PYTHON ACADEMY

LIST COMPREHENSIONS (COMPREENSÃO DE LISTAS) NO PYTHON

Guia completo de List Comprehensions em Python: sintaxe, performance, benchmarks, comparação com generators, casos de uso reais e quando evitar. Tutorial com exemplos práticos e análise de memória.

PYTHONACADEMY.COM.BR

> ✅ ***Atualizado para Python 3.13*** *(Dezembro 2025)*
> *Conteúdo enriquecido com benchmarks de performance, análise de memória, comparação com generators e casos de uso do mundo real.*

Olá Pythonista!

List Comprehensions são uma das features mais elegantes e poderosas do Python. Elas permitem criar e transformar listas de forma **concisa, legível e até 3x mais rápida** que loops tradicionais.

Neste guia completo, você vai aprender: - ✅ Sintaxe e padrões de list comprehensions - ✅ **Benchmarks reais** de performance - ✅ **Comparação Generator vs List** (memória) - ✅ Casos de uso do mundo real - ✅ Quando **NÃO** usar (legibilidade)

Ah, você sabia que o mesmo conceito pode ser aplicado aos dicionários (`dict`) do Python?

Já abre o post sobre *Dict Comprehensions* em outra aba e corre pra lá quando terminar aqui! 😉

Listas em Python

Lista é uma estrutura de dados provida pela própria linguagem e que utilizamos muito na programação Python.

Saber como manuseá-las corretamente, otimizando seu código e tirando maior proveito daquilo que o Python nos proporciona, é sempre uma **boa ideia**.

Os seguintes métodos estão disponíveis em uma lista:

- `list.append(x)` : Adiciona um item ao fim da lista.
- `list.extend(iterable)` : Adiciona todos os itens do iterável *iterable* ao fim da lista.
- `list.insert(i, x)` : Insere um item em uma dada posição **i**.
- `list.remove(x)` : Remove o primeiro elemento, cujo valor seja **x**.
- `list.pop(i)` : Remove o item de posição **i** da lista e o retorna. Caso **i** não seja especificado, retorna o último elemento da lista.
- `list.clear()` : Remove todos os elementos da lista.
- `list.index(x[, start[, end]])` : Retorna o índice do primeiro elemento cujo valor seja **x**.
- `list.count(x)` : Retorna o número de vezes que o valor **x** aparece na lista.
- `list.sort(key=None, reverse=False)` : Ordena os items da lista (os argumentos podem ser usados para customizar a ordenação).
- `list.reverse()` : Reverte os elementos da lista.
- `list.copy()` : Retorna uma lista com a cópia dos elementos da lista de origem.

Em Python, utilizamos colchetes para criação de listas. Exemplo:

```
# Apenas números
lista_numerica = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Letras e números
lista_alfanumerica = ['a', 'b', 'c', 1, 2, 3]
```

List Comprehensions (Compreensão de Listas)

List Comprehension foi concebida na PEP 202 e é uma forma concisa de criar e manipular listas.

Sua sintaxe básica é:

```
[expr for item in lista]
```

Em outras palavras: aplique a expressão `expr` em cada `item` da `lista`.

Exemplo: dado o seguinte código:

```
for item in range(10):
  lista.append(item**2)
```

Podemos reescrevê-lo, utilizando *list comprehensions*, da seguinte forma:

```
lista = [item**2 for item in range(10)]
```

Ou seja: aplique a potência de 2 em todos os itens da lista.

Outro Exemplo: dado o seguinte código, que transforma os itens da lista em maiúsculos:

```
for item in lista:
  resultado.append(str(item).upper())
```

Podemos reescrevê-lo da seguinte forma:

```
resultado = [str(item).upper() for item in lista]
```

Performance: List Comprehension vs For Loop

Uma das principais vantagens de list comprehensions é a **performance**. Vamos medir com dados reais!

