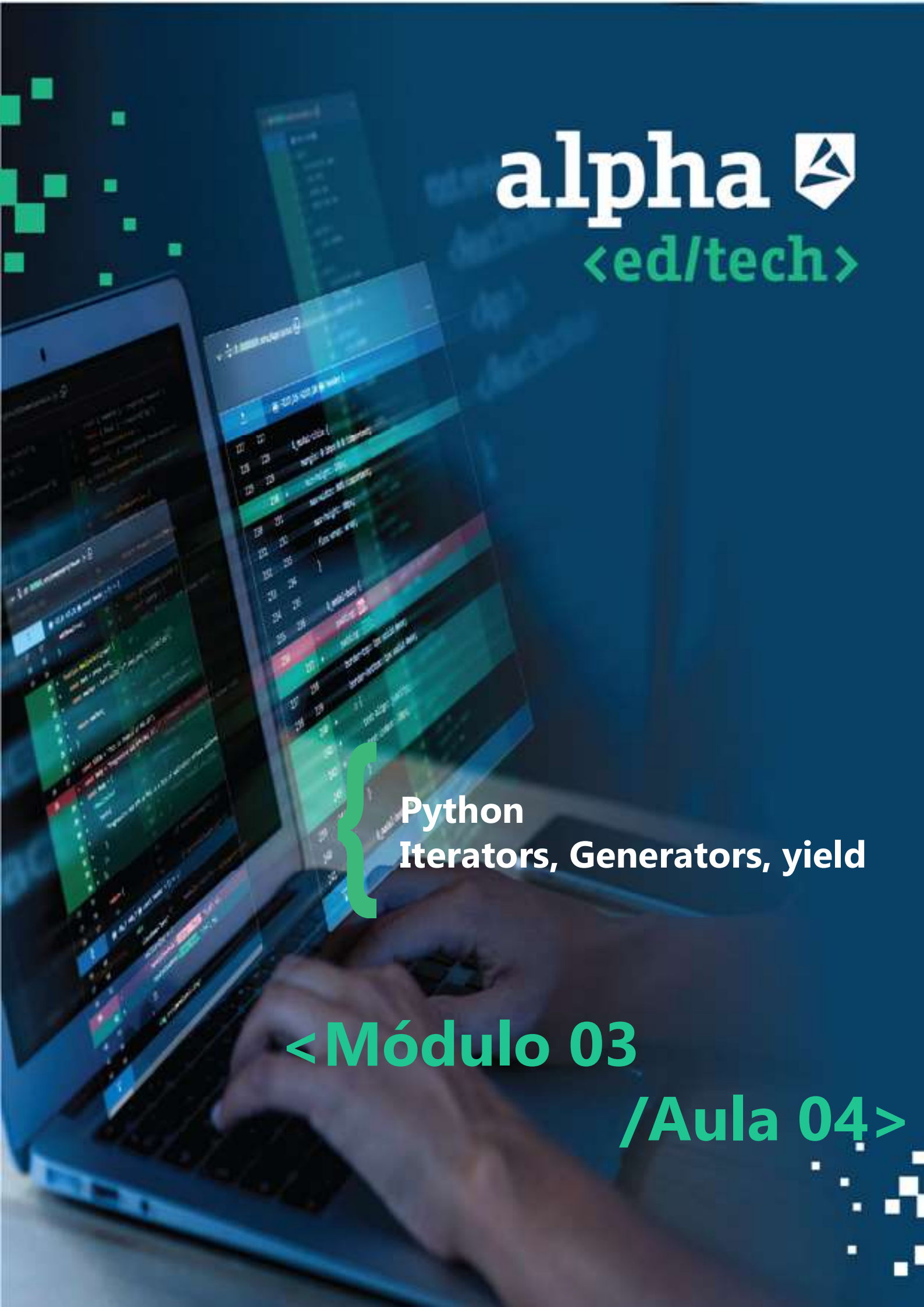
alpha
<ed/tech>
Python
Iterators, Generators, yield
<Módulo 03
/Aula 04>

Iterators, Generators, yield

Introdução

Bem-vindos ao capítulo sobre **Iterators, Generators e yield** em Python.

