



ITERATORS E GENERATORS DO PYTHON

Guia completo de Iterators e Generators em Python: protocolo Iterator, yield, lazy evaluation, economia de memória, casos de uso reais (processar arquivos grandes, streams) e quando usar cada um.

Crie Ebooks técnicos incríveis em minutos com IA

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



Chega de formatar código no Google Docs



Deixe que nossa IA faça o trabalho pesado

 Syntax Highlight

 Adicione Banners Promocionais

 Edite em Markdown em Tempo Real

 Infográficos feitos por IA

TESTE AGORA 

 **Atualizado para Python 3.13** (Dezembro 2025)

Conteúdo enriquecido com comparação de memória (Generator vs List), casos de uso do mundo real e análise de quando usar cada abordagem.

Olá pessoal!

Iterators e Generators são duas das ferramentas mais poderosas do Python para trabalhar com sequências de dados de forma **eficiente em memória**. Generators podem economizar **até 99.9% de memória** comparado a listas!

Neste guia, você vai aprender: -  Protocolo Iterator e como criar seus próprios iterators -  Generators com `yield` - sintaxe simplificada -  Lazy evaluation - processamento sob demanda -  Comparação de memória (Generator vs List) -  Casos de uso reais (arquivos grandes, streams, pipelines de dados) -  Quando usar Iterator vs Generator vs List

#VamosNessa!

Introdução

Se você já passou do “Hello World” do Python, você com certeza já se viu fazendo *loops* e, portanto, já deve ter utilizado *iterators*.

Objetos iteráveis (*iterators*) são objetos que estão em conformidade com o protocolo *Iterator* (criado na [PEP 234](#)) e podem, dessa forma, serem usados em *loops*.

No seguinte exemplo:

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

O `range(5)` é um objeto iterável (que pode ser usado em uma estrutura de repetição) que provê, a cada iteração (ou ciclo do *loop*), um valor diferente à variável `i`.

Até aqui, tranquilo...

E se você quiser criar um objeto iterável por conta própria?

O protocolo *Iterator*

O protocolo *Iterator* facilita muito a criação de um objeto iterável.

Para criá-lo, codificamos uma classe e basta que ela implemente os seguintes métodos:

- `__iter__` : Esse método deve retornar o próprio objeto (`self`) para ser utilizado em *loops* com `for` e `in`.
- `__next__` : Esse método deve retornar o próximo valor da iteração. Caso a condição de parada seja satisfeita, ou seja, quando não houver mais objetos a iterar, ela deve lançar o erro `StopIteration`.

Python sempre simples! 😊

Vamos criar um exemplo.

Vamos criar uma classe que nos permita iterar sobre a sequência de Fibonacci.

Relembrando: A sequência de Fibonacci é uma sequência de números inteiros onde um número, após os dois primeiros números (que são 0 e 1), é a soma dos últimos dois. Assim:

0, 1, 1, 2, 3, 5, 8, 13, 21,...

Com isso, e sabendo que devemos criar os métodos `__iter__` e `__next__`, podemos codificá-lo da seguinte forma:

```

class Fibonacci:
    def __init__(self, maximo=1000000):
        # Inicializa os dois primeiros números
        self.elemento_atual, self.proximo_elemento = 0, 1
        self.maximo = maximo

    def __iter__(self):
        # Retorna o objeto iterável (ele próprio: self)
        return self

    def __next__(self):
        # Fim da iteração, raise StopIteration
        if self.elemento_atual > self.maximo:
            raise StopIteration

        # Salva o valor a ser retornado
        valor_de_retorno = self.elemento_atual

        # Atualiza o próximo elemento da sequência
        self.elemento_atual, self.proximo_elemento = self.proximo_elemento, self.elemento_atual + self.proximo_elemento

        return valor_de_retorno

# Executa nosso código
if __name__ == '__main__':
    # Cria nosso objeto iterável
    objeto_fibonacci = Fibonacci(maximo=1000000)

    # Itera nossa sequência
    for fibonacci in objeto_fibonacci:
        print("Sequencia: {}".format(fibonacci))

```

No código acima:

- Inicializamos o elemento atual, o próximo elemento e o valor máximo da nossa sequência no construtor `__init__`.
- Retornamos `self` no método `__iter__`, conforme citado lá em cima.

- No método `__next__`, primeiro verificamos se a condição de parada foi satisfeita (caso positivo, lançamos a exceção `StopIteration`). Em seguida, atualizamos os valores atual e próximo para iteração seguinte, e retornamos o valor de retorno da iteração atual.

Vejam a simplicidade do Python.

