FIAP ON

CONSOLIDANDO OS CONHECIMENTOS

A LÓGICA COM PYTHON

04

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

LISTA DE COMANDOS DE PROMPT DO SISTEMA OPERACIONAL

SUMÁRIO

1 A LÓGICA COM PYTHON

1.1 Do simples ao composto

No mundo da programação, um dos primeiros ensinamentos é o de armazenamento de dados na memória RAM. Para isso, utilizamos o conceito de **variáveis de memória**, em que cada variável tem um nome, um tipo e pode armazenar apenas uma informação.

Para a resolução dos nossos problemas até aqui, a variável foi suficiente, mas problemas mais complexos podem necessitar de armazenamentos mais sofisticados – manter o histórico dos conteúdos das variáveis seria um exemplo.

Neste caso, devemos utilizar as variáveis complexas: **Estruturas de dados**, que nada mais são do que identificadores com autonomia para armazenar múltiplas informações.

1.2 Aplicação dessa necessidade no cotidiano

Considere que, em uma loja de roupas, trabalham vendedores e um deles é o VENDEDOR1. No fim do mês, precisamos saber quanto o VENDEDOR1 acumulou de vendas para calcular a sua comissão.

Para essa necessidade, podemos simplesmente criar uma variável do tipo float (real) e nela guardarmos esta informação: `vendedor1 = 23456.57`.

Utilizando essa atribuição, a alocação da variável VENDEDOR1 na memória RAM ficará:

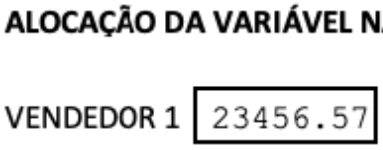

Figura 1 – Alocação de variável simples na memória
Fonte: Elaborado pelo autor (2022)

Visando incentivar os vendedores a venderem mais, a loja decidiu bonificar o funcionário que fizesse a maior venda (em nosso exemplo, o VENDEDOR1) em um dia dentro do mês; sendo assim, o sistema deve descobrir qual é a maior venda de cada vendedor para fazer a comparação no fim do mês, mas qual é o problema?

```
vendedor1 = 345.76
vendedor1 = 3234.73
vendedor1 = 123.45
vendedor1 = 2345.77
vendedor1 = 3485.76
vendedor1 = 3134.79
vendedor1 = 1238.45
vendedor1 = 2335.77
...
```

Código-fonte 1 – Atribuições múltiplas na mesma variável
Fonte: Elaborado pelo autor (2022)

Considere que cada linha acima representa a venda de um dia do VENDEDOR1 a partir do dia 1º. Lembrando, o conceito de variável é: “Variável de memória é o local na memória do computador onde armazenamos UMA informação”. Ou seja, uma variável só guarda uma informação.

Considerando esse conceito, foram atribuídos valores diferentes para a [mesma] variável VENDEDOR1 e acabou que, no fim do processamento dessas linhas, só prevaleceu o último valor, porque os anteriores foram sobrepostos.

Temos um problema! Não é possível usar variáveis simples para essa necessidade. A solução para esse problema é utilizar variáveis compostas: **Estrutura de dados**.

Nesse tipo de estrutura, conseguimos armazenar na mesma variável diversas informações; então a alocação dessa estrutura na memória ficaria parecida com:

ALOCAÇÃO DA VARIÁVEL NA MEMÓRIA

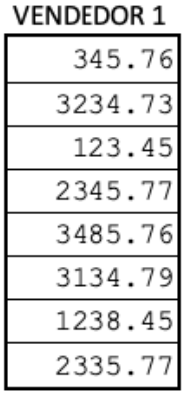

Figura 2 – Alocação de variável composta na memória
Fonte: Elaborado pelo autor (2022)

Utilizando esse conceito, todos os valores poderão ser tratados separadamente dentro da mesma variável. Os detalhamentos de como acessar cada valor estarão nas próximas seções.

1.3 Sobre a aplicação de Estrutura de dados neste capítulo

Nesta seção, utilizaremos a seguinte metodologia:

Vamos aplicar o novo conteúdo – Estrutura de dados – com subalgoritmos

A vantagem de utilizarmos essa abordagem é a aplicação, a prática e a consolidação do conhecimento adquirido anteriormente, que foi subalgoritmos, neste novo aprendizado.

Profissionalmente, todos nós veremos que a maioria das linguagens de programação, e, consequentemente, os códigos-fontes, utiliza POO (programação orientada a objetos), cujo princípio é o subalgoritmo.

2 VARIÁVEIS INDEXADAS

Variáveis indexadas – também conhecidas como arrays – são aquelas que têm a capacidade de armazenar diversas informações dentro de apenas uma variável.

Os valores são alocados em uma posição que é localizada pelo índice que inicia no zero (0) [na maioria das linguagens] e vai até o limite estabelecido em sua definição.

Em Python, não há uma implementação específica para as variáveis indexadas (vetor ou matriz), mas conseguimos simulá-las com listas.

2.1 Variável indexada unidimensional – vetor

A variável unidimensional (vetor – array) é a classe de variável que tem UMA linha e C colunas (sendo C maior que 1); com isso, ela é capaz de armazenar diversos valores. Cada ELEMENTO contém uma CÉLULA, que é o seu conteúdo, e um ÍNDICE, que é a posição de localização do elemento.

Veja a ilustração:

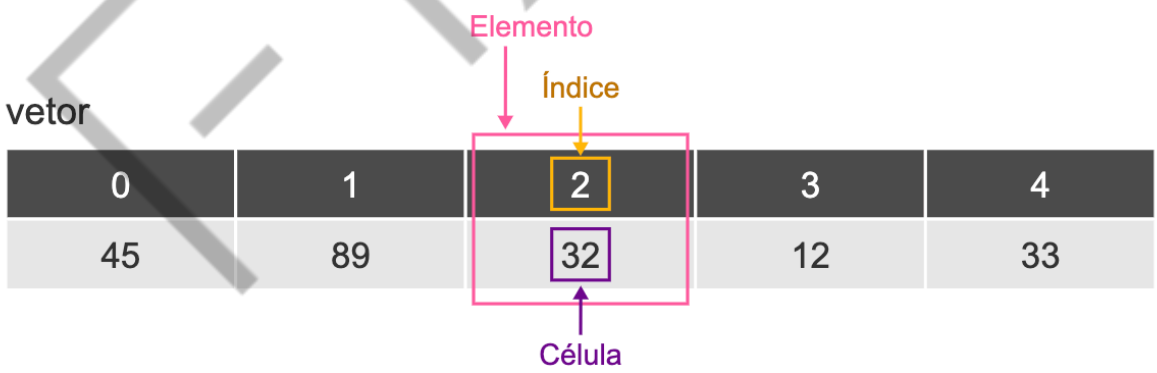

Figura 3 – Componentes de um vetor
Fonte: Elaborado pelo autor (2022)

Nesta literatura, considere como sinônimos as palavras “Célula” e “Conteúdo” e as palavras “Índice” e “Posição”.

Em vetores, as células devem ter os seus conteúdos do mesmo tipo (int, float, str, bool...).

Vejamos em Python a criação e a atribuição dos valores da figura acima de forma direta:

```
     # iniciando o vetor com valores
vetor = [45, 89, 32, 12, 33]
```

Código-fonte 2 – Inicializando um vetor
Fonte: Elaborado pelo autor (2022)

Onde ficam os índices? Na inicialização acima, os índices não são “vistos”. Com o comentário do código abaixo, fica mais evidente:

```
     #índice   0   1   2   3   4
vetor = [45, 89, 32, 12, 33]
```

Código-fonte 3 – Apresentando os índices
Fonte: Elaborado pelo autor (2022)

Concluímos então que esse vetor tem 5 elementos do tipo int.

Um vetor tem o tamanho predefinido no programa pelo programador (veremos em lista que até dá para acrescentar mais elementos). O vetor pode ser manuseado integralmente ou por partes (pelo índice).

Veja o código abaixo:

```
# iniciando o vetor com valores
vetor = [45, 89, 32, 12, 33]

# Exibindo o conteúdo integral do vetor
print("Vetor inteiro.........: ", vetor)

# Exibindo uma célula específica
print("Posição 2.............: ", vetor[2])

# Modificando o valor do índice 0 (zero)
vetor[0] = -55
print("Índice 0 modificado...: ", vetor)
```

Código-fonte 4 – Manipulando o vetor
Fonte: Elaborado pelo autor (2022)

Execução:

```
Vetor inteiro.........:  [45, 89, 32, 12, 33]
Posição 2.............:  32
Índice 0 modificado...:  [-55, 89, 32, 12, 33]
```

Comando de prompt 1 – Mostrando os elementos do vetor
Fonte: Elaborado pelo autor (2022)

No exemplo acima, o vetor de nome `vetor` é iniciado com valores predefinidos. Logo depois, ele é exibido integralmente na tela. Na sequência, é exibido somente o conteúdo da célula cujo índice é o 2 (repare que o índice é delimitado ao lado da variável entre colchetes). Em seguida, o elemento com índice 0 (zero) é modificado para -55 e, logo depois, é exibido.

2.2 Características de um vetor

As variáveis indexadas unidimensionais do tipo vetor têm características específicas (essas características também se aplicam às variáveis bidimensionais – matriz – que veremos adiante), as quais as diferenciam das outras estruturas de dados.