Iterators, **generators** e o comando `**yield**` são conceitos relacionados em Python, todos eles desempenham um papel importante na criação de **iteráveis eficientes** e no gerenciamento de **grandes conjuntos de dados.** Vamos entender cada um deles separadamente:

Iterators (Iteradores)

Um **iterator** é um objeto que implementa os métodos `__iter__()` e `__next__()`. Esses métodos permitem que **um objeto seja iterado**, ou seja, percorrido em um loop.

```
class MyIterator:
      def __init__(self, data):
          self.data = data
          self.index = 0

      def __iter__(self):
          return self

      def __next__(self):
          if self.index < len(self.data):
              result = self.data[self.index]
              self.index += 1
              return result
          else:
              raise StopIteration
```

```
  my_iter = MyIterator([1, 2, 3])

  for item in my_iter:
      print(item)
```

Iterators são poderosos e amplamente usados, especialmente em construções de loop e em operações que envolvem grandes conjuntos de dados.

Exemplo de Iterators

Observação: cuidado com a palavra "**interação**", que refere-se à **ação ou efeito de interagir**, que, por sua vez, implica a **ação mútua entre dois ou mais elementos**. Em diversos contextos, o termo é utilizado para descrever a **comunicação, influência recíproca ou ação conjunta entre entidades**.

Um **iterator** é um objeto que implementa os métodos `__iter__()` e `__next__()` (ou `__iter__()` e `__getitem__()` em **versões mais antigas do Python**). Esses métodos permitem que um **objeto** seja **iterado**, ou seja, **percorrido em um loop**.

- `__iter__()` retorna o **próprio objeto iterator** (geralmente `self`).
- `__next__()` retorna o **próximo elemento na sequência a cada chamada** e lança a exceção `StopIteration` quando não há mais elementos para serem iterados.

```
# class MyIterator:
    def __init__(self, data):
        self.data = data
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.data):
            result = self.data[self.index]
            self.index += 1
            return result
        else:
            raise StopIteration
```

```
my_iter = MyIterator([1, 2, 3])

for item in my_iter:
    print(item)
```

Nem todos os **iteráveis** são **iteradores**, mas todos os **iteradores** são **iteráveis**.

Entendendo Iteração e Iteráveis

Iteráveis: São objetos que podem ser percorridos. Exemplos incluem listas, tuplas, strings, dicionários etc.

Iteradores: São objetos que mantêm o estado durante a iteração e sabem como acessar os próximos elementos em uma sequência. Nem todos os iteráveis são iteradores, mas todos os iteradores são iteráveis.

Iterável que não é Iterador: **Listas, tuplas e strings são iteráveis**, mas não são iteradores. Eles podem ser iterados, mas você não pode usá-los diretamente com a função `next()`.

```
minha_lista = [1, 2, 3, 4]

# Tentar usar next() diretamente em um iterável resultará em um erro.
# Isso gera TypeError: 'list' object is not an iterator
# next(minha_lista)
```

Podemos converter um iterável em um iterador usando a função `iter()`.

```
minha_lista = [1, 2, 3, 4]
iterador = iter(minha_lista)

print(next(iterador))  # 1
print(next(iterador))  # 2
```

Todos os iteradores são iteráveis usando Iterador Embutido:

Os objetos que implementam os métodos `__iter__` e `__next__` são iteradores e, portanto, são iteráveis.
Neste exemplo `meu_iterador` é tanto um iterador quanto um iterável.

```
class MeuIterador:
      def __init__(self, data):
          self.data = data
          self.index = 0

      def __iter__(self):
          return self

      def __next__(self):
          if self.index < len(self.data):
              result = self.data[self.index]
              self.index += 1
              return result
          else:
              raise StopIteration

  # Criando um iterador
  meu_iterador = MeuIterador([1, 2, 3, 4])

  # Usando o iterador em um loop
  for elemento in meu_iterador:
      print(elemento)
```

Podemos converter um iterável em um iterador usando a função `iter()`.

```
minha_lista = [1, 2, 3, 4]
iterador = iter(minha_lista)

print(next(iterador))  # 1
print(next(iterador))  # 2
```

Todos os **iteradores** são **iteráveis**, usando **Iterador Embutido**:

Os objetos que implementam os métodos `__iter__` e `__next__` são iteradores e, portanto, são iteráveis.

Aqui, `meu_iterador` é tanto um iterador quanto um iterável.

```
class MeuIterador:
    def __init__(self, data):
       self.data = data
       self.index = 0

    def __iter__(self):
       return self

    def __next__(self):
       if self.index < len(self.data):
           result = self.data[self.index]
           self.index += 1
           return result
       else:
           raise StopIteration

# Criando um iterador
meu_iterador = MeuIterador([1, 2, 3, 4])

# Usando o iterador em um loop
for elemento in meu_iterador:
    print(elemento)
```

.

