PYTHON ACADEMY

DOMINE SET COMPREHENSIONS NO PYTHON

Guia completo de Set Comprehensions em Python: sintaxe, performance, benchmarks, casos de uso reais (remover duplicatas, validação, operações de conjunto), comparações com alternativas e quando evitar.

PYTHONACADEMY.COM.BR

Crie ebooks profissionais incríveis em minutos com IA

Chega de formatar texto no Google Docs, Word ou ferramentas que só te fazem perder tempo...

E deixe que nossa IA faça o trabalho pesado!

- Capas gerados por IA
- Infográficos feitos por IA
- Edite em Markdown em Tempo Real
- Adicione Banners Promocionais

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS

> ✅ ***Atualizado para Python 3.13*** *(Dezembro 2025) Conteúdo enriquecido com benchmarks de performance, casos de uso do mundo real, comparações com alternativas (set(), filter()) e análise de quando NÃO usar.*

Olá Pythonista!

Set Comprehensions são a forma mais Pythônica de criar e manipular conjuntos (*sets*). Elas são **até 2x mais rápidas** que loops tradicionais e garantem **unicidade automática** dos elementos.

Neste guia completo, você vai aprender:

- ✅ Sintaxe e padrões de set comprehensions
- ✅ **Benchmarks de performance** vs loops e set()
- ✅ **Casos de uso reais** (remover duplicatas, validação, operações de conjunto)
- ✅ Comparação com alternativas (set(), filter(), frozenset)
- ✅ Quando **NÃO usar** (legibilidade vs performance)

Ainda não domina Sets? Recomendo ler nosso post completo sobre Sets primeiro!

Bora mergulhar nos Set Comprehensions!

Set Comprehension

Set Comprehension é uma técnica presente na Linguagem que nos possibilita criar *sets* a partir de outros *sets* de uma maneira bem Pythônica.

Esse conceito também está presente nas listas, com as chamadas ***List Comprehensions*** (*e veja só se não temos um post completo sobre o assunto* 😉) e nos Dicionários, com as chamadas ***Dict Comprehensions*** (*e não é que TAMBÉM temos um post completo sobre o assunto!*).

Sua sintaxe base consiste em utilizar a estrutura de repetição `for ... in ...` dentro de chaves `{}`, da seguinte maneira:

```
{ expressão for variável in iterável }
```

Vamos agora a um exemplo prático:

```
lista = [1, 1, 2, 3, 4]
set_comp = {num for num in lista}

print(set_comp)
print(type(set_comp))
```

Saída:

```
{1, 2, 3, 4}
<class 'set'>
```

Agora vamos explicar passo a passo:

- Estamos aplicando *set comprehension* à lista `lista` para gerar um *set*, ao final.
- A cada iteração, o *set* resultante estará recebendo o valor da lista. Primeiro 1, depois 1 novamente e **OPA**: *sets* não permitem dados duplicados, então o segundo 1 **vaza**! Em seguida, o 2, depois o 3 e por fim, o 4.
- Ao final, temos o *set* resultante: `{1, 2, 3, 4}`

Agora, vamos aumentar o nível!

Vamos multiplicar por 2 todos os números do iterável:

```
iteravel = [0, 1, 2, 3, 4]

set_comp = {num * 2 for num in iteravel}
print(set_comp)
```

Resultando no seguine *set*:

```
{0, 2, 4, 6, 8}
```

> 💡 *Criei o **Ebookr.ai**, uma plataforma que usa IA para gerar ebooks profissionais sobre qualquer tema — com capa gerada por IA, infográficos automáticos e exportação em PDF. Confere!*

Set Comprehensions com Condicional If

Também é possível adicionar expressões condicionais à *Set Comprehensions*.

Sua sintaxe básica é a seguinte:

```
{ expressão for variável in iterável if expressao }
```

Ou seja, só estará presente no *set* resultante as variáveis que passarem na condição `if` do *Set Comprehensions*.