- O tamanho é predefinido na sua criação. O programador é quem define quantos elementos o vetor terá no máximo.
- As células (conteúdos dos elementos) têm conteúdo homogêneo. Uma vez definido o tipo, ele deve ser o mesmo para todas as células.
- O índice começa no zero. Sendo assim, o índice 0 é o primeiro elemento.
- Os índices são tratados entre colchetes. Sempre que quisermos nos referír a um elemento específico, colocamos entre colchetes o número do índice.

2.3 Manipulando o vetor

Trabalhar com os elementos do vetor pode ser exaustivo caso não seja feito da forma adequada. Imagine um vetor com 100 elementos e nele tenhamos que preencher todas as células com valores digitados pelo usuário.

Convém digitar: `vetor[0] = input("Digite um valor:")` 100 vezes (modificando o índice)? Não! Com isso, escreveríamos 100 linhas ao menos. O ideal seria elaborarmos um laço que dê 100 voltas e a cada volta alimente uma célula.

Para exemplificar esse conceito, vamos considerar o seguinte problema:

Um aluno (ALUNO1) tem 10 avaliações por ano. Com esse aluno desejamos executar as seguintes operações:

- Armazenar as notas dessas 10 avaliações.
- Exibir todas as 10 notas.
- Exibir uma nota específica.
- Calcular a média das 10 notas.

Lembre-se, para resolvermos esse problema, utilizaremos subalgoritmos

Primeiramente, para fazermos essas rotinas, vamos considerar um vetor de float chamado `aluno1` com 10 posições inicializadas com a nota zero:

```
    # Inicialização do vetor de float com 10 posições
aluno1 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Código-fonte 5 – Inicializando um vetor com valores
Fonte: Elaborado pelo autor (2022)

Armazenando as notas no vetor com valores digitados:

```
    # Armazena em um vetor as notas das 10 avaliações
def preenche_notas(v: list) -> None:
    for i in range(0, 10, 1):
        v[i] = float(input(margem + f"Avaliação {i + 1}..: "))
```

Código-fonte 6 – Digitando valores no vetor
Fonte: Elaborado pelo autor (2022)

Esse procedimento passa o vetor por parâmetro e permite ao usuário preencher as 10 notas de uma vez.

Veja a execução:

```
        Avaliação 1..: 5.5
        Avaliação 2..: 6
        Avaliação 3..: 7
        Avaliação 4..: 9
        Avaliação 5..: 10
        Avaliação 6..: 4.6
        Avaliação 7..: 1
        Avaliação 8..: 3
        Avaliação 9..: 4
        Avaliação 10.: 7.6
```

Comando de prompt 2 – Digitando valores no vetor
Fonte: Elaborado pelo autor (2022)

Exibindo as notas armazenadas no vetor:

```
    # Exibe as notas das 10 avaliações
def exibe_notas(v: list) -> None:
    for i in range(0, 10, 1):
        print(margem + f"Avaliação {i + 1} = {v[i]}")
```

Código-fonte 7 – Exibindo os valores do vetor
Fonte: Elaborado pelo autor (2022)

O vetor é passado por parâmetro (representado por `v` e já preenchido) e será totalmente exibido na tela. Dentro do print, há um `{i + 1}`, usamos esse recurso pelo fato de o índice começar no zero, ficaria estranho partir a exibição de `Avaliação 0 =`, então parte de `0 + 1 = 1`, ficando `Avaliação 1 =`.

Veja a Execução:

```
   Avaliação 1 => 5.5
   Avaliação 2 => 6.0
   Avaliação 3 => 7.0
   Avaliação 4 => 9.0
   Avaliação 5 => 10.0
   Avaliação 6 => 4.6
   Avaliação 7 => 1.0
   Avaliação 8 => 3.0
   Avaliação 9 => 4.0
   Avaliação 10=> 7.6
```

Comando de prompt 3 – Exibindo os valores preenchidos no vetor
Fonte: Elaborado pelo autor (2022)

Exibindo uma nota específica:

```
     # Exibe uma nota específica
def exibe_uma_nota(v: list, av: int) -> float:
    if av > 0 and av <= 10:
        return v[av - 1] # este -1 é pq o índice e a avaliação
sempre tem 1 de diferença
    else:
        return margem + f"Não há a avaliação {av}"
```

Código-fonte 8 – Construção da função exibe_uma_nota
Fonte: Elaborado pelo autor (2022)

Nesse caso, além do vetor passado por parâmetro, devemos passar o número correspondente da avaliação que será fornecida para a função, e ela retornará a nota da avaliação, ou, caso esse número esteja fora do range, exibirá uma mensagem de erro.

Será necessário que o usuário digite (no programa principal) qual é o número da avaliação da qual ele quer saber a nota. Esse procedimento verifica se foi escolhida uma avaliação entre 0 e 10. Na linha `return v[av - 1]`, o –1 é referente à diferença entre a avaliação e a posição do índice.

Veja a execução:

```
        Digite o número da avaliação: 4
        Avaliação 4 => 9.0
```

Comando de prompt 4 – Resultado da função exibe_uma_nota
Fonte: Elaborado pelo autor (2022)

Calculando a média de todas as notas contidas no vetor:

```
     # Calculando a média geral
def media_geral(v: list) -> float:
    soma = 0
    for i in range(0, 10, 1):
        soma += v[i]
    return soma / 10
```

Código-fonte 9 – Construção da função media_geral
Fonte: Elaborado pelo autor (2022)

Essa função calcula a somatória das 10 notas contidas no vetor e retorna essa somatória dividida por 10, resultando a média.

Veja a execução:

```
        Média = 5.8
```

Comando de prompt 5 – Resultado da função media_geral
Fonte: Elaborado pelo autor (2022)

As demais funções e procedimentos são auxiliares para que o programa principal funcione adequadamente.

Segue o programa completo com os subalgoritmos:

```
    """
Autor.....: Prof. Edson de Oliveira
Data......: 10/11/2022
Assunto...: Manipulação de vetores utilizando subalgoritmos
Problema..: Preencher, exibir e calcular a média de 10 notas
de um aluno
"""


# ----- SUBALGORITMOS
# Armazena em um vetor as notas das 10 avaliações
def preenche_notas(v: list) -> None:
    for i in range(0, 10, 1):
        v[i] = float(input(margem + f"Avaliação {i + 1}..: "))


# Exibe as notas das 10 avaliações
def exibe_notas(v: list) -> None:
    for i in range(0, 10, 1):
        print(margem + f"Avaliação {i + 1} => {v[i]}")


# Exibe uma nota específica
def exibe_uma_nota(v: list, av: int) -> float:
    if av > 0 and av <= 10:
        return v[av - 1]  # este -1 é pq o índice e a
avaliação sempre tem 1 de diferença
    else:
        return margem + f"Não há a avaliação {av}"


# Calculando a média geral
def media_geral(v: list) -> float:
    soma = 0
    for i in range(0, 10, 1):
```

```
        soma += v[i]
    return soma / 10


# ----- SUBALGORITMOS AUXILIARES
def exibe_menu() -> None:
    print("""
        0 - SAIR
        1 - Preencher todas as notas
        2 - Exibir todas as notas
        3 - Exibir uma nota específica
        4 - Exibir a média de todas as notas
    """)


def opcao_menu() -> int:
    exibe_menu()
    return int(input(margem + "Opção desejada: "))


# ----- PROGRAMA PRINCIPAL

# Inicialização do vetor de float com 10 posições
aluno1 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

# Flag que controla se o vetor está preenchido (inicia não
preenchido)
preenchido = False
margem = ' ' * 8

# Rotina que executa o menu indefinidamente
while True:
    opcao = opcao_menu()

    # escolha da opção do menu digitada
    match opcao:
        case 0:
            break
        case 1:
            preenche_notas(aluno1)
            # modifica o flag para True depois que o vetor for
preenchido
            preenchido = True
        case 2:
            # analisa se o vetor está preenchido para exibir
as notas
            if preenchido:
                exibe_notas(aluno1)
            else:
                print(margem + "***** Notas não preenchidas
*****")
```

```
        case 3:
            if preenchido:
                avaliacao = int(input(margem + "Digite o
numero da avaliação: "))
                print(margem + f"Avaliação {avaliacao} =>
{exibe_uma_nota(aluno1, avaliacao)}")
            else:
                print(margem + "***** Notas não preenchidas
*****")
        case 4:
            if preenchido:
                print(margem + f"Média =
{media_geral(aluno1):.1f}")
            else:
                print(margem + "***** Notas não preenchidas
*****")
        case _:
            print(margem + "***** Opção inválida! *****")
```

Código-fonte 10 – Solução integral dos cálculos da média de 10 notas
Fonte: Elaborado pelo autor (2022)

Com esse exemplo pudemos explorar o conceito de Vetor.

2.2 Variável indexada bidimensional – matriz

A matriz, variável indexada bidimensional tem praticamente as mesmas características do vetor, com a exceção de ser uma estrutura L (linhas) por C (colunas) sendo L e C maiores do que 1 obrigatoriamente.

Utilizando uma matriz, criamos uma “tabela” ou “planilha” que armazena informações do mesmo tipo. Graficamente, ela seria representada da seguinte forma:

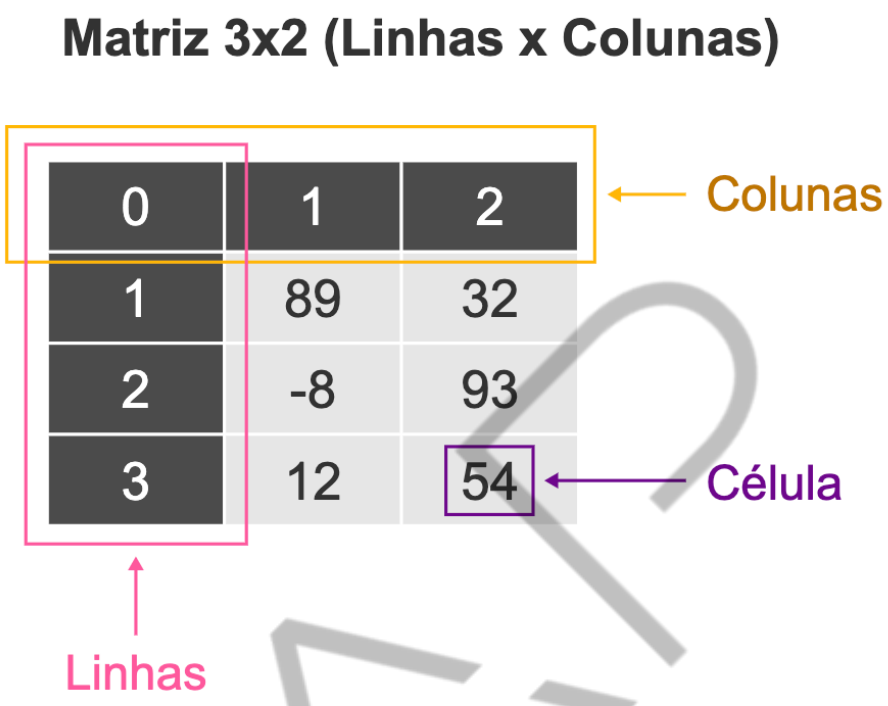

Figura 4 – Elementos de uma matriz com 3 linhas e 2 colunas
Fonte: Elaborado pelo autor (2022)

Nessa figura, temos uma matriz 3 x 2 (três linhas por duas colunas). As **Linhas** são os índices verticais; as **Colunas** são os índices horizontais, e as **Células** nada mais são do que as intersecções das linhas com as colunas.

Existem várias formas de criar uma matriz em Python, utilizaremos a clássica:

```
    # Criando uma matriz 3 x 2 com os valores da figura
matriz = [
    [89, 32],
    [-8, 93],
    [12, 54],
]
```

Código-fonte 11 – Inicialização de uma matriz 3 x 2 com valores
Fonte: Elaborado pelo autor (2022)

Para exibirmos a matriz na forma “bruta”, utilizamos o comando:

```
    # Exibindo a matriz integral
print("Integral: ", matriz)
```

Código-fonte 12 – Exibindo o conteúdo da matriz 3 x 2
Fonte: Elaborado pelo autor (2022)

Veja a execução:

```
Integral: [[89, 32], [-8, 93], [12, 54]]
```

Comando de prompt 6 – Apresentação bruta da matriz 3 x 2
Fonte: Elaborado pelo autor (2022)

Para modificarmos o valor de uma célula da matriz, temos agora dois pares de colchetes, por exemplo: `matriz[0][0] = 99`. O primeiro par de colchetes representa a Linha, e o outro, a Coluna.

Agora, faremos algumas manipulações na matriz:

```
     # Criando uma matriz 3 x 2
matriz = [
    [89, 32],
    [-8, 93],
    [12, 54],
]

# Exibindo a matriz integral
print("Integral: ", matriz)

# Modificando a célula com linha 0 coluna 0 com o valor 99
matriz[0][0] = 100

# Mostrando o conteúdo da célula 0 x 0
print(f"Matriz[{0}][{0}] = {matriz[0][0]}")

# Mostrando novamente a matriz com a célula modificada
print("Integral", matriz)
```

Código-fonte 13 – Manipulando a matriz 3 x 2
Fonte: Elaborado pelo autor (2022)

Agora, a demonstração da execução total:

```
Integral:  [[89, 32], [-8, 93], [12, 54]]
matriz[0][0] = 100
Integral [[100, 32], [-8, 93], [12, 54]]
```

Comando de prompt 7 – Execução da matriz com valor manipulado
Fonte: Elaborado pelo autor (2022)

Repare que a célula com o índice `[0][0]` de conteúdo `89` foi modificada para `100`.

A matriz está sendo exibida de forma bruta, mas seria melhor apresentar os elementos em formato de grade, ou seja, sendo tratados separadamente.

Ao invés de exibirmos a forma bruta, vamos exibir neste novo formato, o de grade, mas antes, vimos que, para manipularmos um vetor integralmente, criávamos um laço de 0 até o tamanho do vetor. No caso da matriz, por ter linhas e colunas maiores que um, criamos um laço de 0 até o número de LINHAS, e dentro dele, um laço de 0 até o número de COLUNAS para percorrermos a matriz integralmente.

Segue a exibição da matriz em formato de grade:

```
     # Percorrendo a matriz e exibindo o seu conteúdo
for linha in range(3):
    for coluna in range(2):
        print(f"{matriz[linha][coluna]}\t", end="")
    print()
```

Código-fonte 14 – Exibindo a matriz 3 x 2 em formato de grade
Fonte: Elaborado pelo autor (2022)

Veja a nova exibição:

```
99   32
-8   93
12   54
```

Comando de prompt 8 – Execução da matriz em formato de grade
Fonte: Elaborado pelo autor (2022)

Para exemplificarmos esse conhecimento, vamos criar uma matriz 3 x 3 e vamos **preenchê-la** com números aleatórios (entre 0 e 99), **exibir** e fazer a **somatória** dos elementos.

O programa completo com a criação dos subalgoritmos fica assim:

```
     """
Autor.....: Prof. Edson de Oliveira
Data......: 11/11/2022
Assunto...: Manipulação de matrizes utilizando subalgoritmos
Problema..: Preencher, exibir e somar os elementos de um vetor
3 x 3
"""


# ----- SUBALGORITMOS
# Este procedimento preenche a matriz com valores entre 0 e 99
def sortear_valores_matriz(m) -> None:
    for linha in range(3):
        for coluna in range(3):
            m[linha][coluna] = random.randint(0, 99)
```

```
# Este procedimento exibe o conteúdo da matriz
def exibir_matriz(m) -> None:
    for linha in range(3):
        for coluna in range(3):
            print(f"{m[linha][coluna]}\t", end="")
        print()


# Esta função calcula a somatória dos elementos da matriz
def somar_matriz(m) -> int:
    soma = 0
    for linha in range(3):
        for coluna in range(3):
            soma += m[linha][coluna]
    return soma


# ----- PROGRAMA PRINCIPAL
# Importa a biblioteca que possibilita os numeros aleatórios
import random

# Definição da matriz 3 x 3
matriz = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]
]

# chamada do procedimento que sorteia os valores da matriz
sortear_valores_matriz(matriz)

# Exibição bruta da matriz
print(f"Exibição Bruta: \n{matriz}\n")

# Exibição da matriz em grade
print(f"Exibição em grade:")
exibir_matriz(matriz)

print(f"\nSomatória = {somar_matriz(matriz)}")
```

Código-fonte 15 – Solução de manipulação da matriz 3 x 2
Fonte: Elaborado pelo autor (2022)

Primeiramente, os valores são sorteados na matriz e exibidos de forma bruta. Depois os valores são exibidos em formato de grade e, por fim, a somatória das células sorteadas.

Veja a execução:

```
Exibição Bruta:
[[22, 31, 83], [21, 47, 52], [79, 40, 29]]

Exibição em grade:
22   31   83
21   47   52
79   40   29
```

Comando de prompt 9 – Execução da manipulação da matriz
Fonte: Elaborado pelo autor (2022)

Assim como no vetor a matriz também contém valores de tipo homogêneo e tamanho definido pelo programador.

3 TRABALHANDO COM LISTA

A manipulação das listas é semelhante à manipulação de vetores no que tange ao acesso aos elementos pelo índice, mas as semelhanças param por aí.

3.1 Diferenças entre listas e vetor

Considere que, em um problema, o programador previu uma quantidade de elementos em um vetor que, no fim das contas, ficou aquém do esperado. Pelo vetor ter a quantidade de elementos fixos, isso seria um problema.

Se trocássemos os vetores pelas listas, isso não seria mais um problema.

Vejamos as diferenças entre vetores e listas:

- Vetor tem o seu conteúdo homogêneo, enquanto lista pode ter o seu conteúdo heterogêneo, ou seja, cada célula pode armazenar uma informação de tipo diferente (int, float, str, bool...).
- Vetor tem o seu tamanho fixo, definido pelo programador. Lista tem o seu tamanho dinâmico, podendo ser definido pelo usuário.

- Para tratarmos a parte dinâmica da Lista, precisamos utilizar métodos específicos (veremos em breve), enquanto, no caso do vetor, os manipulamos pela posição dos índices.

Enfim, basicamente vetor tem conceitos fixos, enquanto as listas têm tratamentos dinâmicos.

3.2 Métodos de manipulação de listas

Como dito, para manipularmos listas, utilizamos métodos específicos. Vejamos alguns deles.

3.2.1 Método list() – iniciando uma lista

Como a lista é dinâmica, é comum inicializarmos uma lista vazia.

Vejam as duas formas:

```
    # Inicializar entre colchetes
lista = []
# ou Inicializar com o método list()
lista = list()
```

Código-fonte 16 – Inicializando uma lista sem elementos
Fonte: Elaborado pelo autor (2022)

Em ambos os casos, as listas não têm valores inseridos.

3.2.2 Método append() – inserindo elementos em uma lista

Há duas formas de inserir elementos na lista: a primeira é parecida com o que fizemos com o vetor, ou seja, simplesmente inserimos os dados entre os colchetes, por exemplo:

```
    lista = [4, "ESO", 6.7, True]
```

Código-fonte 17 – Inicializando uma lista com valores
Fonte: Elaborado pelo autor (2022)

Os índices funcionam da mesma forma que vimos com vetor, assim, podemos dizer que essa lista tem quatro elementos.

Repare, no exemplo da lista acima, que as células podem ter conteúdos de tipos diferentes (diferentemente do vetor, que deve ser de apenas um tipo).

Se mandarmos exibir a lista, ficará:

```
     # Atribuindo valores na lista
lista = [4, "ESO", 6.7, True]

# Exibindo a lista bruta
print(lista)
```

Código-fonte 18 – Exibindo, de forma bruta, uma lista preenchida
Fonte: Elaborado pelo autor (2022)

A execução será:

```
[4, 'ESO', 6.7, True]
```

Comando de prompt 10 – Exibição bruta de uma lista
Fonte: Elaborado pelo autor (2022)

É possível inserirmos mais elementos na lista?

Sim! Este é um dos diferenciais das listas para vetores.

Vejam como inserimos por meio do método append:

```
     # Atribuindo valores na lista
lista = [4, "ESO", 6.7, True]

# Exibindo a lista bruta
print(lista)

# Inserindo um elemento com o método append() de forma direta
lista.append("novo elemento")

# Exibindo a lista bruta
print(lista)
```

Código-fonte 19 – Aplicando o método append
Fonte: Elaborado pelo autor (2022)

Na execução, o elemento inserido no final da lista:

```
[4, 'ESO', 6.7, True]
[4, 'ESO', 6.7, True, 'novo elemento']
```

Comando de prompt 11 – Exibição bruta da aplicação do método append
Fonte: Elaborado pelo autor (2022)

Podemos também inserir um elemento por meio de uma variável, seja de forma direta, seja pedindo a digitação de um valor na variável:

```
    # Atribuindo valores na lista
lista = [4, "ESO", 6.7, True]

# Exibindo a lista bruta
print(lista)

# Inserindo um elemento com o método append() de forma direta
lista.append("novo elemento")

# Exibindo a lista bruta
print(lista)

# Inserindo um elemento com o método append() de forma
indireta
elem = input("Digite um elemento: ")
lista.append(elem)

# Exibindo a lista bruta
print(lista)
```

Código-fonte 20 – Aplicando o método append com apoio de variáveis
Fonte: Elaborado pelo autor (2022)

Funcionou da mesma forma:

```
[4, 'ESO', 6.7, True]
[4, 'ESO', 6.7, True, 'novo elemento']
Digite um elemento: 45
[4, 'ESO', 6.7, True, 'novo elemento', '45']
```

Comando de prompt 12 – Exibição bruta da aplicação do método append com apoio de variáveis
Fonte: Elaborado pelo autor (2022)

Podemos, ainda, digitar diretamente o valor no elemento da lista.

Nesse exemplo, considere que o usuário digitará um número int:

```
    # digitando o elemento diretamente na lista
lista.append(int(input("Digite um elemento: ")))
```

Código-fonte 21 – Aplicando o método append com leitura direta

Fonte: Elaborado pelo autor (2022)

3.2.3 Concatenando listas com + e extended()

Há duas formas de agrupar duas listas: por concatenação (+) ou a estendendo (extended).

Utilizando o +, ele simplesmente concatena (junta) as listas:

```
    # Concatenando listas com +
l1 = [3, 6]
l2 = ['O', '@']
l3 = l1 + l2
print(f"l1 = {l1}")
print(f"l2 = {l2}")
print(f"l3 = {l3}")
```

Código-fonte 22 – Concatenando listas com +
Fonte: Elaborado pelo autor (2022)

Veja a execução:

```
l1 = [3, 6]
l2 = ['O', '@']
l3 = [3, 6, 'O', '@']
```

Comando de prompt 13 – Exibição bruta da concatenação de listas com +
Fonte: Elaborado pelo autor (2022)

Agora, utilizando o método extended:

```
    # extendendo listas com o método extended()
l1 = [3, 6]
l2 = ['O', '@']
l1.extend(l2)
print(f"l1 = {l1}")
print(f"l2 = {l2}")
```

Código-fonte 23 – Estendendo uma lista com o método extended
Fonte: Elaborado pelo autor (2022)

Execução:

```
l1 = [3, 6, 'O', '@']
l2 = ['O', '@']
```

Comando de prompt 14 – Exibição da lista estendida

Fonte: Elaborado pelo autor (2022)

A principal diferença entre o + e o extended() é que o primeiro pode ser adicionado em uma lista nova, enquanto o segundo precisa ser adicionado em uma lista existente.

3.2.4 Inserindo um elemento com o método insert()

Para inserirmos um elemento na lista (não necessariamente no final), usamos o método insert().

Sua sintaxe é: <lista>.insert(<índice>, <conteúdo>)

Dessa forma, temos dois parâmetros: o índice no qual será inserido o elemento e o conteúdo da célula.

Veja o exemplo:

```
    # Inserindo no índice 2 o conteúdo "Engenharia"
lista = [4, "ESO", 6.7, True]
print(lista)
lista.insert(2, "Engenharia")
print(lista)
```

Código-fonte 24 – Inserindo um elemento no meio da lista com o método insert
Fonte: Elaborado pelo autor (2022)

Execução:

```
[4, 'ESO', 6.7, True]
[4, 'ESO', 'Engenharia', 6.7, True]
```

Comando de prompt 15 – Exibição da lista com elemento inserido
Fonte: Elaborado pelo autor (2022)

Consequentemente, os elementos à direita da inserção são movidos para os índices subsequentes.

3.2.5 Removendo o último elemento da lista com o método pop()

Caso seja necessário remover algum elemento da lista, nós utilizamos o método pop(), passando como parâmetro o índice.

Veja um exemplo:

```
     # Inicializando a lista
lista = [4, "ESO", "Engenharia", 6.7, True]
print(lista)

# Remove o elemento "ESO" que tem o índice 1
lista.pop(1)
print(lista)
```

Código-fonte 25 – Removendo um elemento da lista com o método pop
Fonte: Elaborado pelo autor (2022)

Execução:

```
[4, 'ESO', 'Engenharia', 6.7, True]
[4, 'Engenharia', 6.7, True]
```

Comando de prompt 16 – Exibição da lista com elemento removido
Fonte: Elaborado pelo autor (2022)

3.2.6 Apagando todos os elementos da lista com o clear()

Se você precisar excluir todos os elementos da lista, utilize o método clear(). Veja o exemplo:

```
     # Inicializando a lista
lista = [4, "ESO", "Engenharia", 6.7, True]
print(lista)
lista.clear()
print(lista)
```

Código-fonte 26 – Apagando todos os elementos da lista com o método clear
Fonte: Elaborado pelo autor (2022)

Agora a execução:

```
[4, 'ESO', 'Engenharia', 6.7, True]
[]
```

Comando de prompt 17 – Exibição da lista com todos os elementos removidos
Fonte: Elaborado pelo autor (2022)

Repare que a lista continua existindo, mas ficou vazia.

3.2.7 Excluindo a lista com del()

O método clear() apaga os elementos e deixa a lista vazia. O método del() exclui a lista, independentemente de esta ter elementos ou não, ou seja, desaloca da memória RAM a lista (identificador):

```
     # Inicializando a lista
lista = [4, "ESO", "Engenharia", 6.7, True]
print(lista)
del(lista)
print(lista)
```

Código-fonte 27 – Apagando a lista da memória com o método del
Fonte: Elaborado pelo autor (2022)

Veja a execução:

```
[4, 'ESO', 'Engenharia', 6.7, True]
Traceback (most recent call last):
  File
"/Users/edsondeoliveira/Desktop/fontes/python/estudo/_capitulo5
/joao.py", line 48, in <module>
    print(lista)
NameError: name 'lista' is not defined. Did you mean: 'list'?
```

Comando de prompt 18 – Comprovação da remoção dos elementos da lista
Fonte: Elaborado pelo autor (2022)

Repare que, na execução, ele exibe o conteúdo da lista, e, depois de excluir a lista com del(), o programa tenta imprimir novamente a lista, mas o interpretador acusa um erro dizendo que a lista não existe mais na memória.

Vamos fazer um programa que preencha uma lista até que seja digitado ponto, e depois a exibi-la da forma bruta, clássica e “Python”.

Seguem todas as rotinas. Cada rotina terá o seu comentário.

Começando pelo preenchimento da lista:

```
     # Preenchendo a lista até que seja digitado .
def preenche_lista(l: list) -> None:
    # Solicita ao usuário a digitação de algo
    algo = input("Digite algo: ")
    # Verifica se não digitou ponto
    while algo != '.':
        # Adiciona o elemento na lista
        l.append(algo)
```

```
        # Solicita ao usuário uma nova digitação
        algo = input("Digite algo: ")
```

Código-fonte 28 – Construção da rotina preenche_lista até que seja digitado ponto
Fonte: Elaborado pelo autor (2022)

Essa rotina preenche a lista que é passada por parâmetro `l` até que o usuário digite ponto. Essa rotina mostra o dinamismo da lista porque não há um tamanho predefinido e/ou um tipo específico para ser digitado.

Veja a execução da sua chamada:

```
Digite algo: 4
Digite algo: r
Digite algo: Fiap
Digite algo: True
Digite algo: 9.8
Digite algo: .
```

Comando de prompt 19 – Execução da rotina preenche_lista
Fonte: Elaborado pelo autor (2022)

Aplicando a exibição clássica (forma 1):

```
    # Exibindo a lista da forma clássica
def exibe_lista_forma1(l: list) -> None:
    for elem in range(0, len(l), 1):
        print(l[elem])
```

Código-fonte 29 – Construção da rotina exibe_lista_formal
Fonte: Elaborado pelo autor (2022)

Esse módulo utiliza o recurso `len(l)` (para contar quantos elementos existem na lista). Dentro do `for`, são exibidos todos os elementos. Dessa forma, fica parecido com a exibição dos elementos de um vetor.

Veja a execução da sua chamada:

```
4
r
Fiap
True
9.8
```

Comando de prompt 20 – Execução da rotina exibe_lista_formal
Fonte: Elaborado pelo autor (2022)

Exibição com recursos exclusivos do Python (forma 2):

```
     # Exibindo a lista decom recursos do Python
def exibe_lista_forma2(l: list) -> None:
    for elem in l:
        print(elem)
```

Código-fonte 30 – Construção da rotina exibe_lista_formal2
Fonte: Elaborado pelo autor (2022)

Observe a linha do for: `for elem in l:` ela quer dizer algo como “Extraia elemento `elem` da lista `l` enquanto houver”. Esse recurso simplifica a aplicação do código porque, posteriormente, apenas utiliza `print(elem)` para exibir os elementos, ao invés de tratá-los pelos índices, como no exemplo anterior.

Repare que a execução é a mesma:

```
4
r
Fiap
True
9.8
```

Comando de prompt 21 – Execução da rotina exibe_lista_formal2
Fonte: Elaborado pelo autor (2022)

Segue o código-fonte integral para você treinar essa solução:

```
    """
Autor.....: Prof. Edson de Oliveira
Data......: 14/11/2022
Assunto...: Manipulando listas
Problema..: Preenche uma lista até que seja digitado ponto,
depois a exibe
"""

# ----- SUBALGORITMOS
# Preenchendo a lista até que seja digitado .
def preenche_lista(l: list) -> None:
    # Solicita ao usuário a digitação de algo
    algo = input("Digite algo: ")
    # Verifica se não digitou ponto
    while algo != '.':
        # Adiciona o elemento na lista
        l.append(algo)
        # Solicita ao usuário uma nova digitação
```

```
        algo = input("Digite algo: ")


# Exibindo a lista da forma clássica
def exibe_lista_forma1(l: list) -> None:
    for elem in range(0, len(l), 1):
        print(l[elem])


# Exibindo a lista decom recursos do Python
def exibe_lista_forma2(l: list) -> None:
    for elem in l:
        print(elem)

# ----- PROGRAMA PRINCIPAL
# inicia a lista vazia
lista = list()
# Preenche a lista até que seja digitado '.'
preenche_lista(lista)
# Exibe a lista Bruta
print(lista)
# Exibe a lista da forma 1
print("\nForma 1:")
exibe_lista_forma1(lista)
# Exibe a lista da forma 2
print("\nForma 2:")
exibe_lista_forma2(lista)
```

Código-fonte 31 – Solução preencher e exibir uma lista
Fonte: Elaborado pelo autor (2022)

Agora a execução desse código-fonte:

```
Digite algo: 34
Digite algo: FIAP
Digite algo: False
Digite algo: 4.5
Digite algo: .
['34', 'FIAP', 'False', '4.5']

Forma 1:
34
FIAP
False
4.5

Forma 2:
34
FIAP
False
4.5
```

Comando de prompt 22 – Execução da solução preencher e exibir uma lista
Fonte: Elaborado pelo autor (2022)

Concluímos que a lista é dinâmica e o seu conteúdo pode ter elementos de quaisquer tipos, facilitando o armazenamento dos dados.

4 TRABALHANDO COM TUPLA

Todas as estruturas vistas até então (vetor, matriz e listas) têm os seus elementos editáveis. A tupla se encaixa nesse contexto? Não!

4.1 Definição de tupla

Em todas as necessidades vistas até agora com relação às estruturas de dados, notamos que, por algum motivo ou em algum momento, os usuários puderam modificar o conteúdo dessas estruturas.

Existem situações em programação nas quais precisamos de estruturas de dados que devem ter as suas células inalteráveis, é nesse momento que entra a tupla.

Tuplas são estruturas de dados que têm células fixas, ou seja, somente o programador pode definir o seu conteúdo, logo o usuário não possui essa autonomia.

A lista tem uma dupla de colchetes para delimitar os seus elementos. A tupla utiliza uma dupla de parênteses. Sendo assim, podemos inicializar uma tupla com o comando: `tupla=()`. Ou podemos utilizar, também, `tupla = tuple()`. Em ambos os casos é criada uma tupla vazia, o que não faz muito sentido, porque uma tupla não pode ser alimentada.

A diferença é que não conseguimos criar rotinas para preencher essa tupla, e sim atribuir diretamente:

```
tupla = (12, 56, "@", True)
```

Código-fonte 32 – Iniciando uma tupla com valores
Fonte: Elaborado pelo autor (2022)

Se mandarmos printar na tela, o resultado será:

```
(12, 56, '@', True)
```

Comando de prompt 23 – Exibição dos elementos da tupla
Fonte: Elaborado pelo autor (2022)

Não podemos usar strings em uma tupla porque as strings são iteráveis. Em outras palavras, uma string é uma estrutura de dados que pode ter o seu conteúdo modificado.

Vamos exemplificar esse conteúdo?

Algo comum para nós que recebemos salário é pagar o imposto de renda (IR). Vamos criar uma solução que peça um salário ao usuário e retorne quanto ele pagará de IR. Ah! Utilizando tuplas.

Primeiramente, considere esta tabela do IR 2022:

Tabela de alíquota IRPF a deduzir

Conforme o salário, você confere em qual faixa de contribuição se encaixa.

BASE DO CÁLCULO (R$)	ALÍQUOTA	PARCELA A DEDUZIR DO IRPF (R$)
Até R$1.903,98	-	-
De R$1.903,99 a R$2.826,65	7,5%	R$142,80
De R$2.826,66 a R$23.751,05	15%	R$354,80
De R$3.751,06 a R$4.664,68	22,5%	R$636,13
Acima de R$4.664,69	27,5%	R$869,36
Parcela por dependente	R$189,59	

Figura 5 – Tabela das alíquotas do Imposto de Renda
Fonte: site Leoa (2022)

O salário de todos nós, contribuintes, se encaixa em uma das cinco faixas. Na primeira, há isenção; nas demais, é cobrado um percentual do salário bruto do trabalhador. Esse imposto calculado tem parte deduzida do arrecadado.

Vamos construir os módulos dessa solução?

Primeiramente, vamos criar três tuplas: uma para a base de cálculo, outra para a alíquota e a última para a dedução:

```
    # --------------- DEFINIÇÃO DAS TUPLAS -------------------
--
# FAIXA              1     2         3         4         5
    # ÍNDICE               0     1         2         3         4
tuplaBaseCalculo = (0, 1903.98, 2826.65, 3751.05, 4664.68)
tuplaAliquota    = (0,    7.50,   15.00,   22.50,   27.50)
tuplaDeducao     = (0,  142.80,  354.80,  636.13,  869.36)
```

Código-fonte 33 – Criando três tuplas para base dos cálculos
Fonte: Elaborado pelo autor (2022)

Agora, vamos criar a função `retornaFaixa`, que vai percorrer a tupla e retornar o índice da faixa na qual o salário informado por parâmetro se enquadra:

```
     # Retorna a faixa do IR correspondente
def retornaFaixa(tbc: tuple, sal: float) -> int:
    for faixa in range(len(tbc) - 1, -1, -1):
        if sal > tbc[faixa]:
            return faixa
            break
    return 0
```

Código-fonte 34 – Construção da função retornaFaixa
Fonte: Elaborado pelo autor (2022)

Nessa rotina, é passada como parâmetro a tupla `tbc`, que contém a base de cálculo e o salário `sal`. A rotina, então, percorre os índices em ordem decrescente até achar qual conteúdo está acima do salário informado como parâmetro, retornando assim o índice correspondente à faixa.

Fica fácil, agora, calcularmos o Imposto de Renda. Vamos então criar a rotina `calculoIR`. Devemos alimentar os parâmetros e efetuar o cálculo subtraindo a eventual dedução:

```
     # Efetua o cálculo do IR
def calculoIr(f: int, sal: float, ta: tuple, td: tuple) ->
float:
    return sal * (ta[f] / 100) - td[f]
```

Código-fonte 35 – Construção da função calculoIr
Fonte: Elaborado pelo autor (2022)

O parâmetro `f` representa o índice (faixa) no qual o salário se encaixou, `sal` é o salário que servirá como base de cálculo, `ta` é a tupla contendo a alíquota que será cobrada e `td` é a tupla contendo a dedução.

Como tudo isso vai funcionar?

Segue o código-fonte completo:

```
    """
Autor.....: Prof. Edson de Oliveira
Data......: 14/11/2022
Assunto...: Manipulando tuplas
Problema..: Cálculo do Imposto de Renda
"""
    # ----- DEFINIÇÃO DOS SUBALGORITMOS ---------------------
# Retorna a faixa do IR correspondente
def retornaFaixa(tbc: tuple, sal: float) -> int:
    for faixa in range(len(tbc) - 1, -1, -1):
        if sal > tbc[faixa]:
            return faixa
            break
    return 0


# Efetua o cálculo do IR
def calculoIr(f: int, sal: float, ta: tuple, td: tuple) ->
float:
    return sal * (ta[f] / 100) - td[f]