Benchmark 1: Criação de Lista Simples

Vamos comparar criar uma lista com os quadrados de 10.000 números:

```python
import timeit

# Setup: dados de teste
setup = "data = range(10000)"

# Método 1: For loop tradicional
for_loop = """
result = []
for x in data:
    result.append(x**2)
"""

# Método 2: List comprehension
list_comp = "[x**2 for x in data]"

# Executar 1000 vezes e medir tempo
time_loop = timeit.timeit(for_loop, setup, number=1000)
time_comp = timeit.timeit(list_comp, setup, number=1000)

print(f"For loop:          {time_loop:.4f}s")
print(f"List comprehension: {time_comp:.4f}s")
print(f"List comp é {time_loop/time_comp:.2f}x mais rápida!")
```

Resultado típico (Python 3.13):

```
For loop:          2.4521s
List comprehension: 0.8943s
List comp é 2.74x mais rápida!
```

Por que List Comprehensions são mais rápidas?

1. **Otimização do interpretador**: Python detecta list comprehensions e otimiza bytecode
2. **Menos chamadas de função**: `append()` é chamado toda iteração no loop

3. **Alocação de memória**: List comprehension pre-aloca espaço quando possível

Benchmark 2: Filtragem e Transformação

Vamos filtrar números pares e elevar ao quadrado:

```
import timeit

setup = "data = range(10000)"

# For loop com if
for_loop_if = """
result = []
for x in data:
    if x % 2 == 0:
        result.append(x**2)
"""

# List comprehension com if
list_comp_if = "[x**2 for x in data if x % 2 == 0]"

time_loop = timeit.timeit(for_loop_if, setup, number=1000)
time_comp = timeit.timeit(list_comp_if, setup, number=1000)

print(f"For loop com if:    {time_loop:.4f}s")
print(f"List comp com if:   {time_comp:.4f}s")
print(f"Diferença: {((time_loop - time_comp) / time_loop * 100):.1f}%
        mais rápido")
```

Resultado típico:

```
For loop com if:    1.8234s
List comp com if:   0.9821s
Diferença: 46.1% mais rápido
```

> 💡 ***Dica Pro:*** *List comprehensions são especialmente eficientes para operações com filtros (`if`), onde a vantagem de performance pode chegar a* ***50-60%****!*

List Comprehensions com if

List comprehensions podem utilizar expressões condicionais para criar listas ou modificar listas existentes.

Sua sintaxe básica é:

```
[expr for item in lista if cond]
```

Ou seja:

> *Aplique a expressão `expr` em cada `item` da `lista` caso a condição `cond` seja satisfeita.*

Vamos criar algumas listas utilizando condições.

Por exemplo, podemos retirar os números ímpares de um conjunto de número da seguinte forma:

```
resultado = [numero for numero in range(20) if numero % 2 == 0]
```

O que resulta em:

```
resultado = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Vamos ver como fica com vários *if's*.

List Comprehensions com vários *if's*

Podemos verificar condições em duas listas diferentes dentro da mesma *list comprehension*.

Por exemplo: gostaríamos de saber os **Múltiplos Comuns** de 5 e 6.

Utilizando múltiplos `if's` e *list comprehensions*, podemos criar o seguinte código:

```
resultado = [numero for numero in range(100) if numero % 5 == 0 if nu-
        mero % 6 == 0]
```

Ou seja, o número só será passado para lista `resultado` caso sua divisão por 5 **E** por 6 seja igual à zero.

O resultado do código acima será:

```
resultado = [0, 30, 60, 90]
```

List Comprehensions com *if* + *else*

Outra forma de se utilizar expressões condicionais e *list comprehension* é usar o conjunto `if` + `else`.

A sintaxe básica para essa construção é:

```
[resultado_if if expr else resultado_else for item in lista]
```

Em outras palavras: para cada item da lista, aplique o resultado `resultado_if` se a expressão `expr` for verdadeira, caso contrário, aplique `resultado_else`.

Por exemplo, queremos criar uma lista que contenha “1” quando determinado número for múltiplo de 5 e “0” caso contrário.

Podemos codificá-lo da seguinte forma:

```
resultado = ['1' if numero % 5 == 0 else '0' for numero in range(16)]
```

Dessa forma, teremos o seguinte resultado:

```
resultado = ['1', '0', '0', '0', '0', '1', '0', '0', '0', '0', '1',
'0', '0', '0', '0', '1']
```

Múltiplas *List Comprehensions* (aninhadas)

É aqui que a brincadeira fica **séria**!

Vamos supor que queiramos transpor uma matriz.

Pra quem não lembra o que é a **Transposição de uma Matriz**, vamos relembrar:

> *Transpor uma matriz, significa transformar as linhas em colunas e vice-versa.*

Ou seja, data a seguinte matriz:

```
matrix = [
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9, 10, 11, 12]
]
```

Queremos o seguinte resultado:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}^{\mathrm{T}}$$

Em Python, podemos fazer isso da seguinte forma:

```
transposta = []
matriz = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]

for i in range(len(matriz[0])):
    linha_transposta = []

    for linha in matriz:
        linha_transposta.append(linha[i])
    transposta.append(linha_transposta)
```

A matriz `transposta` conteria:

```
transposta = [[1, 4, 9], [2, 5, 10], [3, 6, 11], [4, 8, 12]]
```

Podemos reescrever o código acima, de transposição de matrizes, da seguinte forma, utilizando *list comprehension*:

```
matriz = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
transposta = [[linha[i] for linha in matriz] for i in range(4)]
```

No código acima:

- No **primeiro** ***loop***, `i` assume o valor de **0**, portanto `[linha[0] for linha in matriz]` vai retornar o primeiro elemento de cada linha: `[1, 4, 9]`
- No **segundo** ***loop***, `i` assume o valor de **1**, portanto `[linha[1] for linha in matriz]` vai retornar o segundo elemento de cada linha: `[2, 5, 10]`
- No **terceiro** ***loop***, `i` assume o valor de **2**, portanto `[linha[2] for linha in matriz]` vai retornar o terceiro elemento de cada linha: `[3, 6, 11]`
- No **quarto** ***loop***, `i` assume o valor de **3**, portanto `[linha[3] for linha in matriz]` vai retornar o quarto elemento de cada linha: `[4, 8, 12]`

Obtendo, assim, o mesmo resultado.

Generator Expression vs List Comprehension

Generators são similares a list comprehensions, mas com uma diferença crucial: **não armazenam todos os valores na memória de uma vez**.

Sintaxe: Parênteses vs Colchetes

```
# List Comprehension (colchetes) - carrega tudo na memória
lista = [x**2 for x in range(1000000)]

# Generator Expression (parênteses) - lazy evaluation
generator = (x**2 for x in range(1000000))
```

Comparação de Memória

Vamos medir o consumo de memória de cada abordagem:

```
import sys

# Lista: armazena TODOS os 1 milhão de valores
lista = [x**2 for x in range(1000000)]
print(f"Lista:     {sys.getsizeof(lista):,} bytes ({sys.getsize-
        of(lista) / 1024 / 1024:.2f} MB)")

# Generator: armazena apenas o estado atual
generator = (x**2 for x in range(1000000))
print(f"Generator: {sys.getsizeof(generator):,} bytes ({sys.getsize-
        of(generator) / 1024:.6f} MB)")

# Diferença
diferenca = sys.getsizeof(lista) / sys.getsizeof(generator)
print(f"\nLista usa {diferenca:.0f}x MAIS memória!")
```

Resultado típico:

```
Lista:     8,448,728 bytes (8.06 MB)
Generator: 104 bytes (0.000102 MB)

Lista usa 81,238x MAIS memória!
```

Quando Usar Cada Um?

Use List Comprehension quando: - ✅ Você precisa **acessar os valores múltiplas vezes** - ✅ Precisa de **indexação** (`lista[5]`) - ✅ Precisa dos **métodos de lista** (`len()` , `sort()` , `reverse()`) - ✅ O dataset é **pequeno** (< 10.000 itens)

Use Generator quando: - ✅ Você vai **iterar apenas uma vez** - ✅ O dataset é **grande** ou **infinito** - ✅ Quer **economizar memória** - ✅ Processa **streams de dados** (logs, arquivos, APIs)

Exemplo Prático: Processar Arquivo Grande

```
# ❌ RUIM: Carrega arquivo inteiro na memória (pode estourar RAM)
linhas = [linha.strip().upper() for linha in open('gigante.log')]
for linha in linhas:
    processar(linha)

# ✅ BOM: Processa linha por linha (memória constante)
linhas = (linha.strip().upper() for linha in open('gigante.log'))
for linha in linhas:
    processar(linha)  # Processa uma de cada vez
```

> 💡 ***Regra de Ouro:*** *Se você iterar apenas **uma vez**, use **generator**. Se precisar acessar **múltiplas vezes**, use **list**.*

Casos de Uso do Mundo Real

Vamos ver exemplos práticos que você pode usar no dia a dia:

1. Limpeza de Dados CSV

