados no dicionário
     dicionario["André David"] = "Coordenador"


     #exibindo o dicionário completo
     for chave, valor in dicionario.items():
         print("O colaborador {} desempenha a função de
{}".format(chave, valor))
```

Código-fonte 22 – Substituição de valores antigos em um dicionário
Fonte: Elaborado pelo autor (2020)

É claro que quem cria, inclui e altera também vai precisar... excluir! Nossa próxima etapa é remover itens que estejam no nosso dicionário.

2.4 Removendo dados de um dicionário

Existem, basicamente, três formas de removermos itens de um dicionário: remover um item específico por meio de sua chave, remover o último item inserido ou apagar o dicionário inteiro.

Para remover um item específico, utilizamos o método *pop*, indicando entre parênteses qual é a chave do item em questão, como mostra o Código-fonte “Removendo um item específico do dicionário”.

```
     #criando um dicionário com dados
     dicionario = {"Yoda":"Mestre Jedi", "Mace Windu": "Mestre
Jedi", "Anakin Skywalker":"Cavaleiro Jedi", "R2-D2":"Dróide",
"Dex":"Balconista"}

     #exibindo o dicionário completo
     for chave, valor in dicionario.items():
         print("O personagem {} é da categoria
{}".format(chave, valor))
```

```
    #removendo o item cuja chave é R2-D2
    dicionario.pop("R2-D2")

    #exibindo o dicionário completo após a remoção
    for chave, valor in dicionario.items():
        print("O    personagem    {}    é    da    categoria
{}".format(chave, valor))
```

Código-fonte 23 – Removendo um item específico do dicionário
Fonte: Elaborado pelo autor (2020)

Se quisermos remover apenas o último item inserido, o método a ser usado é o *popitem()*. É importante ficarmos atento, pois, em versões antigas do Python, esse método removia um item aleatório.

```
    #criando um dicionário com dados
    dicionario = {"Yoda":"Mestre Jedi", "Mace Windu": "Mestre
Jedi", "Anakin Skywalker":"Cavaleiro Jedi", "R2-D2":"Dróide",
"Dex":"Balconista"}

    #exibindo o dicionário completo
    for chave, valor in dicionario.items():
        print("O    personagem    {}    é    da    categoria
{}".format(chave, valor))

    #removendo o último item
    dicionario.popitem()

    #exibindo o dicionário completo após a remoção
    for chave, valor in dicionario.items():
        print("O    personagem    {}    é    da    categoria
{}".format(chave, valor))
```

Código-fonte 24 – Removendo o último item inserido no dicionário
Fonte: Elaborado pelo autor (2020)

Por fim, o dicionário pode ser totalmente limpo ao usar o método *clear()*.

```
    #criando um dicionario com dados
    dicionario = {"Yoda":"Mestre Jedi", "Mace Windu": "Mestre
Jedi", "Anakin Skywalker":"Cavaleiro Jedi", "R2-D2":"Dróide",
"Dex":"Balconista"}

    #exibindo o dicionário completo
    for chave, valor in dicionario.items():
        print("O    personagem    {}    é    da    categoria
{}".format(chave, valor))

    #removendo todos os itens do dicionário
    dicionario.clear()

    #exibindo o dicionário completo após a remoção
    for chave, valor in dicionario.items():
```