Todos os **iteradores** são **iteráveis**, com o uso de **Objetos Incorporados**:

Muitos objetos em Python são iteráveis e também são iteradores.
Por exemplo, as strings podem ser percorridas usando um loop `for` ou convertidas em um iterador.

```
minha_string = "Python"

# Iterando sobre uma string usando um loop for
for char in minha_string:
    print(char)

# Convertendo a string em um iterador
iterador_string = iter(minha_string)

print(next(iterador_string))  # P
print(next(iterador_string))  # y
```

Portanto, enquanto **nem todo iterável é um iterador (como listas)**, é possível converter muitos iteráveis em iteradores. No entanto, **objetos que são iteradores são, por definição, também iteráveis**.

Criando um Iterador Simples

Vamos criar uma classe que representa **um iterador simples para uma lista**:

```
class MeuIterador:
    def __init__(self, data):
        self.data = data
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.data):
            result = self.data[self.index]
            self.index += 1
            return result
        else:
            raise StopIteration
```

•`**__init__(self, data)**`: Inicializa o iterador com uma coleção de dados.
•`**__iter__(self)**`: Retorna o próprio objeto iterador (necessário para o protocolo de iteração).
•`**__next__(self)**`: Retorna o próximo elemento na sequência ou levanta `StopIteration` quando não há mais elementos.

Utilizando o Iterador, podemos usar nosso iterador para percorrer uma lista:

```
meus_dados = [1, 2, 3, 4, 5]
meu_iterador = MeuIterador(meus_dados)

for elemento in meu_iterador:
    print(elemento)
```

O Python fornece **iteradores embutidos** como `iter()` e `next()`.

Vamos reescrever nosso exemplo usando essas funções:

```
meus_dados = [1, 2, 3, 4, 5]
meu_iterador = iter(meus_dados)

while True:
    try:
        elemento = next(meu_iterador)
        print(elemento)
    except StopIteration:
        break
```

Usando **Iteradores** com o Statement (comando) `for`:

O loop `for` em Python usa iteradores automaticamente. Podemos simplificar ainda mais:

```
meus_dados = [1, 2, 3, 4, 5]

for elemento in meus_dados:
    print(elemento)
```

Lembre-se de que, na prática, em Python, muitos objetos são iteráveis e já têm iteradores embutidos (como listas, tuplas, strings etc.), então você geralmente não precisa criar seus próprios iteradores, a menos que esteja criando uma estrutura de dados personalizada.

Essa é uma explicação simplificada para começar com iteradores em Python. Eles são poderosos e amplamente usados, especialmente em construções de loop e em operações que envolvem grandes conjuntos de dados.

Generators

Os **Generators** são uma maneira mais concisa e eficiente de criar **Iterators** em Python. Eles são definidos usando funções com a palavra-chave `yield`.

Quando uma função com `yield` é chamada, ela retorna um objeto `generator`, mas a **execução é pausada** até que o **próximo valor seja solicitado**. Isso permite que os **gerators** sejam utilizados para gerar valores sob demanda, economizando memória.

```
def my_generator(data):
    for item in data:
        yield item

gen = my_generator([1, 2, 3])

for item in gen:
    print(item)
```

O uso de `**yield**` permite que a função seja suspensa entre as chamadas, mantendo seu estado.

Vamos entender mais aprofundadamente a utilização de **generators**:

1. Funções Geradoras:

•Funções geradoras são definidas usando a palavra-chave `yield`. O `yield` pausa a execução da função e retorna um valor para quem está chamando, **mantendo o estado da função**.

•A cada chamada subsequente da função, ela **retoma a execução de onde foi pausada**, mantendo o estado interno.

Criando um Generator Simples

```
def meu_gerador(inicio, fim):
    while inicio <= fim:
        yield inicio
        inicio += 1
```

A função `meu_gerador` é um generator que gera números de `inicio` até `fim`.

Generators em Python são uma forma mais simples e eficiente de criar **Iterators**. Eles permitem a criação de sequências de valores de uma maneira mais econômica em termos de memória e mais fácil de entender em comparação com a implementação manual de **iterators**.

3. Utilizando o Generator

```
gen = meu_gerador(1, 5)

for numero in gen:
    print(numero)

#print(gen)
#print(type(gen))
```

Ao iterar sobre o **generator**, ele produzirá valores sequencialmente sem gerar a sequência completa de uma vez.

4. Vantagens dos Generators:

•**Economia de Memória**: Os **generators** geram valores sob demanda, economizando memória, especialmente útil para grandes conjuntos de dados.