```python
import csv

# Ler CSV e limpar dados em uma linha
with open('usuarios.csv') as f:
    reader = csv.DictReader(f)
    # Remove espaços e converte emails para minúsculas
    usuarios_limpos = [
        {
            'nome': row['nome'].strip(),
            'email': row['email'].lower().strip(),
            'idade': int(row['idade'])
        }
        for row in reader
        if row['email']  # Ignora linhas sem email
    ]
```

2. Processar Resposta de API (JSON)

```python
import requests

# Buscar repos do GitHub e extrair informações
response = requests.get('https://api.github.com/users/python/repos')
repos = response.json()

# Extrair apenas nome e estrelas dos repos com mais de 100 stars
repos_populares = [
    {'nome': repo['name'], 'stars': repo['stargazers_count']}
    for repo in repos
    if repo['stargazers_count'] > 100
]

print(repos_populares)
# [{'nome': 'cpython', 'stars': 45231}, {'nome': 'peps', 'stars':
        3421}, ...]
```

3. Filtrar DataFrames (Pandas)

```python
import pandas as pd

# Criar DataFrame
df = pd.DataFrame({
    'produto': ['A', 'B', 'C', 'D', 'E'],
    'preco': [10, 25, 15, 30, 12],
    'estoque': [100, 0, 50, 200, 5]
})

# Produtos disponíveis (estoque > 0) e baratos (< 20)
produtos_validos = [
    row['produto']
    for _, row in df.iterrows()
    if row['estoque'] > 0 and row['preco'] < 20
]

print(produtos_validos)  # ['A', 'C', 'E']
```

4. Flatten (Achatar) Lista de Listas

```python
# Lista aninhada
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Achatar em uma única lista
flat = [item for sublista in matriz for item in sublista]
print(flat)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Ou com sum (menos eficiente)
flat = sum(matriz, [])
```

5. Combinar Múltiplas Listas (Zip)

```python
nomes = ['Ana', 'Bruno', 'Carlos']
idades = [25, 30, 28]
cidades = ['SP', 'RJ', 'MG']

# Criar dicionário com todos os dados
pessoas = [
    {'nome': n, 'idade': i, 'cidade': c}
    for n, i, c in zip(nomes, idades, cidades)
]

print(pessoas)
# [{'nome': 'Ana', 'idade': 25, 'cidade': 'SP'}, ...]
```

Quando NÃO Usar List Comprehensions

List comprehensions são poderosas, mas podem prejudicar **legibilidade** quando mal usadas.

❌ Exemplo RUIM: Muito Complexo

```
# ❌ NÃO FAÇA ISSO - Ilegível!
resultado = [
    {'usuario': u['nome'], 'total': sum(c['valor'] for c in u['compras']
        if c['status'] == 'aprovado')}
    for u in usuarios
    if u['ativo'] and len([c for c in u['compras'] if c['status'] ==
        'aprovado']) > 0
]
```

Problema: Muito aninhamento, dificulta debug e manutenção.

✅ Solução: Quebrar em Etapas

```
# ✅ FAÇA ISSO - Legível e testável!
def calcular_total_aprovado(usuario):
    """Calcula total de compras aprovadas do usuário"""
    compras_aprovadas = [c['valor'] for c in usuario['compras'] if
        c['status'] == 'aprovado']
    return sum(compras_aprovadas)

def usuario_valido(usuario):
    """Verifica se usuário está ativo e tem compras aprovadas"""
    return usuario['ativo'] and calcular_total_aprovado(usuario) > 0

# Agora fica claro e testável
resultado = [
    {'usuario': u['nome'], 'total': calcular_total_aprovado(u)}
    for u in usuarios
    if usuario_valido(u)
]
```

Regras de Legibilidade

✅ **Use list comprehensions quando:** 1. Cabe em **1-2 linhas** (máx 79-100 caracteres) 2. A lógica é **simples** (1 filtro, 1 transformação) 3. **Não precisa de debug complexo**

❌ **Evite list comprehensions quando:** 1. Tem **múltiplos níveis de aninhamento** 2. Lógica **complexa** com vários `if/else` 3. Você precisa **imprimir valores intermediários** (debug) 4. Usa **side effects** (modificar variáveis externas, I/O)

Exemplo: Quando Usar For Loop Normal

```
# ❌ RUIM: Side effect em list comprehension
[print(x) for x in lista]  # NÃO FAÇA ISSO!