```
        print("O personagem {} é da categoria {}".format(chave, valor))
```

Código-fonte 25 – Removendo todos os itens de um dicionário
Fonte: Elaborado pelo autor (2020)

É importante atentarmos que todas essas tarefas são irreversíveis, então, muita atenção na hora de remover itens de um dicionário.

2.5 Dicionários dentro de dicionários

Durante todo este capítulo, você deve ter percebido como os dicionários são complexos e úteis para um desenvolvedor que utilize a linguagem Python, mas, e se eu lhe contar que eles podem fazer mais?

Colocando um dicionário dentro de outro, podemos ter uma estrutura de dados realmente poderosa para solucionar nossos problemas.

Se você fosse um funcionário da Google e tivesse que criar a lista de contatos de um smartphone, precisaria escolher uma estrutura que possibilitasse associar uma pessoa a diversas formas de contato, não é?

Quem resolverá nosso problema são os *nested dictionaries* ou dicionários aninhados.

Teremos o nome do contato como chave. O valor dessa chave, porém, será outro dicionário, no qual cada chave será o nome da forma de contato e cada valor será a forma de contato em si.

Ficou confuso? Vamos tentar enxergar, na Figura “Contato em um smartphone Android”, a imagem de um contato em um smartphone Android.

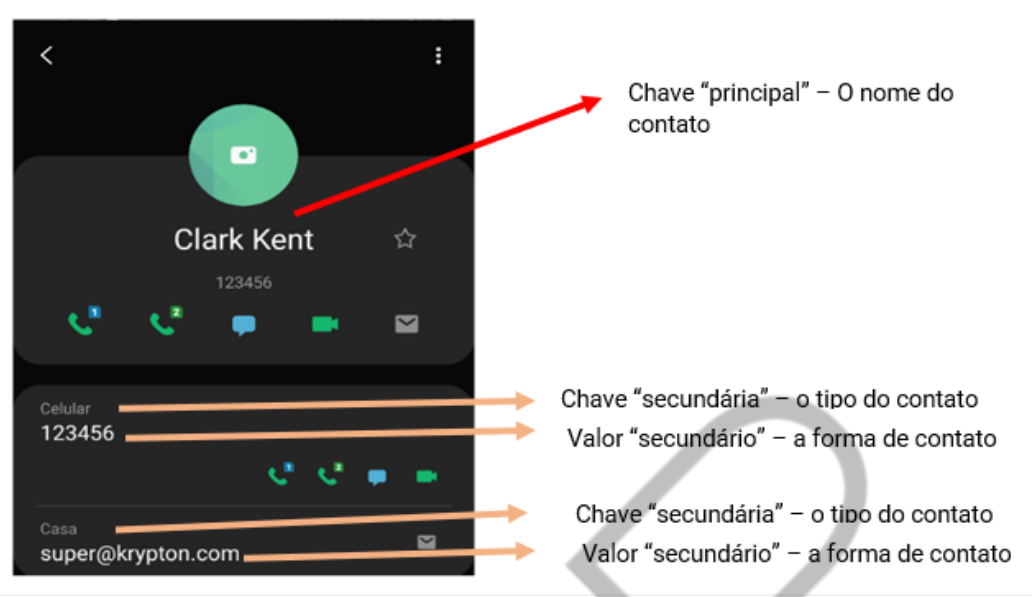

Figura 5 – Contato em um smartphone Android
Fonte: Elaborado pelo autor (2020)

Para criar esse dicionário aninhado, poderíamos seguir a estrutura apresentada no Código-fonte “Criando um dicionário aninhado para a lista de contatos”.

```
#criando o dicionário dos contatos
contatos = {
    "Clark Kent":
        {"Celular":"123456",
         "Email":"super@krypton.com"},
    "Bruce Wayne":
        {"Celular":"654321",
         "Email":"bat@caverna.com.br"}
}
```

Código-fonte 26 – Criando um dicionário aninhado para a lista de contatos
Fonte: Elaborado pelo autor (2020)

Perceba que o dicionário contatos é composto pelas chaves “Clark Kent” e “Bruce Wayne”. Cada uma das chaves tem como valores um segundo dicionário, composto pelas chaves “Celular” e “Email”, com seus respectivos valores.

Para navegar nessa estrutura, podemos criar dois loops: um para o dicionário mais externo, com o nome dos contatos, e outro para o dicionário mais interno, com as formas de contato.

```
    #esse for passará por todos os items do dicionário
contatos, com a variável "contato" contendo as chaves desses
items e o objeto formas contendo os valores, que são os
dicionários de formas de contatos
    for contato, formas in contatos.items():
        #para cada item encontrado no dicionário anterior,
que estão contidos no dicionário "formas", vamos recuperar as
chaves "celular" e "email" e seus valores
        for celular, email in formas.items():
            #exibimos aqui o nome do contato e as suas formas
de contato
            print("O contato {} pode ser encontrado no celular
{} e no email {}".format(contato, celular, email))
```

Código-fonte 27 – Navegando pelo dicionário aninhado
Fonte: Elaborado pelo autor (2020)

Os dicionários aninhados são extremamente úteis!

Vale a pena explorar essa estrutura e fazer suas próprias experiências para descobrir novas potencialidades dessa ferramenta.

É hora de dar asas à imaginação e ver o quão longe o Python pode levá-lo!

REFERÊNCIAS

PIVA J., D. **Algoritmos e programação de computadores**. São Paulo: Elsevier, 2012.

PUGA, S.; RISSETTI, G. **Lógica de programação e estrutura de dados**. São Paulo: Pearson Prentice Hall, 2009.

RAMALHO, L. **Python Fluente**: **programação clara, concisa e eficaz**. São Paulo: Novatec, 2015.