Bruno Felipe de Souza Araujo

3º Semestre CDN - Fatec Votorantim

```python
from faker import Faker

fake = Faker('pt_BR')
clientes = {}

for _ in range(5):
    nome = fake.name()
    dados = {
        'CPF': fake.cpf(),
        'Email': fake.email(),
        'Endereço': fake.address(),
        'Data de Nascimento': fake.date_of_birth(minimum_age=18, maximum_age=65)
    }
    clientes [nome] = dados

# Exibir os dados
for nome, info in clientes.items():
    print(f"Cliente: {nome}")
    for chave, valor in info.items():
        print(f" {chave}: {valor}")
    print("-" * 40)
```

✓ 2.9s

```
Cliente: Leonardo da Luz
 CPF: 376.401.852-62
 Email: barbara06@example.org
 Endereço: Viela Emilly Albuquerque
Conjunto Califórnia Ii
15630921 Fonseca de Alves / PR
 Data de Nascimento: 1972-09-03
----------------------------------------
Cliente: Alexia Souza
 CPF: 806.971.423-96
 Email: pandrade@example.com
 Endereço: Vila Leão, 86
Vila São João Batista
76718356 Ferreira do Oeste / TO
 Data de Nascimento: 2001-09-06
----------------------------------------
Cliente: Rafaela Guerra
 CPF: 639.174.580-39
 Email: mourafernando@example.net
 Endereço: Esplanada Vieira, 7
Glória
20916689 Brito da Serra / MA
 Data de Nascimento: 1987-09-03
----------------------------------------
Cliente: Isaac Peixoto
 CPF: 047.128.695-85
 Email: stella35@example.com
 Endereço: Estrada Alícia Camargo
Vila Sesc
97358-392 Viana da Mata / MT
 Data de Nascimento: 2003-06-23
----------------------------------------
Cliente: Maria Clara Mendonça
 CPF: 316.048.257-07
 Email: emanuelfernandes@example.net
 Endereço: Ladeira Leão, 94
Solar Do Barreiro
36603595 da Cunha / AP
 Data de Nascimento: 2001-03-11
----------------------------------------
```

2. Dicionários em Python

- O que acontece se você tentar acessar uma chave que não existe?
  R: Ao tentar acessar uma chave que não existe em dicionário em python gera um erro KeyError.

- É possível adicionar uma nova chave ao dicionário depois de criado?
  R: Sim é possivel adicionar uma nova chave ao dicionário depois de criado, e será demonstrado na sequencia.

```
print("1. Tentando acessar uma chave inexistente diretamente (gera KeyError):")
try:
    print(clientes['João'])  # chave que não existe
except KeyError as e:
    print(f"KeyError: {e}")

print("\n2. Formas seguras de acessar chaves:")
# Usando get() (retorna None ou valor padrão)
print("Usando get():", clientes.get('João'))
print("Usando get() com valor padrão:", clientes.get('João', 'Cliente não encontrado'))

# Usando in para verificar existência
print("\n3. Verificando se a chave existe antes de acessar:")
cliente_busca = 'João'
if cliente_busca in clientes:
    print(f"Cliente {cliente_busca} encontrado")
else:
    print(f"Cliente {cliente_busca} não encontrado")
```

✓ 0.0s

```
1. Tentando acessar uma chave inexistente diretamente (gera KeyError):
KeyError: 'João'

2. Formas seguras de acessar chaves:
Usando get(): None
Usando get() com valor padrão: Cliente não encontrado

3. Verificando se a chave existe antes de acessar:
Cliente João não encontrado
```

```
novo_cliente = "Ana Silva"
novos_dados = {
    'CPF': fake.cpf(),
    'Email': fake.email(),
    'Endereço': fake.address(),
    'Data de Nascimento': fake.date_of_birth(minimum_age=18, maximum_age=65),
    'Telefone': fake.phone_number()
}
clientes[novo_cliente] = novos_dados

for nome in clientes:
    if 'Telefone' not in clientes[nome]:
        clientes[nome]['Telefone'] = fake.phone_number()

print("Cliente novo adicionado:")
print(f"Nome: {novo_cliente}")
for chave, valor in clientes[novo_cliente].items():
    print(f"{chave}: {valor}")
```