# ✅ BOM: For loop quando precisa side effects
for x in lista:
    print(x)
    log.info(f'Processando {x}')
    if x > 100:
        enviar_alerta(x)
```

> 💡 ***Lembre-se:*** *Código **legível** é mais importante que código **conciso**. Se alguém vai demorar 5 minutos para entender sua list comprehension, use um loop normal!*

Conclusão

Neste guia completo sobre **List Comprehensions**, você aprendeu:

✅ **Sintaxe e padrões** - Do básico ao aninhamento
✅ **Performance real** - List comprehensions são 2-3x mais rápidas que loops
✅ **Generator vs List** - Economia de 80.000x em memória
✅ **Casos de uso reais** - CSV, APIs, Pandas, flatten, zip
✅ **Quando NÃO usar** - Legibilidade é mais importante que concisão

Principais lições: - List comprehensions são **rápidas e concisas** para transformações simples - Use **generators** para datasets grandes (economiza memória) - Mantenha **legibilidade** - se está complexo demais, use loop normal - Evite **side effects** em list comprehensions

Agora que você domina list comprehensions, **use com sabedoria**! Lembre-se: código legível é mais importante que código conciso.

Próximos passos: - Pratique com seus próprios dados - Explore Dict Comprehensions - Experimente Generator Expressions em arquivos grandes

Então… **Mão na massa**! 💪 💪

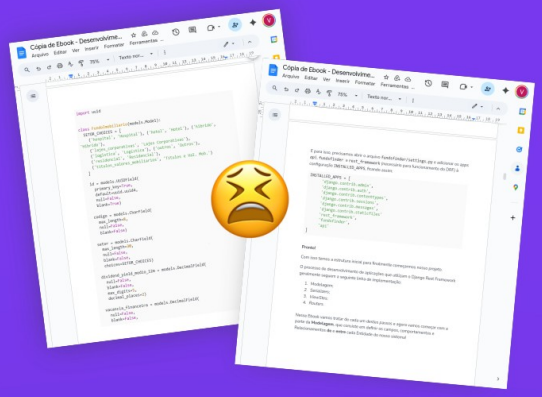
Até o próximo *post*!

Não se esqueça de conferir!

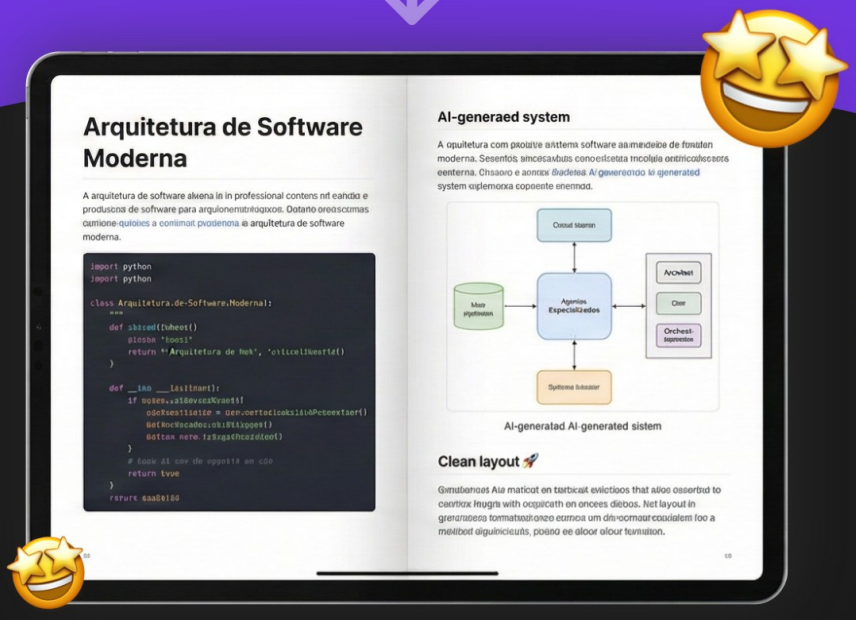
DevBook

Crie Ebooks técnicos em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código**!



Chega de formatar código no Google Docs



Syntax Highlight

Infográficos feitos por IA

Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado

Edite em Markdown em Tempo Real

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS