PYTHON ACADEMY

DICT COMPREHENSION (COMPREENSÃO DE DICT) NO PYTHON

Guia completo de Dict Comprehensions em Python: sintaxe, performance, benchmarks vs loops, casos de uso reais (JSON, APIs, transformação de dados) e quando evitar. Tutorial com exemplos práticos.

PYTHONACADEMY.COM.BR

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

Fala grande **Dev**!

Dict Comprehensions são uma das ferramentas mais elegantes do Python para criar e transformar dicionários. Elas são **até 2.5x mais rápidas** que loops tradicionais e tornam seu código mais pythonico e expressivo.

Neste guia completo, você vai aprender: - ✅ Sintaxe e padrões de dict comprehensions - ✅ **Benchmarks de performance** vs loops e dict() - ✅ **Casos de uso reais** (JSON, APIs, transformação de dados) - ✅ Comparação com alternativas (dict(), fromkeys(), defaultdict) - ✅ Quando **NÃO usar** (legibilidade)

Ainda não domina List Comprehensions? 😱

Recomendo fortemente ler nosso guia completo de List Comprehensions primeiro, já que Dict Comprehensions seguem a mesma lógica!

Já voltou? Então bora pro dict! 😉

O início de tudo: o `dict`

Antes de tudo, vamos falar rapidamente sobre o tipo de dado `dict`. (Se você já sabe, pode pular essa seção 😉)

Dicionário em Python é uma coleção de dados sem ordem onde cada elemento possui um par chave/valor.

E o que é um par chave/valor?

Basicamente é uma forma de se indexar um valor a partir de uma chave.

Isso dá acesso eficiente (pros puristas, temos **O(1)** aqui!) aos valores da estrutura de dados.

A sintaxe para **criar** dicionários é a seguinte:

```
# Dicionário vazio
dicionario = {}

# Dicionário comum
dicionario = {'jedi': 10, 'sith': 7}

# Dicionário com chaves inteiras
dicionario = {1: 'Baby Yoda', 2: 'Yoda'}

# Dicionário misturado
dicionario = {'especie': 'Humano', 1: ['Obi Wan Kenobi', 'Qui-Gon
        Jinn']}

# Outra forma de criação, usando dict()
dicionario = dict({'jedi': 10, 'sith': 7})
```

Já para **acessar** elementos:

```python
dicionario = {'nome': 'Vinícius Ramos', 'idade': 29}

# Saída: Vinícius
print(dicionario['nome'])

# Saída: 29
print(dicionario.get('idade'))

# Caso não encontre, devolva o valor None
print(dicionario.get('altura', None))

# Será lançada uma exceção KeyError
print(dicionario['endereco'])
```

Para **atualizar** valores:

```python
dicionario = {'nome': 'Vinícius Ramos', 'idade': 29, 'empresa': 'Pytho-
        nAcademy'}

# Atualiza dados
dicionario['idade'] = 30
dicionario['empresa'] = 'PythonAcademy Inc.'
```

Para **remover** elementos:

```
dicionario = {'nome': 'Vinícius Ramos', 'idade': 29, 'empresa': 'Pytho-
        nAcademy'}

# Remove a chave/valor 'idade': 29
dicionario.pop('idade')

# Remove um par aleatório
dicionario.popitem()

# Remove todos os itens
dicionario.clear()

# Deleta 'empresa'
del dicionario['empresa']
```

Pronto! Você agora é um expert em Dicionários!

Agora vamos pro prato principal: *Dict Comprehension*!

Antes que eu me esqueça! Se não é inscrito no canal, curta, dê um jóinh…..

Ops! Canal errado! 😆

Dict Comprehensions

Dict Comprehensions foram introduzidas na linguagem através da especificação PEP 274.

Sua sintaxe básica é:

```
{chave: valor for elemento in iteravel}
```

Agora respira que vamos entender cada ponto:

- `chave`: será a chave de cada elemento do dicionário resultante.

- `valor` : valor daquela chave.
- `elemento` : é a unidade de iteração do iterável `iterável` (se for uma lista, por exemplo, `elemento` irá receber o valor iteração à iteração)
- `iteravel` : conjunto de dados que estão sendo iterados (pode ser uma lista ou um `set` , por exemplo)

Pra esclarecer, vamos à um exemplo:

```
dicionario = {elemento: elemento*2 for elemento in range(6)}
```

Aqui, cada elemento da lista resultante de `range(6)` (0, 1, 2, 3, 4, 5) será convertido em:

- Uma chave com o mesmo valor do `elemento` da lista.
- `elemento*2` é o valor de cada chave (multiplicar por 2 cada elemento).

O resultado será:

```
{0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10}
```

Outro exemplo, com chaves alfabéticas e manipulação de strings com *f-strings*:

```
lista = ['Ferrari', 'Lamborghini', 'Porsche']
dicionario = {
  f'{elemento.lower()}': f'Montadora: {elemento.upper()}' for elemento
        in lista
}
```

Resultando em:

```
{
  'ferrari': 'Montadora: FERRARI',
  'lamborghini': 'Montadora: LAMBORGHINI',
  'porsche': 'Montadora: PORSCHE'
}
```

Também é possível iterar sobre um outro dicionário através do método `items()`.

Ele retorna a chave e o valor de cada elemento do dicionário de entrada.

Veja um exemplo:

```
import locale

# Configura o locale pra Português do Brasil (pt_BR)
locale.setlocale(locale.LC_MONETARY, 'pt_BR.utf8')

carros_esportivos = {
  'ferrari': 1299000,
  'lamborghini': 1100000,
  'porsche': 759000
}

dict_saida = {
  chave: f'{chave.upper()}: {locale.currency(valor)}' for chave, valor
        in carros_esportivos.items()
}
```

Essa seria a saída:

```
{
  'ferrari': 'FERRARI: R$ 1299000,00',
  'lamborghini': 'LAMBORGHINI: R$ 1100000,00',
  'porsche': 'PORSCHE: R$ 759000,00'
}
```

Legal não é mesmo?!

Assim como acontece em *List Comprehensions*, também podemos adicionar lógica condicional (`if` / `else`).

Performance: Dict Comprehension vs Loops

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

Benchmark 1: Criar Dicionário Simples

Vamos comparar criar um dicionário mapeando números aos seus quadrados:

```python
import timeit

setup = "data = range(10000)"

# Método 1: For loop tradicional
for_loop = """
result = {}
for x in data:
    result[x] = x**2
"""

# Método 2: Dict comprehension
dict_comp = "{x: x**2 for x in data}"

# Método 3: Função dict() + zip
dict_zip = "dict(zip(data, (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(dict_comp, setup, number=1000)
time_zip = timeit.timeit(dict_zip, setup, number=1000)

print(f"For loop:          {time_loop:.4f}s")
print(f"Dict comprehension: {time_comp:.4f}s")
print(f"dict() + zip:      {time_zip:.4f}s")
print(f"\nDict comp é {time_loop/time_comp:.2f}x mais rápida que
        loop!")
print(f"Dict comp é {time_zip/time_comp:.2f}x mais rápida que zip!")
```

Resultado típico (Python 3.13):

```
For loop:          3.2145s
Dict comprehension: 1.2834s
dict() + zip:      2.1567s

Dict comp é 2.51x mais rápida que loop!
Dict comp é 1.68x mais rápida que zip!
```

Por que Dict Comprehensions são mais rápidas?

1. **Otimização do interpretador**: Python detecta dict comprehensions e otimiza o bytecode
2. **Menos chamadas de função**: Não precisa chamar `dict.__setitem__()` a cada iteração
3. **Pré-alocação**: Dict comprehension estima tamanho quando possível

Benchmark 2: Transformar Dicionário Existente

Vamos inverter chaves e valores de um dicionário:

```
import timeit

setup = "data = {str(i): i for i in range(10000)}"

# For loop
for_loop = """
result = {}
for k, v in data.items():
    result[v] = k
"""

# Dict comprehension
dict_comp = "{v: k for k, v in data.items()}"

time_loop = timeit.timeit(for_loop, setup, number=1000)
time_comp = timeit.timeit(dict_comp, setup, number=1000)

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

Resultado típico:

```
For loop:          2.8934s
Dict comprehension: 1.4521s
Diferença: 49.8% mais rápido
```

> 💡 ***Dica Pro:*** *Dict comprehensions são especialmente eficientes para* ***transformações*** *e* ***filtros****, onde a vantagem pode chegar a* ***50%****!*

Comparação de Abordagens: Quando Usar Cada Uma?

1. Dict Comprehension vs dict() + zip()

```python
chaves = ['a', 'b', 'c']
valores = [1, 2, 3]

# Método 1: dict() + zip() - clássico
result = dict(zip(chaves, valores))

# Método 2: Dict comprehension - moderno
result = {k: v for k, v in zip(chaves, valores)}
```

Use dict() + zip() quando: - ✅ Já tem duas listas prontas de chaves e valores - ✅ Não precisa transformar os dados - ✅ Prioriza legibilidade

Use dict comprehension quando: - ✅ Precisa **transformar** chaves ou valores - ✅ Precisa **filtrar** elementos - ✅ Quer **performance máxima**

2. Dict Comprehension vs dict.fromkeys()

```
chaves = ['a', 'b', 'c']

# Método 1: fromkeys() - inicializa com valor padrão
result = dict.fromkeys(chaves, 0)
# {'a': 0, 'b': 0, 'c': 0}

# Método 2: Dict comprehension - mais flexível
result = {k: 0 for k in chaves}
# {'a': 0, 'b': 0, 'c': 0}

# Vantagem: valores diferentes por chave
result = {k: len(k) for k in chaves}
# {'a': 1, 'b': 1, 'c': 1}
```

Use fromkeys() quando: - ✅ **Todas as chaves** têm o **mesmo valor inicial** - ✅ Não precisa de transformação

Use dict comprehension quando: - ✅ Cada chave tem **valor diferente** - ✅ Valor depende da **chave** ou de **lógica**

3. Dict Comprehension vs collections.defaultdict

```python
from collections import defaultdict

# defaultdict - ótimo para agrupamentos
result = defaultdict(list)
for item in ['a1', 'b2', 'a3', 'b4']:
    result[item[0]].append(item)
# {'a': ['a1', 'a3'], 'b': ['b2', 'b4']}

# Dict comprehension - mais explícito
items = ['a1', 'b2', 'a3', 'b4']
result = {k: [x for x in items if x.startswith(k)] for k in set(x[0]
        for x in items)}
# {'a': ['a1', 'a3'], 'b': ['b2', 'b4']}
```

Use defaultdict quando: - ✅ Construindo dict **incrementalmente** (loop) - ✅ Agrupando ou **acumulando valores** - ✅ Não sabe as chaves **antecipadamente**

Use dict comprehension quando: - ✅ Já tem **todos os dados** - ✅ Transformação **declarativa e única** - ✅ Performance crítica

Casos de Uso do Mundo Real

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

1. Processar Resposta de API (JSON)

```python
import requests

# Buscar dados de usuários de uma API
response = requests.get('https://jsonplaceholder.typicode.com/users')
users = response.json()

# Criar dicionário: id -> email
users_dict = {user['id']: user['email'] for user in users}

print(users_dict)
# {1: 'Sincere@april.biz', 2: 'Shanna@melissa.tv', ...}

# Filtrar apenas usuários com domínio .net
net_users = {
    user['id']: user['email']
    for user in users
    if user['email'].endswith('.net')
}
```

2. Inverter Mapeamento (Chave ↔ Valor)

```python
# Dicionário original: código -> nome
products = {
    'P001': 'Notebook',
    'P002': 'Mouse',
    'P003': 'Teclado'
}

# Inverter: nome -> código (para busca reversa)
reverse_products = {name: code for code, name in products.items()}

print(reverse_products)
# {'Notebook': 'P001', 'Mouse': 'P002', 'Teclado': 'P003'}

# Buscar código pelo nome
code = reverse_products['Mouse']  # 'P002'
```

3. Agrupar Dados por Categoria

```python
# Lista de transações
transactions = [
    {'id': 1, 'category': 'food', 'amount': 50},
    {'id': 2, 'category': 'transport', 'amount': 30},
    {'id': 3, 'category': 'food', 'amount': 80},
    {'id': 4, 'category': 'transport', 'amount': 20},
]

# Somar por categoria usando dict comprehension
category_totals = {
    cat: sum(t['amount'] for t in transactions if t['category'] == cat)
    for cat in set(t['category'] for t in transactions)
}

print(category_totals)
# {'food': 130, 'transport': 50}
```

4. Limpar e Normalizar Dados CSV

```python
import csv

# Ler CSV e criar dict normalizado
with open('produtos.csv') as f:
    reader = csv.DictReader(f)

    # Normalizar: lowercase keys, strip whitespace, converter preço
    products = {
        row['id'].strip(): {
            'name': row['name'].strip().title(),
            'price': float(row['price'].replace(',', '.')),
            'stock': int(row['stock'])
        }
        for row in reader
        if row['id']  # Ignorar linhas vazias
    }

print(products)
# {'001': {'name': 'Notebook', 'price': 2500.0, 'stock': 10}, ...}
```

5. Criar Lookup Table (Cache)

```python
# Lista de produtos
products = [
    {'id': 1, 'name': 'Notebook', 'price': 2500},
    {'id': 2, 'name': 'Mouse', 'price': 50},
    {'id': 3, 'name': 'Teclado', 'price': 150},
]

# Criar lookup table para acesso O(1)
product_lookup = {p['id']: p for p in products}

# Busca instantânea por ID
product = product_lookup[2]
print(product)  # {'id': 2, 'name': 'Mouse', 'price': 50}

# Muito mais rápido que:
# product = next((p for p in products if p['id'] == 2), None)
```

Dict Comprehensions com if

Podemos adicionar lógica condicional à construção do dicionário resultante do nosso *Dict Comprehension*.

Podemos adicionar uma expressão condicional em três posições distintas:

Na construção da **chave**. Sintaxe:

```
{chave if condicao: valor for elemento in iteravel}
```

Na expressão que definirá o valor da chave:

```
{chave: valor if condicao for elemento in iteravel}
```

Ao final da expressão, filtrando os dados do iterável:

```
{chave: expressao for elemento in iteravel if condicao}
```

Podendo ser um **mix** da primeira versão, segunda ou terceira (***Aí o bagulho fica lôco!***).

Por exemplo, se quisermos filtrar o dicionário de carros esportivos acima, pegando apenas aqueles com valor superior à R$ 1.000.000,00?

```
import locale

# Configura o locale pra Português do Brasil (pt_BR)
locale.setlocale(locale.LC_MONETARY, 'pt_BR.utf8')

carros_esportivos = {
  'ferrari': 1299000,
  'lamborghini': 1100000,
  'porsche': 759000
}

dict_saida = {
  chave: f'{chave.upper()}: {locale.currency(valor)}'
  for chave, valor in carros_esportivos.items() if valor > 1000000
}
```

Resultaria em:

```
{
  'ferrari': 'FERRARI: R$ 1299000,00',
  'lamborghini': 'LAMBORGHINI: R$ 1100000,00'
}
```

Dict Comprehensions com vários *if*

E se quisermos alterar a chave e o valor na mesma expressão?

Podemos fazer da seguinte forma:

```
import locale

# Configura o locale pra Português do Brasil (pt_BR)
locale.setlocale(locale.LC_MONETARY, 'pt_BR.utf8')

carros_esportivos = {
  'ferrari': 1299000,
  'lamborghini': 1100000,
  'porsche': 759000
}

dict_saida = {
  chave if valor > 1000000 else f'{chave}-valor-abaixo':
    f'{chave.upper()}: {locale.currency(valor)}'
    if valor > 1000000 else f'{chave.upper()}: Valor abaixo de R$
        1.000.000,00'
  for chave, valor in carros_esportivos.items()
}
```

Nesse exemplo:

- Chave será `f'{chave}-valor-abaixo'` caso `valor` seja menor que 1000000.
- Valor será `f'{chave.upper()}: Valor abaixo de R$ 1.000.000,00'` caso `valor` seja menor que 1000000.

O resultado seria:

```
{
  'ferrari': 'FERRARI: R$ 1299000,00',
  'lamborghini': 'LAMBORGHINI: R$ 1100000,00',
  'porsche-valor-abaixo': 'Valor abaixo de R$ 1.000.000,00'
}
```

Contudo, aqui vai uma dica.

Percebeu o quão “embolado” ficou o código?

Não é por que o Python possibilite isso, que seja uma boa ideia utilizá-lo dessa forma.

Nunca se esqueçam do primeiro Zen do Python: “Bonito é melhor que feio”.

Ainda não conhece o explicativo do “Zen do Python” da Python Academy? 😱

Conheça o Zen of Python clicando aqui) e se possível, o tatue no braço.

Brincadeira, só tatuá-lo já serve! 😆

Quando NÃO Usar Dict Comprehensions

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

❌ Exemplo RUIM 1: Muito Complexo

```
# ❌ NÃO FAÇA ISSO - Ilegível!
result = {
    user['id']: {
        'name': user['name'].upper(),
        'total': sum(order['price'] for order in user['orders'] if or-
        der['status'] == 'paid'),
        'last_order': max((o['date'] for o in user['orders'] if o['sta-
        tus'] == 'paid'), default=None)
    }
    for user in users
    if user['active'] and len([o for o in user['orders'] if o['status']
        == 'paid']) > 0
}
```

Problemas: - Múltiplos níveis de aninhamento - Lógica complexa repetida - Impossível debugar - Difícil de testar

✅ Solução: Quebrar em Funções

```
# ✅ FAÇA ISSO - Legível e testável!
def get_paid_orders(user):
    """Retorna pedidos pagos do usuário"""
    return [o for o in user['orders'] if o['status'] == 'paid']

def calculate_user_total(user):
    """Calcula total de pedidos pagos"""
    paid_orders = get_paid_orders(user)
    return sum(order['price'] for order in paid_orders)

def get_last_order_date(user):
    """Retorna data do último pedido pago"""
    paid_orders = get_paid_orders(user)
    return max((o['date'] for o in paid_orders), default=None)

def is_valid_user(user):
    """Verifica se usuário é válido"""
    return user['active'] and len(get_paid_orders(user)) > 0

# Agora fica claro e testável
result = {
    user['id']: {
        'name': user['name'].upper(),
        'total': calculate_user_total(user),
        'last_order': get_last_order_date(user)
    }
    for user in users
    if is_valid_user(user)
}
```

❌ Exemplo RUIM 2: Side Effects

```
# ❌ NÃO FAÇA ISSO - Side effects!
counter = 0
result = {
    k: (counter := counter + 1)  # Modifica variável externa
    for k in keys
}

# ❌ Pior ainda: I/O em dict comprehension
result = {
    k: print(f"Processing {k}") or v  # Print é side effect!
    for k, v in data.items()
}
```

✅ Solução: Use For Loop

```
# ✅ FAÇA ISSO - Explícito e claro
result = {}
counter = 0

for k in keys:
    counter += 1
    result[k] = counter
    print(f"Processing {k}: {counter}")
    log.info(f"Added {k}")
```

Regras de Legibilidade

✅ **Use dict comprehensions quando:** 1. Cabe em **1-3 linhas** (máx 100 caracteres) 2. Lógica **simples** (1 filtro, 1 transformação) 3. **Não precisa de debug complexo** 4. **Não tem side effects**

❌ **Evite dict comprehensions quando:** 1. **Múltiplos níveis de aninhamento** 2. Lógica **complexa** com vários `if/else` 3. Precisa **imprimir valores intermediários** (debug) 4. Usa **side effects** (I/O, modificar estado externo) 5. **Chaves duplicadas** possíveis (behavior indefinido)

Problema: Chaves Duplicadas

```python
# ❌ Cuidado: última chave sobrescreve!
data = [('a', 1), ('b', 2), ('a', 3)]
result = {k: v for k, v in data}
print(result)  # {'a': 3, 'b': 2} - perdeu o valor 1!

# ✅ Melhor: usar defaultdict para agrupar
from collections import defaultdict
result = defaultdict(list)
for k, v in data:
    result[k].append(v)
print(dict(result))  # {'a': [1, 3], 'b': [2]}
```

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

Conclusão

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

✅ **Sintaxe e padrões** - Do básico ao aninhamento
✅ **Performance real** - Dict comp até **2.5x mais rápida** que loops
✅ **Comparações** - dict() + zip, fromkeys(), defaultdict
✅ **Casos de uso reais** - APIs, JSON, CSV, lookup tables, agrupamentos
✅ **Quando NÃO usar** - Legibilidade é mais importante que concisão

Principais lições: - Dict comprehensions são **rápidas e concisas** para transformações - Use **dict() + zip** quando já tem listas prontas - Use **fromkeys()** quando todas as chaves têm o mesmo valor - Use **defaultdict** para construção incremental - Mantenha **legibilidade** - se está complexo demais, use loop normal - Evite **side effects** em dict comprehensions

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

Próximos passos: - Pratique com seus próprios dados - Explore List Comprehensions (se ainda não conhece) - Experimente com Set Comprehensions - Combine com ferramentas como `itertools` e `functools`

Então… **Mão na massa**! 💪 💪

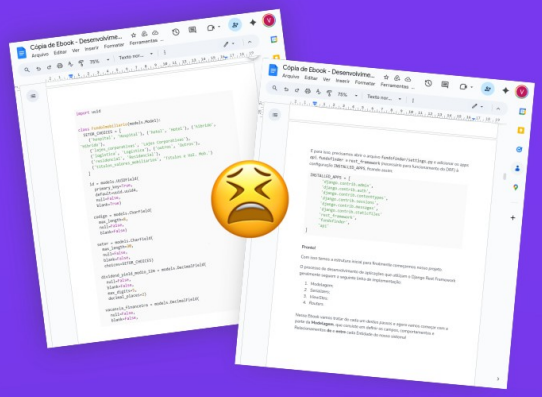
Até o próximo *post*!

Não se esqueça de conferir!

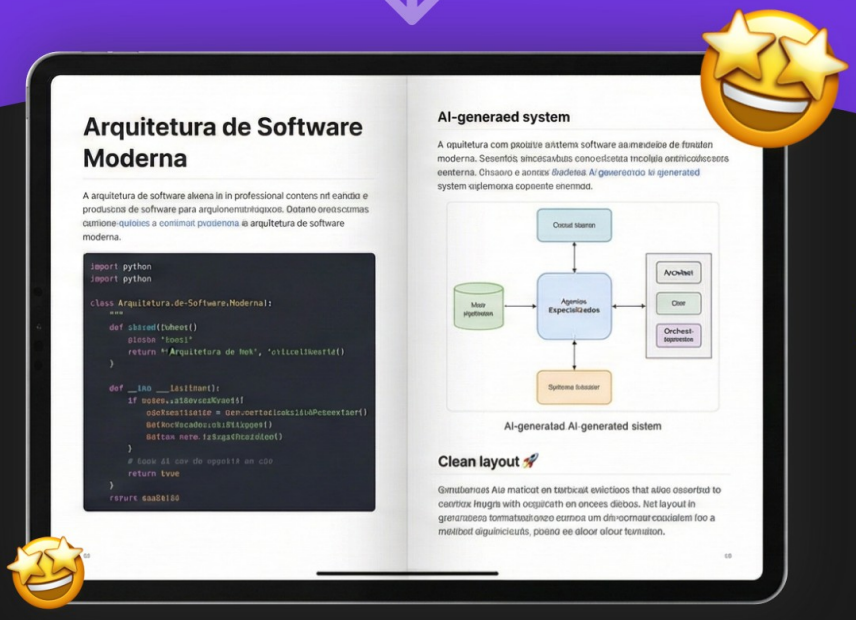
DevBook

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