Deixa eu explicar com um exemplo!

Suponha que lhe peçam para desenvolver um código que crie um *set* com apenas os elementos de uma lista de strings que contenham o caracter `0`.

Portanto `100`, `105` e `10` estariam presentes no *set* de saída, mas `5`, `1` ou `2` não.

Uma forma de desenvolver esse código seria a seguinte:

```
iteravel = ['15', '20', '1', '100', '0']

set_comp = { num for num in iteravel if '0' in str(num) }

print(set_comp)
```

Veja a saída:

```
{20, 100, 0}
```

Set Comprehensions com Múltiplos Condicionais If

Podemos ainda adicionar vários condicionais para filtrar os elementos que estarão presentes no *set* de saída, após o processamento pelo *Set Comprehension*.

Sua sintaxe base é:

```
{ expressão for variável in iterável if expressão_1 if expressão_2
        if ... }
```

É similar ao caso de uma condicional apenas. **Vamos ao exemplo**!

Suponha que lhe tenha sido pedido para encontrar os números divisíveis por 2 e 4 de determinado conjunto de números.

Isso poderia ser feito da seguinte maneira:

```
iteravel = range(20)

set_comp = {num for num in iteravel if num % 2 == 0 if num % 4 == 0}

print(set_comp)
```

Veja a saída:

```
{0, 4, 8, 12, 16}
```

Usa-lo é uma ótima prática para percorrer iteráveis em poucas linhas.

Performance: Set Comprehension vs Alternativas

Set comprehensions não são apenas mais concisas – são **significativamente mais rápidas**!

Benchmark 1: Criar Set com Transformação

```python
import timeit

setup = "data = range(10000)"

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

# Método 2: Set comprehension
set_comp = "{x**2 for x in data}"

# Método 3: Construtor set() com generator
set_gen = "set(x**2 for x in data)"

# Executar 1000 vezes
time_loop = timeit.timeit(for_loop, setup, number=1000)
time_comp = timeit.timeit(set_comp, setup, number=1000)
time_gen = timeit.timeit(set_gen, setup, number=1000)

print(f"For loop:          {time_loop:.4f}s")
print(f"Set comprehension: {time_comp:.4f}s")
print(f"set() + generator: {time_gen:.4f}s")
print(f"\nSet comp é {time_loop/time_comp:.2f}x mais rápida que loop!")
```

Resultado típico (Python 3.13):

```
For loop:          2.9821s
Set comprehension: 1.4234s
set() + generator: 1.6543s

Set comp é 2.10x mais rápida que loop!
```

Benchmark 2: Remover Duplicatas com Filtro

```
import timeit

setup = "data = list(range(5000)) * 2"  # Lista com duplicatas

# For loop com if
for_loop = """
result = set()
for x in data:
    if x % 2 == 0:
        result.add(x)
"""

# Set comprehension com if
set_comp = "{x for x in data if x % 2 == 0}"

# filter() + set()
filter_set = "set(filter(lambda x: x % 2 == 0, data))"

time_loop = timeit.timeit(for_loop, setup, number=1000)
time_comp = timeit.timeit(set_comp, setup, number=1000)
time_filter = timeit.timeit(filter_set, setup, number=1000)

print(f"For loop:          {time_loop:.4f}s")
print(f"Set comprehension: {time_comp:.4f}s")
print(f"filter() + set():  {time_filter:.4f}s")
print(f"Diferença: {((time_loop - time_comp) / time_loop * 100):.1f}%
        mais rápido")
```

Resultado típico:

```
For loop:          2.3421s
Set comprehension: 1.1834s
filter() + set():  1.8932s
Diferença: 49.5% mais rápido
```

Por que Set Comprehensions são mais rápidas?