✓ 0.0s

```
Cliente novo adicionado:
Nome: Ana Silva
CPF: 275.639.804-74
Email: esousa@example.net
Endereço: Núcleo Silva, 214
Diamante
84457428 Gonçalves Grande / PA
Data de Nascimento: 1978-08-23
Telefone: +55 (071) 0846 1429
```

```python
coordenadas = (3.5, 7.2, 0.0)
print("Tupla:", coordenadas)
try:
    coordenadas[0] = 10
except TypeError as e:
    print("Erro ao tentar alterar a tupla:", e)
print("Segundo valor (índice 1):", coordenadas [1])
print("Existe o valor 7.2?", 7.2 in coordenadas)
PI = (3.14159,)
RGB_BRANCO = (255, 255, 255)
dias_da_semana = ('segunda', 'terça', 'quarta', 'quinta', 'sexta', 'sábado', 'domingo')
cidades = {
    ('São Paulo', 'SP'): 12_000_000,
    ('Rio de Janeiro', 'RJ'): 6_700_000
}
print("População de São Paulo:", cidades [('São Paulo', 'SP')])
```

✓ 0.0s

```
Tupla: (3.5, 7.2, 0.0)
Erro ao tentar alterar a tupla: 'tuple' object does not support item assignment
Segundo valor (índice 1): 7.2
Existe o valor 7.2? True
População de São Paulo: 12000000
```

3. Tuplas

- O que acontece se você tentar alterar um valor de uma tupla?
  R: Erro ao tentar alterar a tupla: 'tuple' object does not support item assignment.

- Quando seria melhor usar uma tupla em vez de uma lista?
  R: Seria melhor usar tupla ao invés de uma lista quando:
  1. Os dados não devem mudar (imutabilidade)
     Tuplas são imutáveis, ou seja, não podem ser alteradas depois de criadas. Isso é útil quando você quer garantir que os dados não sejam modificados por engano.
  2. Você quer usar como chave em um dicionário ou colocar em um set
     Só objetos imutáveis podem ser usados como chave em um dicionário ou dentro de um set.
  3. A estrutura representa um "registro" fixo de dados
     Tuplas funcionam bem como uma estrutura leve de dados, parecida com um registro.
  4. Performance é uma preocupação
     Tuplas são levemente mais rápidas e ocupam menos memória que listas.

- Como se acessa os elementos de uma tupla?
  R: Acessando por índice

```python
pessoa = ("João", 30, "Engenheiro")
print(pessoa[0])  # "João"
print(pessoa[1])  # 30
print(pessoa[2])  # "Engenheiro"
```

Acessando com índice negativo (do fim pro começo)

```python
print(pessoa[-1])  # "Engenheiro"
print(pessoa[-2])  # 30
```

Iterando com for

```python
for item in pessoa:
print(item)
```

Desempacotamento

```python
nome, idade, profissao = pessoa
print(nome)      # João
print(idade)     # 30
print(profissao) # Engenheiro
```

```
import numpy as np

numeros = np.array([2,3,5,7,11])
print(type(numeros))
print(numeros)
```

✓ 5.2s

```
<class 'numpy.ndarray'>
[ 2  3  5  7 11]
```

```
np.array([[1,2,3], [4,5,6]])
inteiros = np.array([[1,2,3], [4,5,6]])
print(inteiros)
reais = np.array([0.0,0.1,0.3,0.4])
print(reais)

inteiros.dtype
reais.dtype

print(inteiros.ndim)
print(reais.ndim)
print(inteiros.shape)
print(reais.shape)

print(inteiros.size)
print(inteiros.itemsize)
print(reais.size)
print(reais.itemsize)

for row in inteiros:
    for column in row:
        print(column, end= "")
    print()
```

✓ 0.0s

```
[[1 2 3]
 [4 5 6]]
[0.  0.1 0.3 0.4]
2
1
(2, 3)
(4,)
6
8
4
8
123
456
```

```
np.zeros(5)
```

✓ 0.0s

```
array([0., 0., 0., 0., 0.])
```

```
np.ones((2,4), dtype=int)
```

✓ 0.0s

```
array([[1, 1, 1, 1],
       [1, 1, 1, 1]])
```

```
np.full((3,5), 13)
```

✓ 0.0s

```
array([[13, 13, 13, 13, 13],
       [13, 13, 13, 13, 13],
       [13, 13, 13, 13, 13]])
```

```
np.linspace(0.0,1.0,num=5)
```

✓ 0.0s

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

```
np.arange(1,21).reshape(4,5)
```

✓ 0.0s

```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20]])
```

```
np.arange(1,100001).reshape(4,25000)
```

✓ 0.0s

```
array([[     1,      2,      3, ...,  24998,  24999,  25000],
       [ 25001,  25002,  25003, ...,  49998,  49999,  50000],
       [ 50001,  50002,  50003, ...,  74998,  74999,  75000],
       [ 75001,  75002,  75003, ...,  99998,  99999, 100000]],
      shape=(4, 25000))
```

```
np.arange(1,100001).reshape(100,1000)
```

✓ 0.0s

```
array([[     1,      2,      3, ...,    998,    999,   1000],
       [  1001,   1002,   1003, ...,   1998,   1999,   2000],
       [  2001,   2002,   2003, ...,   2998,   2999,   3000],
       ...,
       [ 97001,  97002,  97003, ...,  97998,  97999,  98000],
       [ 98001,  98002,  98003, ...,  98998,  98999,  99000],
       [ 99001,  99002,  99003, ...,  99998,  99999, 100000]],
      shape=(100, 1000))
```

4. Arrays NumPy

- Qual é o tipo de dados armazenado em um array? Como verificar isso?
  R: O tipo de dados em um array NumPy é controlado pelo atributo .dtype

```
import numpy as np
arr = np.array([1, 2, 3])
print(arr.dtype)  # int64, por exemplo
```

- Todos os elementos têm o mesmo tipo?
  R: Sim, todos os elementos de um array NumPy têm o mesmo tipo.
  Isso é o que diferencia ele de uma lista Python

- Qual a diferença entre arange() e linspace()?
  R: np.arange(início, fim, passo)
  Cria uma sequência com espaçamento fixo (como range())
  np.arange(0, 5, 1) → [0, 1, 2, 3, 4]
  np.linspace(início, fim, num)
  Cria um array com quantidade fixa de elementos igualmente espaçados
  np.linspace(0, 5, 5) → [0. , 1.25, 2.5, 3.75, 5.]

  arange() quando se sabe o passo.
  linspace() quando se sabe o número de pontos que quer.

- Para que serve a função reshape()?
  R: A função reshape() muda a forma (shape) de um array sem alterar os dados.

```
a = np.arange(6)  # [0 1 2 3 4 5]
b = a.reshape((2, 3))
print(b)
# [[0 1 2]
#  [3 4 5]]
```

```
import random
import time

start = time.time()
rolls_list = [random.randint(1, 6) for _ in range(6_000_000)]
end = time.time()
print(f"Tempo: {end - start:.4f} segundos")
```

✓ 4.9s

```
Tempo: 4.9500 segundos
```

```
import numpy as np
import time

start = time.time()
rolls_array = np.random.randint(1, 7, size=6_000_000)
end = time.time()
print(f"Tempo: {end - start:.4f} segundos")
```

✓ 3.4s

```
Tempo: 3.4836 segundos
```

5. Performance: lista vs array

- Qual estrutura foi mais rápida? Por quê?
  R: A execução do array foi mais rápida devido a tipagem ser fixa para todos os elementos, devido ao fato do numPy ser implementado em C e NumPy faz operações com arrays inteiros de uma vez (sem loops explícitos).

```python
numeros = np.arange(1,6)
print(numeros)
print(numeros * 2)
print(numeros ** 3)
print(numeros)

print(numeros * [2,2,2,2,2])

numeros2 = np.linspace(1.1,5.5,5)
print(numeros2)
print(numeros * numeros2)
```

✓ 0.0s

```
[1 2 3 4 5]
[ 2  4  6  8 10]
[  1   8  27  64 125]
[1 2 3 4 5]
[ 2  4  6  8 10]
[1.1 2.2 3.3 4.4 5.5]
[ 1.1  4.4  9.9 17.6 27.5]
```

```python
numeros = np.array([11,12,13,14,15])
print(numeros ⩾ 13)
numeros2 = np.array([1.1,2.2,3.3,4.4,5.5])
print(numeros2 < numeros)
print(numeros2 == numeros)
print(numeros2 ≠ numeros)
```