# ----- DEFINIÇÃO DAS TUPLAS --------------------
# FAIXA             1       2         3         4         5
tuplaBaseCalculo = (0, 1903.98, 2826.65, 3751.05, 4664.68)
tuplaAliquota    = (0, 7.5    , 15     , 22.5   , 27.5)
tuplaDeducao     = (0, 142.80 , 354.80 , 636.13 , 869.36)

# ----- PROGRAMA PRINCIPAL --------------------
salario = float(input("Salario: "))
faixa = retornaFaixa(tuplaBaseCalculo, salario)
print("IR = ",calculoIr(faixa, salario, tuplaAliquota,
tuplaDeducao))
```

Código-fonte 36 – Solução integral do cálculo do IR
Fonte: Elaborado pelo autor (2022)

Execução da solução com salários variados:

```
# Digitando 1000.00
Salario: 1000
IR =  0.0

# Digitando 2500.00
Salario: 2500
IR =  44.69999999999999

# Digitando 3500.00
Salario: 3500
IR =  170.2

# Digitando 4500.00
Salario: 4500
IR =  376.37

# Digitando 10000.00
Salario: 10000
IR =  1880.6399999999999
```

Comando de prompt 24 – Exibição da solução do cálculo do IR
Fonte: Elaborado pelo autor (2022)

5 TRABALHANDO COM DICIONÁRIO

Todas as estruturas vistas até o momento (vetor, matriz, listas e tuplas) têm em comum o seu manuseio pelo índice. Dicionário também trabalha com índices? Não!

5.1 Definição de dicionários

Enquanto as estruturas anteriores eram baseadas em índices, os dicionários usam keys em vez de índices. Dessa forma, o dicionário se parece mais com um registro de uma tabela de banco de dados, porém criado na memória.

Veja a estrutura de um dicionário:

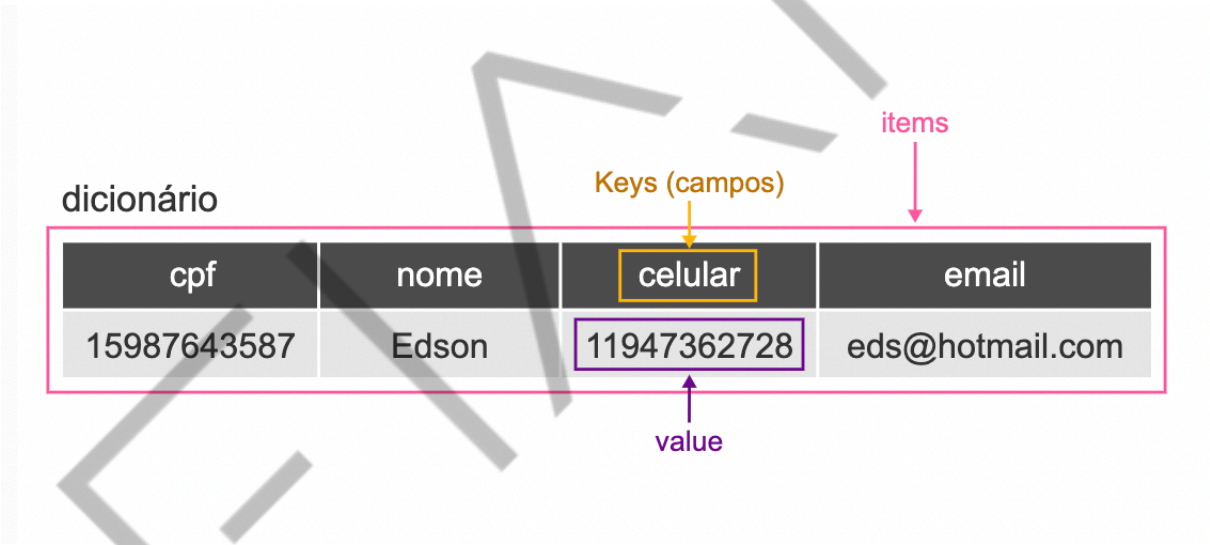

Figura 6 – Componentes de um dicionário
Fonte: Elaborado pelo autor (2022)

O dicionário possui **keys** (também conhecidas como campos), que nada mais são do que as identificações das colunas. As keys substituem os índices das outras estruturas de dados.

Value é o valor de uma célula do registro; dessa forma, values são todos os valores contidos no registro.

Items são todos os pares de keys/values de um registro.

5.2 Criando dicionários

Os colchetes delimitam os valores de uma lista; e os parênteses, os valores de uma tupla. Há algum par de caracteres específicos para dicionários? Sim! Os dicionários são delimitados com um par de chaves.

5.2.1 Criando um dicionário vazio

Há duas formas de criarmos um dicionário vazio.

A primeira é simplesmente abrindo e fechando chaves: `dicionario = {}`, ou podemos utilizar o método `dict()` da seguinte forma: `dicionario = dict()`.

Em ambos os casos, o dicionário é criado vazio.

5.2.2 Inicializando um dicionário com valores

Assim como nas estruturas de dados anteriores, conseguimos inicializar um dicionário com valores. Vamos criar um dicionário chamado `contato`, com valores predefinidos:

```
     contato = {
    'cpf':2345678900,
    'nome':'Edson de Oliveira',
    'celular':'1194837363',
    'email':'profedson.oliveira@fiap.com.br',
}
```

Código-fonte 37 – Construção de uma lista com valores iniciais
Fonte: Elaborado pelo autor (2022)

Contato é o nome do dicionário. Entre chaves estará toda a estrutura com valores predefinidos. Todos os **items** (pares de keys e values) são separados por vírgula. Todas as keys ficam entre apóstrofos (ou aspas). O **value** das **keys** fica também entre apóstrofos se forem strings; caso contrário, respeita o tipo da key.

Se dermos um print na estrutura acima, ela ficará: `print(contato)`.

O resultado será uma exibição bruta:

```
{'cpf': 2345678900, 'nome': 'Edson de Oliveira', 'celular':
'1194837363', 'email': 'profedson.oliveira@fiap.com.br'}
```

Comando de prompt 25 – Exibição bruta do conteúdo de um dicionário
Fonte: Elaborado pelo autor (2022)

Também podemos formatar esta exibição:

```
    print("EXIBINDO O DICIONÁRIO")
print("CPF...........: ", contato['cpf'])
print("Nome..........: ", contato['nome'])
print("Celular.......: ", contato['celular'])
print("E-mail........: ", contato['email'])
```

Código-fonte 38 – Construção da exibição do dicionário com os elementos separados
Fonte: Elaborado pelo autor (2022)

O resultado ficará com melhor exibição:

```
EXIBINDO O DICIONÁRIO
CPF...........:  2345678900
Nome..........:  Edson de Oliveira
Celular.......:  1194837363
E-mail........:  profedson.oliveira@fiap.com.br
```

Comando de prompt 26 – Exibição do dicionário formatado
Fonte: Elaborado pelo autor (2022)

5.2.3 Preenchendo um dicionário

Como nas demais estruturas (exceto a tupla), o usuário pode inserir os dados via teclado.

Veja:

```
    # cria um dicionário de nome 'contato' vazio
contato = dict()

# permite ao usuário preencher o dicionário
print("PREENCHENDO O DICIONÁRIO")
contato['cpf'] = input("CPF...........:")
contato['nome'] = input("Nome..........:")
contato['celular'] = input("Celular.......:")
contato['email'] = input("E-mail........:")
```

Código-fonte 39 – Digitação dos elementos pelo usuário
Fonte: Elaborado pelo autor (2022)

As keys não precisaram ser criadas. Nós simplesmente as colocamos entre aspas dentro dos colchetes e é entendido que é uma key. Se já existir, ele a considera; se não existir, ele a cria.

Veja a execução:

```
PREENCHENDO O DICIONÁRIO
CPF...........: 12345678912
Nome..........: Vania da Silva
Celular.......: 11948372334
E-mail........: vania@hotmail.com
```

Comando de prompt 27 – Exibição do preenchimento do dicionário formatado
Fonte: Elaborado pelo autor (2022)

Nesse caso, o usuário digitou estas informações.

Se repetirmos os comandos:

```
    print("EXIBINDO O DICIONÁRIO")
print("CPF...........: ", contato['cpf'])
print("Nome..........: ", contato['nome'])
print("Celular.......: ", contato['celular'])
print("E-mail........: ", contato['email'])
```

Código-fonte 40 – Exibição dos elementos digitados pelo usuário
Fonte: Elaborado pelo autor (2022)

O resultado será:

```
EXIBINDO O DICIONÁRIO
CPF...........: 12345678912
Nome..........: Vania da Silva
Celular.......: 11948372334
E-mail........: vania@hotmail.com
```

Comando de prompt 28 – Exibição do preenchimento do dicionário
Fonte: Elaborado pelo autor (2022)

6 TABELAS (LISTA COM DICIONÁRIO)

Quando utilizamos um dicionário, vimos que foi possível preencher uma entidade por meio de suas keys. Caso precisássemos digitar um segundo registro na mesma variável do tipo dicionário, ele sobreporia o anterior.

É possível resolver isso? Sim! Criando lista de dicionários.

6.1 Lista de dicionários

Lista de dicionário nada mais é do que a simulação de uma tabela (mas de memória). Já que listas têm índices e células, e dicionários têm keys e values, então unimos os dois para que cada entidade cadastrada seja independente.

Vamos explicar graficamente. Considere um dicionário criado na memória:

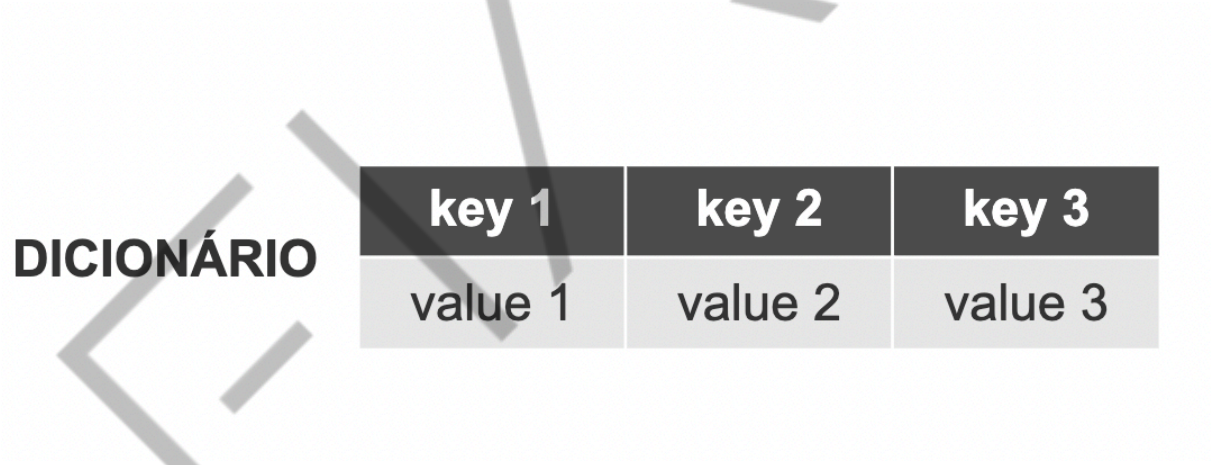

Figura 7 – Estrutura de um dicionário
Fonte: Elaborado pelo autor (2022)

O dicionário terá um nome, suas keys (campos) e seus values (valores). Agora, imagine uma lista criada na memória:

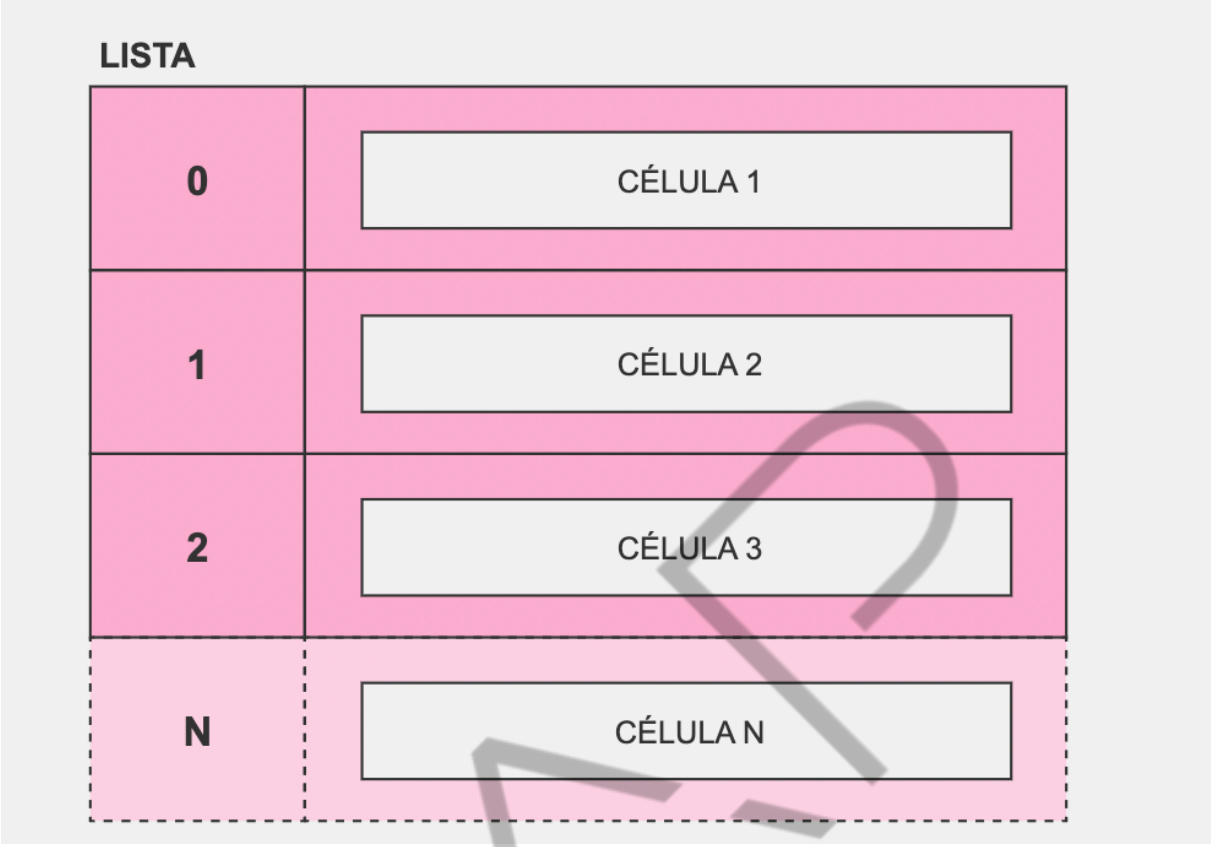

Figura 8 – Estrutura de uma lista
Fonte: Elaborado pelo autor (2022)

A lista é composta de elementos; e dentro de cada elemento, há um índice e uma célula (conteúdo).

Que tal unirmos os dois? Vamos criar uma lista cuja célula (conteúdo) é um dicionário:

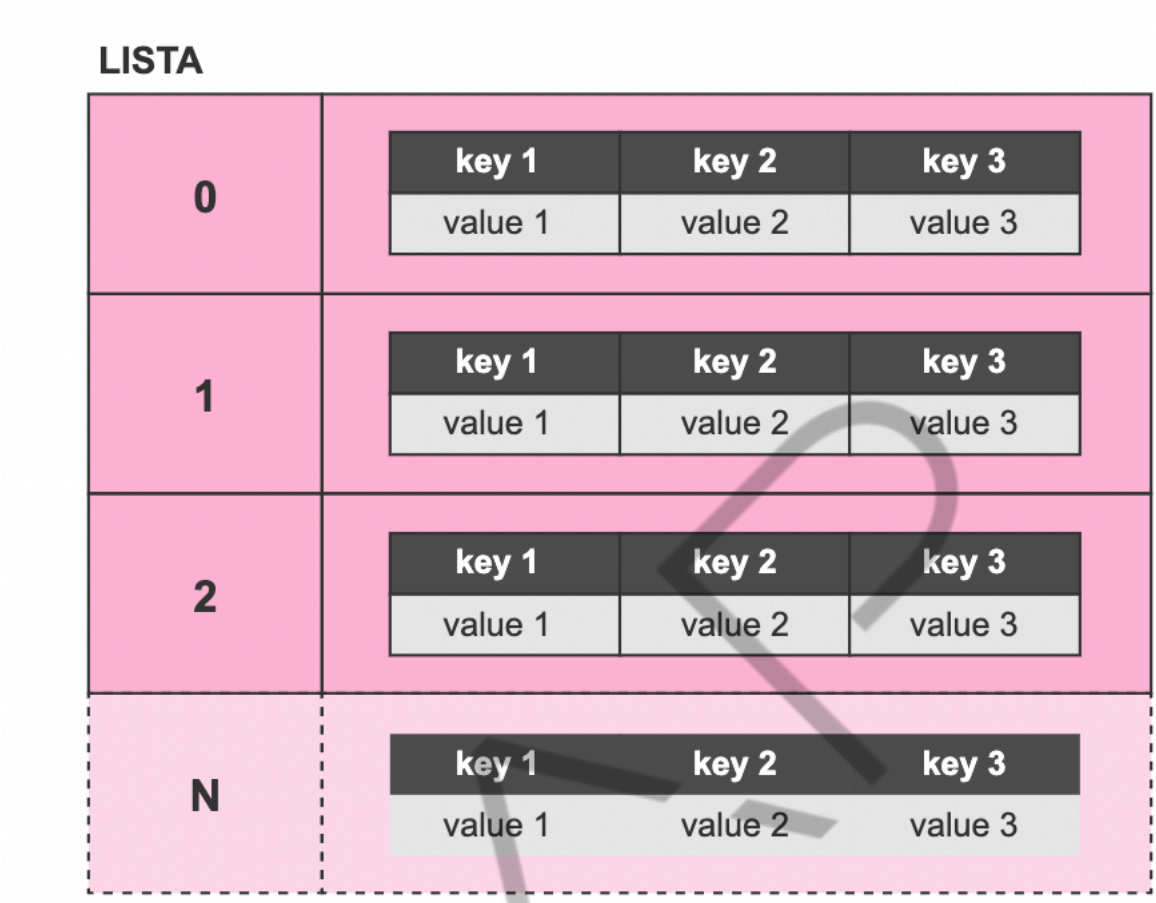

Figura 9 – Combinação da lista com o dicionário
Fonte: Elaborado pelo autor (2022)

Utilizando esse conceito, conseguimos criar uma “tabela” na qual cada campo e tabela podem ser armazenados e acessados individualmente.

6.2 Cadastrando contatos

Chegou o momento de colocarmos toda essa teoria em prática.

Vamos considerar um registro com os campos:

Estrutura do dicionário CONTATO	
Key	Tipo
Cpf	Int
Nome	Str
Celular	Str
Email	Str

Tabela 1 – Estrutura do registro CONTATO
Fonte: Elaborado pelo autor (2022)

E uma lista que representará a nossa tabela com o nome TABELA.

Esta será a sugestão do menu:

```
0 - SAIR
1 - CADASTRAR UM CONTATO
2 - EXIBIR OS CONTATOS CADASTRADOS
3 - EXIBIR UM CONTATO ESPECÍFICO


Digite uma opção desejada...:
```

Comando de prompt 29 – Sugestão da solução – CONTATOS
Fonte: Elaborado pelo autor (2022)

A opção 0 sairá do sistema.

A opção 1 cadastrará um novo contato.

A opção 2 exibirá todos os contatos cadastrados.

A opção 3 exibirá um contato específico a partir de um índice fornecido.

Vamos comentar a construção de cada rotina, começando pela criação das estruturas de dados:

```
    # Cria uma 'Tabela' que armazenará um dicionário
tabela = list()
# cria um dicionário de nome 'contato' vazio
contato = dict()
```

Código-fonte 41 – Inicialização da tabela (lista x dicionário)
Fonte: Elaborado pelo autor (2022)

Simples assim! Uma lista que representa a TABELA e um dicionário que representa o CONTATO.

Agora, mostraremos a rotina que cadastra os contatos:

```
    # Procedimento que preenche um registro
def preenche_registro(t: list, reg: dict) -> None:
    # permite ao usuário preencher o dicionário
    print("PREENCHENDO O REGISTRO")
    reg['cpf'] = input("CPF...........: ")
    reg['nome'] = input("Nome..........: ")
    reg['celular'] = input("Celular.......: ")
    reg['email'] = input("E-mail........: ")
    # cria uma cópia do ponteiro
    t.append(reg.copy())
```

Código-fonte 42 – Construção do procedimento preenche_registro
Fonte: Elaborado pelo autor (2022)

A nossa rotina se chama `preenche_registro` e é um procedimento. Ela passará como parâmetros `t`, que representa a tabela, e `reg`, que representa o registro do contato. A rotina permite que o `reg` seja preenchido pelo usuário, e quando vai dar um `append` em `t`, há um `copy()` em reg.

O método `copy()` permite criar outra posição na memória, ou seja, não utiliza o ponteiro da estrutura inicial.

E para exibir um registro específico?

```
    # procedimento que exibe um registro especifico
def exibe_registro(t: list, i: int) -> None:
    print(f"REGISTRO {i}:")
    print("CPF.......:" + str(t[i]['cpf']))
    print("Nome......:" + t[i]['nome'])
    print("Celular...:" + t[i]['celular'])
    print("E-mail....:" + t[i]['email'])
    print()
```

Código-fonte 43 – Construção do procedimento exibe_registro
Fonte: Elaborado pelo autor (2022)

Passamos por parâmetro a tabela `t` e o índice `i` que desejamos exibir. Combinamos a tabela `t` com o índice e key para exibir o registro completo. Vale ressaltar que esse procedimento exibe somente um registro.

Agora vem a rotina que exibe todos os registros da tabela:

```
    # Procedimento que exibe todos os registros da tabela
def exibe_tabela(t: list) -> None:
    qtd_registros = len(t)
    for indice in range(qtd_registros):
        exibe_registro(t, indice)
```

Código-fonte 44 – Construção do procedimento exibe_tabela
Fonte: Elaborado pelo autor (2022)

A tabela `t` é passada como parâmetro. Dentro do procedimento, primeiramente é contada a quantidade de registros e essa informação é armazenada na variável `qtd_registros`. Dessa forma, o laço percorre todos os registros, a partir do índice 0 e a cada volta é invocado o procedimento `exibe_registro` (demonstrado anteriormente) que exibe os registros pelo índice.

Segue agora a solução completa para você testar:

```
    '''
Autor.....: Prof. Edson de Oliveira
Data......: 14/11/2022
Assunto...: Utilizando listas x dicionários
Problema..: Cadastrando e exibindo registros
'''


# Procedimento que preenche um registro
def preenche_registro(t: list, reg: dict) -> None:
    # permite ao usuário preencher o dicionário
    print("PREENCHENDO O REGISTRO")
    reg['cpf'] = input("CPF...........: ")
    reg['nome'] = input("Nome..........: ")
    reg['celular'] = input("Celular.......: ")
    reg['email'] = input("E-mail........: ")
    # cria uma cópia do ponteiro
    t.append(reg.copy())


# procedimento que exibe um registro especifico
def exibe_registro(t: list, i: int) -> None:
    print(f"REGISTRO {i}:")
    print("CPF.......:" + str(t[i]['cpf']))
    print("Nome......:" + t[i]['nome'])
    print("Celular...:" + t[i]['celular'])
    print("E-mail....:" + t[i]['email'])
    print()


# Procedimento que exibe todos os registros da tabela
def exibe_tabela(t: list) -> None:
    qtd_registros = len(t)
    for indice in range(qtd_registros):
        exibe_registro(t, indice)


# ----- PROGRAMA PRINCIPAL
# Cria uma 'Tabela' que armazenará um dicionário
tabela = list()
# cria um dicionário de nome 'contato' vazio
contato = dict()

while True:
    print("""
0 - SAIR
1 - CADASTRAR UM CONTATO
2 - EXIBIR OS CONTATOS CADASTRADOS
3 - EXIBIR UM CONTATO ESPECÍFICO
    """)
    opcao = int(input("Digite uma opção desejada...: "))
```

```
match opcao:
    case 0:
        break
    case 1:
        preenche_registro(tabela, contato)
    case 2:
        exibe_tabela(tabela)
    case 3:
        tot_reg = len(tabela)
        ind = int(input("Digite o índice...: "))
        if ind > tot_reg - 1 or ind < 0:
            print("Índice inexistente!")
        else:
            exibe_registro(tabela, ind)
    case _:
        print("Opção inválida!")
```

Código-fonte 45: Construção integral da solução CONTATOS
Fonte: Elaborado pelo autor (2022)

Confira a simulação da execução.

Cadastrando os registros:

```
0 - SAIR
1 - CADASTRAR UM CONTATO
2 - EXIBIR OS CONTATOS CADASTRADOS
3 - EXIBIR UM CONTATO ESPECÍFICO

Digite uma opção desejada...: 1
PREENCHENDO O REGISTRO
CPF...........: 12345654398
Nome..........: Edson de Oliveira
Celular.......: 11947363388
E-mail........: profedson.oliveira@fiap.com.br

0 - SAIR
1 - CADASTRAR UM CONTATO
2 - EXIBIR OS CONTATOS CADASTRADOS
3 - EXIBIR UM CONTATO ESPECÍFICO

Digite uma opção desejada...: 1
PREENCHENDO O REGISTRO
CPF...........: 23432388898
Nome..........: Vanderley Nogueira
Celular.......: 1294837363
E-mail........: vand@hotmail.com

0 - SAIR
1 - CADASTRAR UM CONTATO
2 - EXIBIR OS CONTATOS CADASTRADOS
```

```
     3 - EXIBIR UM CONTATO ESPECÍFICO

     Digite uma opção desejada...: 1
     PREENCHENDO O REGISTRO
     CPF...........: 5433736344
     Nome..........: Estela Silva
     Celular.......: 21948473737
     E-mail........: estela@hotmail.com
```

Comando de prompt 30 – Exibição do preenchimento dos contatos
Fonte: Elaborado pelo autor (2022)

Exibindo os registros:

```
0 - SAIR
1 - CADASTRAR UM CONTATO
2 - EXIBIR OS CONTATOS CADASTRADOS
3 - EXIBIR UM CONTATO ESPECÍFICO

Digite uma opção desejada...: 2
REGISTRO 0:
CPF.......:12345654398
Nome......:Edson de Oliveira
Celular...:11947363388
E-mail....:profedson.oliveira@fiap.com.br

REGISTRO 1:
CPF.......:23432388898
Nome......:Vanderley Nogueira
Celular...:1294837363
E-mail....:vand@hotmail.com

REGISTRO 2:
CPF.......:543373644
Nome......:Estela Silva
Celular...:21948473737
E-mail....:estela@hotmail.com
```

Comando de prompt 31 – Exibição de todos os contatos
Fonte: Elaborado pelo autor (2022)

Exibindo um registro específico:

```
0 - SAIR
1 - CADASTRAR UM CONTATO
2 - EXIBIR OS CONTATOS CADASTRADOS
3 - EXIBIR UM CONTATO ESPECÍFICO

Digite uma opção desejada...: 3
Digite o índice...: 1
REGISTRO 1:
CPF.......:23432388898
Nome......:Vanderley Nogueira
Celular...:1294837363
E-mail....:vand@hotmail.com



0 - SAIR
1 - CADASTRAR UM CONTATO
2 - EXIBIR OS CONTATOS CADASTRADOS
3 - EXIBIR UM CONTATO ESPECÍFICO

Digite uma opção desejada...: 0
```

Comando de prompt 32 – Exibição de um contato específico
Fonte: Elaborado pelo autor (2022)

Com a construção dessa solução, pudemos contemplar praticamente todos os conceitos trabalhados neste capítulo.

GLOSSÁRIO

Índice	Número inteiro que representa uma célula específica.
Key	Chave/campo de um registro.
Value	Conteúdo de uma key.
Items	Combinação dos pares key/value.
Método	Comando específico de um objeto.
Objeto	Neste capítulo, são as estruturas de dados aprendidas.
Dicionário	Termo em Python para definir uma estrutura de dados.