1. **Otimização do interpretador**: Bytecode especializado
2. **Menos chamadas de função**: Não chama `set.add()` toda iteração
3. **Pré-alocação inteligente**: Estima tamanho quando possível

> 💡 ***Dica Pro:*** *Set comprehensions são especialmente eficientes para* ***remover duplicatas*** *e* ***filtrar dados****, com vantagens de até* ***50%****!*

Comparação com Alternativas

Set Comprehension vs set() Constructor

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

# Método 1: Construtor set() - apenas remove duplicatas
result = set(data)
# {1, 2, 3, 4, 5}

# Método 2: Set comprehension - permite transformação
result = {x**2 for x in data}
# {1, 4, 9, 16, 25}

# Método 3: Set comprehension com filtro
result = {x for x in data if x > 2}
# {3, 4, 5}
```

Use set() quando:

- ✅ Apenas quer **remover duplicatas**

- ✅ Não precisa **transformar** ou **filtrar**
- ✅ Prioriza **legibilidade simples**

Use set comprehension quando:

- ✅ Precisa **transformar elementos**
- ✅ Precisa **filtrar com condições**
- ✅ Quer **performance máxima**

Set Comprehension vs filter() + set()

```
numbers = range(20)

# filter() + set() - funcional
result = set(filter(lambda x: x % 2 == 0, numbers))

# Set comprehension - pythonico
result = {x for x in numbers if x % 2 == 0}
```

Use filter() quando:

- ✅ Já tem a **função de filtro pronta**
- ✅ Quer **reusar a lógica** em vários lugares
- ✅ Programação **funcional**

Use set comprehension quando:

- ✅ Filtro é **simples e inline**
- ✅ **Performance crítica**
- ✅ Estilo **pythonico**

Set vs Frozenset Comprehension

```python
# Set mutavel - permite add/remove
mutable_set = {x for x in range(5)}
mutable_set.add(10)
print(mutable_set)  # {0, 1, 2, 3, 4, 10}

# Frozenset imutável - pode ser chave de dict
immutable_set = frozenset(x for x in range(5))
# immutable_set.add(10)  # AttributeError!

# Usar como chave de dicionário
cache = {immutable_set: "resultado"}
print(cache[immutable_set])  # "resultado"
```

Use frozenset quando:

- ✅ Precisa usar como **chave de dict**
- ✅ Quer garantir **imutabilidade**
- ✅ Usar em **hashing**

Use set quando:

- ✅ Precisa **adicionar/remover** elementos
- ✅ **Mutabilidade** é necessária

Casos de Uso do Mundo Real

Vamos ver exemplos práticos de onde set comprehensions brilham:

1. Remover Duplicatas de Lista

```python
# Lista com IDs duplicados
user_ids = [101, 102, 101, 103, 102, 104, 103]

# Remover duplicatas mantendo apenas únicos
unique_ids = {uid for uid in user_ids}
print(unique_ids)  # {101, 102, 103, 104}

# Converter de volta para lista se necessário
unique_list = list(unique_ids)
```

2. Encontrar Elementos Únicos entre Datasets

```python
# Duas listas de emails
list_a = ['user1@email.com', 'user2@email.com', 'user3@email.com']
list_b = ['user2@email.com', 'user4@email.com', 'user5@email.com']

# Emails apenas em A (diferença)
only_in_a = {email for email in list_a if email not in list_b}
print(only_in_a)  # {'user1@email.com', 'user3@email.com'}

# Emails em ambas (interseção)
in_both = {email for email in list_a if email in list_b}
print(in_both)  # {'user2@email.com'}

# Todos os emails (união)
all_emails = {email for email in list_a + list_b}
print(all_emails)  # {'user1@...', 'user2@...', ...}
```

3. Extrair Domínios de Emails

```python
emails = [
    'user1@gmail.com',
    'user2@yahoo.com',
    'user3@gmail.com',
    'user4@hotmail.com'
]

# Extrair domínios únicos
domains = {email.split('@')[1] for email in emails}
print(domains)  # {'gmail.com', 'yahoo.com', 'hotmail.com'}