✓ 0.0s

```
[False False  True  True  True]
[ True  True  True  True  True]
[False False False False False]
[ True  True  True  True  True]
```

```python
notas = np.array([[87,96,70], [100,87,90], [94,77,90], [100,81,82]])
print(notas.sum())
print(notas.min())
print(notas.max())
print(notas.mean())
print(notas.std())
print(notas.var())
```

✓ 0.0s

```
1054
70
100
87.83333333333333
8.792357792739987
77.30555555555556
```

```python
print(notas.mean(axis=0))
```

✓ 0.0s

```
[95.25 85.25 83.  ]
```

```python
print(notas.mean(axis=1))
```

✓ 0.0s

```
[84.33333333 92.33333333 87.         87.66666667]
```

6. Operadores e broadcasting

- O que acontece se os arrays tiverem tamanhos diferentes?

  R: Se os tamanhos forem compatíveis via broadcasting, a operação é feita. Se não forem, o NumPy lança um erro.

```
a = np.array([1, 2, 3])
b = np.array([10, 20])
print(a + b)  # ValueError: shapes (3,) and (2,) not compatible
```

- A operação foi feita elemento a elemento? Como você sabe?

  R: Todas as operações com arrays NumPy são feitas elemento a elemento, desde que os shapes sejam compatíveis.

  - As saídas mostram os resultados de cada par de elementos.
  - NumPy é projetado para operações vetorizadas, que são, por definição, feitas elemento por elemento.
  - Se você fizer a operação manualmente em um loop, os resultados batem

```
numeros = np.array([1,4,9,16,25,36])
print(np.sqrt(numeros))
numeros2 = np.arange(1,7) * 10
print(numeros2)

np.add(numeros, numeros2)
print(np.multiply(numeros2,5))

numeros3 = numeros2.reshape((2,3))
print(numeros3)
numeros4 = np.array([2,4,6])
print(np.multiply(numeros3, numeros4))
```

✓ 0.0s

```
[1. 2. 3. 4. 5. 6.]
[10 20 30 40 50 60]
[ 50 100 150 200 250 300]
[[10 20 30]
 [40 50 60]]
[[ 20  80 180]
 [ 80 200 360]]
```

7. ufuncs (funções universais)

- O que a função faz exatamente?

  R: Uma ufunc (universal function) aplica operações matemáticas elemento a elemento de forma vetorizada.

```
np.add, np.subtract, np.multiply, np.divide
np.sqrt, np.sin, np.exp, np.log, etc.
```

- A função altera o array original ou retorna um novo?

  R: O array original não é alterado, a menos que você use o parâmetro out= para sobrescrever.

```
notas = np.array([[87,96,70], [100,87,90], [94,77,90], [100,81,82]])
print(notas)
print(notas[0,1])
print(notas[1])
print(notas[0:2,1])
print(notas[:,1:3])
print(notas[::2])
```

✓ 0.1s

```
[[ 87  96  70]
 [100  87  90]
 [ 94  77  90]
 [100  81  82]]
96
[100  87  90]
[96 87]
[[96 70]
 [87 90]
 [77 90]
 [81 82]]
[[87 96 70]
 [94 77 90]]
```

8. Indexação e slicing

- O que representa o índice negativo?
  R: Um índice negativo acessa elementos a partir do fim do array. -1 é o último elemento, -2 é o penúltimo, e assim por diante.

- O slicing inclui o índice final?
  R: Não. O índice final é excluído

```
a = np.array([10, 20, 30, 40, 50])
print(a[1:4])  # [20 30 40] → posição 1 até 3 (exclui o índice 4 - a[início:fim:passo])
```

- O resultado é uma cópia ou uma view?
  R: O resultado de um slicing em NumPy é geralmente uma view — ou seja, não copia os dados,
  só referencia os mesmos valores na memória, se você quiser uma cópia de verdade, use .copy():

```
numeros = np.arange(1,6)
print(numeros)
numeros2 = numeros.view()
```

✓ 0.0s

```
[1 2 3 4 5]
```

```
print(id(numeros))
print(id(numeros2))
```

✓ 0.0s

```
1770301071856
1770302062096
```

```
numeros[1] *= 10
print(numeros2)
```