•**Lazy Evaluation**: Os valores são gerados apenas quando necessário, tornando o código mais eficiente em termos de tempo de execução.

5. Outro Exemplo de Generator

```
def fibonacci(n):
    a, b = 0, 1
    contador = 0
    while contador < n:
        yield a
        a, b = b, a + b
        contador += 1
```

Este **generator** gera os primeiros `n` números da sequência de Fibonacci.

6. List Compreension vs. Generator Expression

```
# Compreensão de Lista
lista_quadrados2 = []
for x in range(1, 6)
    lista_quadrados2.append(x**2)
print(lista_quadrados2)
print(type(lista_quadrados2))

# Compreensão de Lista
lista_quadrados = [x**2 for x in range(1, 6)]
print(lista_quadrados)
print(type(lista_quadrados))

# Generator Expression
generator_quadrados = (x**2 for x in range(1, 6))
print(generator_quadrados)
print(type(generator_quadrados))
```

A compreensão de lista cria uma lista completa na memória, enquanto o **generator expression** gera valores sob demanda.

7. Uso do método `next()` para obter próximos valores

```
gen = meu_gerador(1, 3)

print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
# print(next(gen))
# Isso resultaria em StopIteration, indicando o fim da sequência
```

A função `next()` obtém o próximo valor do **generator**.

Generators são poderosos para trabalhar com grandes conjuntos de dados e para implementar lógica iterativa de uma maneira mais clara e eficiente.
Eles são especialmente úteis quando você não precisa de todos os valores de uma vez e deseja economizar recursos computacionais.

yield

O `**yield**` é utilizado dentro de funções para criar **generators**. Quando a função é chamada, ela **não é executada imediatamente**. Em vez disso, um **objeto generator** é retornado e a **execução da função é adiada até que o próximo valor seja solicitado**.

Cada vez que a função com `**yield**` é chamada, ela retorna o valor especificado após o `**yield**` e é suspensa até a próxima chamada. O estado da função é mantido entre chamadas, o que é útil para manter o contexto e os valores das variáveis.

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for i in countdown(5):
    print(i)
```

Este exemplo cria um contador regressivo que gera valores de 5 a 1 quando iterado.

Esses conceitos são essenciais para lidar com grandes conjuntos de dados ou para criar iteráveis eficientes em Python. Os geradores, em particular, são frequentemente preferidos devido à sua eficiência em termos de uso de memória.

List Comprehension

List comprehension é uma construção sintática em Python que permite criar listas de maneira concisa e elegante.

Essa técnica é útil quando você deseja criar uma nova lista aplicando uma expressão a cada item de uma sequência existente (por exemplo, uma lista, uma string ou um intervalo numérico).

A sintaxe básica de uma **list comprehension** é a seguinte:

```
nova_lista = [expressao for item in sequencia if condicao]
```

`**expressao**`: A expressão que será aplicada a cada item da sequência.
`**item**`: A variável que representa cada item da sequência.
`**sequencia**`: A sequência de onde os itens são retirados (por exemplo, lista, string, intervalo).
`**condicao**` (opcional): Uma condição que filtra os itens. A nova lista incluirá apenas os itens que atendem a essa condição.

Aqui estão alguns exemplos para ilustrar o uso de **list comprehension**:

Exemplo 1: Criando uma lista de quadrados dos números de 0 a 4

```
quadrados = [x**2 for x in range(5)]
# Resultado: [0, 1, 4, 9, 16]
```

Exemplo 2: Filtrando apenas os números pares de uma lista

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]
pares = [x for x in numeros if x % 2 == 0]
# Resultado: [2, 4, 6, 8]
```

Exemplo 3: Criando uma lista de strings maiúsculas

```
palavras = ["python", "é", "legal"]
maiusculas = [palavra.upper() for palavra in palavras]
# Resultado: ['PYTHON', 'É', 'LEGAL']
```

Exemplo 4: Gerando uma lista de tuplas combinando elementos de duas listas

```
lista1 = [1, 2, 3]
lista2 = ['a', 'b', 'c']
combinacao = [(x, y) for x in lista1 for y in lista2]
# Resultado: [(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'), (3,
'a'), (3, 'b'), (3, 'c')]
```

List comprehension é uma ferramenta poderosa que torna o código mais conciso e legível. Entretanto, é importante usá-la com moderação para garantir que o código permaneça compreensível. Quando a lógica se torna muito complexa, pode ser mais apropriado usar um loop `for` tradicional.