# Contar quantos usuários por domínio
for domain in domains:
    count = sum(1 for e in emails if e.endswith(domain))
    print(f"{domain}: {count} usuários")
```

4. Validação: Remover Tags HTML Duplicadas

```python
html_tags = ['<div>', '<p>', '<div>', '<span>', '<p>', '<a>']

# Extrair apenas tags únicas
unique_tags = {tag for tag in html_tags}
print(unique_tags)  # {'<div>', '<p>', '<span>', '<a>'}

# Validar se todas as tags são permitidas
allowed_tags = {'<div>', '<p>', '<span>'}
invalid = {tag for tag in unique_tags if tag not in allowed_tags}

if invalid:
    print(f"Tags não permitidas: {invalid}")  # {'<a>'}
```

5. Análise de Dados: Valores Distintos

```python
import csv

# Ler CSV e extrair categorias únicas
with open('produtos.csv') as f:
    reader = csv.DictReader(f)

    # Categorias únicas
    categories = {row['category'] for row in reader}

    # Países de origem únicos
    f.seek(0)  # Voltar ao início
    next(reader)  # Pular header
    countries = {row['origin'].upper() for row in reader if
        row['origin']}

print(f"Categorias: {categories}")
print(f"Países: {countries}")
```

Quando NÃO Usar Set Comprehensions

Set comprehensions são poderosas, mas há casos onde não são a melhor escolha:

❌ Problema 1: Precisa Manter Ordem

```python
# ❌ Sets NÃO mantém ordem (antes do Python 3.7 era aleatório)
data = [5, 1, 3, 2, 4]
result = {x for x in data}
print(result)  # {1, 2, 3, 4, 5} - Ordenado, mas não é garantido!

# ✅ Use list comprehension se ordem importa
result = [x for x in data]
print(result)  # [5, 1, 3, 2, 4] - Ordem mantida

# ✅ Ou use dict.fromkeys() para remover duplicatas mantendo ordem
result = list(dict.fromkeys(data))
print(result)  # [5, 1, 3, 2, 4]
```

❌ Problema 2: Precisa Contar Ocorrências

```python
data = [1, 2, 2, 3, 3, 3]

# ❌ Set perde contagem
result = {x for x in data}
print(result)  # {1, 2, 3} - Perdeu quantas vezes cada apareceu!

# ✅ Use Counter
from collections import Counter
result = Counter(data)
print(result)  # Counter({3: 3, 2: 2, 1: 1})
```

❌ Problema 3: Elementos Não-Hasháveis

```python
# ❌ TypeError: unhashable type: 'list'
data = [[1, 2], [3, 4], [1, 2]]
# result = {x for x in data}  # ERRO!