✓ 0.0s

```
[ 1 20  3  4  5]
```

```
numeros2[1] /= 10
print(numeros)
```

✓ 0.0s

```
[1 2 3 4 5]
```

```
numeros2 = numeros[0:3]
print(numeros2)
```

✓ 0.0s

```
[1 2 3]
```

```
print(id(numeros))
print(id(numeros2))
```

✓ 0.0s

```
1770301071856
1770302354032
```

```
numeros = np.arange(1,6)
print(numeros)
numeros2 = numeros.copy()
print(numeros2)
```
✓ 0.0s

```
[1 2 3 4 5]
[1 2 3 4 5]
```

```
numeros[1] *= 10
print(numeros)
print(numeros2)
```
✓ 0.0s

```
[ 1 20  3  4  5]
[1 2 3 4 5]
```

```
notas = np.array([[87,96,70], [100,87,90]])
print(notas)
notas.reshape((1,6))
print(notas)
```
✓ 0.0s

```
[[ 87  96  70]
 [100  87  90]]
[[ 87  96  70]
 [100  87  90]]
```

```
notas.resize(1,6)
print(notas)
```
✓ 0.0s

```
[[ 87  96  70 100  87  90]]
```

9. View (shallow copy) vs cópia profunda

- Qual é o id() do array original e da cópia? Eles são iguais?
  R: id do array original (1770301071856) e id da cópia (1770302354032).
  Todos têm IDs diferentes, pois são objetos distintos.
  Mas a view compartilha a mesma área de dados na memória do original.

- O que acontece ao alterar a view? Isso afeta o original?
  R: Alterações na view afetam o array original, porque eles compartilham os mesmos dados.
  Por outro lado, a cópia profunda é totalmente independente.

- Quando devemos usar cópia profunda?
  R: Você vai modificar os dados e NÃO quer que o original seja afetado;
  Vai passar o array pra outra função que pode alterar e você quer preservar o original;
  Vai armazenar dados em cache ou backup que não podem ser alterados acidentalmente;
  Quer evitar bugs por efeitos colaterais.

```
notas = np.array([[87,96,70], [100,87,90]])
print(notas)
flattened = notas.flatten()
print(flattened)
flattened[0] = 100
print(flattened)
print(notas)
```

✓ 0.0s

```
[[ 87  96  70]
 [100  87  90]]
[ 87  96  70 100  87  90]
[100  96  70 100  87  90]
[[ 87  96  70]
 [100  87  90]]
```

```
reveled = notas.ravel()
print(reveled)
reveled[0] = 100
print(reveled)
print(notas)
```

✓ 0.0s

```
[ 87  96  70 100  87  90]
[100  96  70 100  87  90]
[[100  96  70]
 [100  87  90]]
```

10. Flatten vs Ravel

- Ambos retornam arrays unidimensionais? Qual a diferença real entre eles?
  R: Sim, tanto flatten() quanto ravel() retornam arrays unidimensionais. A diferença principal é que flatten() sempre retorna uma cópia independente dos dados (deep copy), enquanto ravel() tenta retornar uma view (shallow copy) do array original sempre que possível — ou seja, ele não copia os dados, apenas reorganiza a visualização deles na memória.

- O que acontece se você alterar um valor no array retornado por ravel()?
  R: Se ravel() retornou uma view (o mais comum), qualquer alteração no array retornado também modifica o array original, já que ambos compartilham a mesma área de memória. Para evitar isso, deve-se usar flatten(), que retorna uma cópia independente.

```
notas.T
notas
```

✓ 0.0s

```
array([[100,  96,  70],
       [100,  87,  90]])
```

```
notas2 = np.array([[94,77,90], [100,81,82]])
print(np.hstack((notas, notas2)))
print(np.vstack((notas, notas2)))
```

✓ 0.0s

```
[[100  96  70  94  77  90]
 [100  87  90 100  81  82]]
[[100  96  70]
 [100  87  90]
 [ 94  77  90]
 [100  81  82]]
```

11. Transposição e empilhamento

- Qual é o formato do array após a transposição?
  R: A transposição (.T) inverte as dimensões do array. Ou seja, linhas viram colunas e colunas viram linhas.

- hstack aumenta o número de colunas ou de linhas?
  R: hstack (horizontal stack) aumenta o número de colunas. Ele junta os arrays lado a lado (horizontalmente).

- Quais as condições para que vstack e hstack funcionem corretamente?
  R: As formas dos arrays precisam ser compatíveis:
  Para vstack (vertical):
  → Todos os arrays devem ter o mesmo número de colunas.

  Para hstack (horizontal):
  → Todos os arrays devem ter o mesmo número de linhas.