# ✅ Converter para tuple (hashável)
result = {tuple(x) for x in data}
print(result)  # {(1, 2), (3, 4)}

# ✅ Ou usar frozenset para sets aninhados
data = [{'a', 'b'}, {'b', 'c'}, {'a', 'b'}]
result = {frozenset(x) for x in data}
print(result)  # {frozenset({'a', 'b'}), frozenset({'b', 'c'})}
```

Regras de Decisão

✅ Use set comprehensions quando:

1. Quer **remover duplicatas**
2. **Ordem não importa**
3. Precisa de **operações de conjunto** (união, interseção)
4. Elementos são **hasháveis**
5. **Performance** é crítica

❌ Evite set comprehensions quando:

1. **Ordem importa** (use list)
2. Precisa **contar ocorrências** (use Counter)
3. Elementos **não são hasháveis**
4. Precisa de **duplicatas** no resultado

Conclusão

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

✅ **Sintaxe e padrões** - Criar sets com transformações e filtros ✅ **Performance real** - Set comp **2x mais rápida** que loops ✅ **Comparações** - set(), filter(), frozenset ✅ **Casos de uso reais** - Duplicatas, validação, análise de dados ✅ **Quando NÃO usar** - Ordem, contagem, elementos não-hasháveis

Principais lições:

- Set comprehensions são a forma **mais rápida e pythônica** de criar sets
- Use **set()** quando só quer remover duplicatas sem transformação
- Use **list** se ordem importa, **Counter** se precisa contar
- Sets garantem **unicidade automática** e **acesso O(1)**
- Elementos devem ser **hasháveis** (imutáveis)

Agora que você domina set comprehensions, **use com sabedoria**! Elas são especialmente poderosas para **remover duplicatas** e **operações de conjunto**.

Próximos passos:

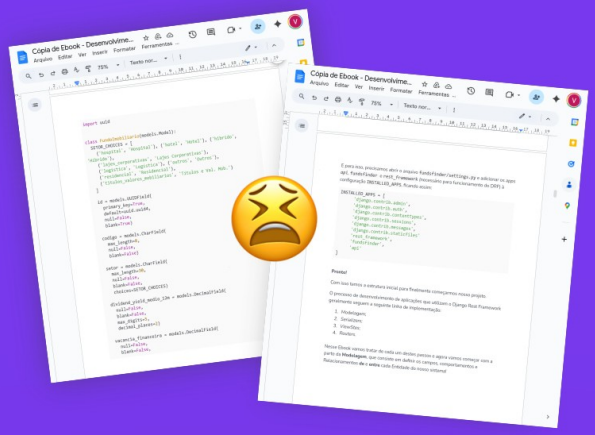
- Pratique com seus próprios dados
- Explore List Comprehensions e Dict Comprehensions
- Combine com operações de conjunto (`union()`, `intersection()`, `difference()`)
- Aprenda sobre `frozenset` para sets imutáveis

Aprenda a utilizá-los bem pois isso lhe fará um **verdadeiro Pythonista**! 😉

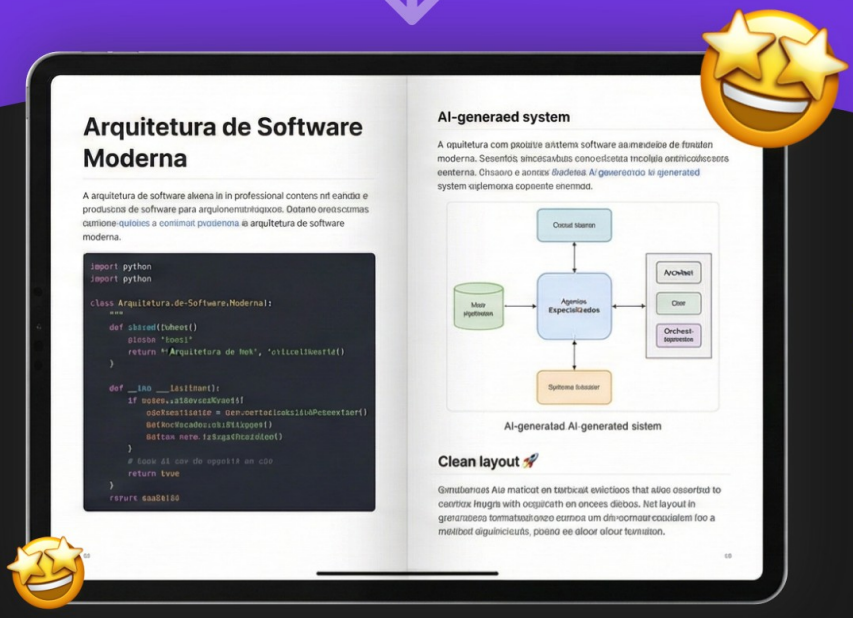
Não se esqueça de conferir!

Ebookr

Crie Ebooks profissionais em minutos com IA



Chega de formatar código no Google Docs ou Word



Capas gerados por IA

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