Diferente do Java ou outras linguagens, onde temos que herdar uma classe ou implementar uma interface, tornando nosso código muito mais extenso e verboso, em Python, basta definir o comportamento (`__iter__` e `__next__`) que a linguagem já entende que estamos implementando o protocolo *Iterator*.

Mágico né?!

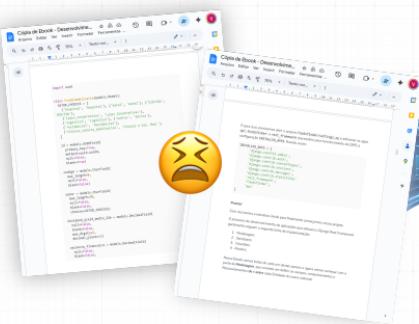
Com isso, podemos passar para a melhor parte: os *Generators*!



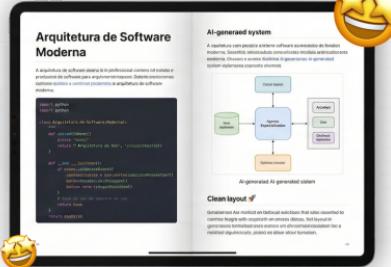
*Estou desenvolvendo o **DevBook**, uma plataforma que usa IA para gerar ebooks técnicos profissionais. Te convido a conhecer clicando no botão abaixo!*

Crie Ebooks técnicos incríveis em minutos com IA

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



Chega de formatar código no Google Docs



Deixe que nossa IA faça o trabalho pesado

 Syntax Highlight

 Adicione Banners Promocionais

 Edite em Markdown em Tempo Real

 Infográficos feitos por IA

TESTE AGORA! PRIMEIRO CAPÍTULO 100% GRÁTIS 

Generator

O conceito de *Generator* em Python, criado pela [PEP 255](#), é uma forma mais simples e rápida de se implementar o protocolo *Iterator*, pois não necessita a criação de uma classe para tal.

Para criar um *Generator* basta definir uma função e utilizar a palavra reservada `yield`, ao invés de `return`.

Vamos ver como criar a nossa sequência de Fibonacci, mas agora utilizando o conceito de *Generator*:

```

def fibonacci(maximo):
    # Inicialização dos elementos
    elemento_atual, proximo_elemento = 0, 1

    # Defina a condição de parada
    while elemento_atual < maximo:
        # Retorna o valor do elemento atual
        yield elemento_atual

        elemento_atual, proximo_elemento = \
            proximo_elemento, elemento_atual + proximo_elemento

if __name__ == '__main__':
    # Cria um generator de números fibonacci menor que 1 milhão
    fibonacci_generator = fibonacci(1000000)

    # Mostra na tela toda a sequencia
    for numero_fibonacci in fibonacci_generator:
        print(numero_fibonacci)

```

Vamos entender melhor o **fluxo de execução** para fixar o entendimento:

- Na primeira chamada à nossa função `fibonacci()`, o Python vai executar da **linha 1** à **linha 8** com `elemento_atual = 0` e `proximo_elemento = 1`.
- Como o *generator* salva o estado da função no momento do retorno (`yield`), a segunda execução não começará na linha 1, mas sim na linha subsequente ao `yield`, ou seja, a partir da **linha 10**. Em seguida continua a repetição dentro do `while` (linha 6), mas agora com o valor atualizado de `elemento_atual, retorna em yield`` e assim sucessivamente.
- Diferente da implementação por classe, onde tivemos que lançar um erro `StopIteration` para sinalizar o fim da iteração, aqui o fim da execução é sinalizado, apenas, por não retornar um valor. Ou seja, a primeira vez que o código do nosso *generator* não retornar um valor, o Python entende que esse é o fim da iteração e finaliza o `for / in` da **linha 18**.

Ficou claro? Entender esse fluxo é **MUITO importante!** Qualquer dúvida, não hesite em postar no *box* de comentários aqui embaixo!

Lazy Evaluation

Outro ponto importante para se ressaltar aqui, é o conceito de *lazy evaluation* (“avaliação preguiçosa” em português).

Iterators e Generators não computam todos os valores do seu *loop* quando são criados ou instanciados.

Eles computam **sob demanda**, isto é: **APENAS** quando pedimos o próximo valor da sequência.

Nos exemplos acima (com a classe Fibonacci e com a função `fibonacci()`), o Python não calculou a sequência inteira até o milionésimo número (`maximo=10000000`), no momento de sua criação.

O que ele fez foi calcular o primeiro número da sequência, aguardar o próximo ciclo do *loop*, calcular o segundo número, aguardar o próximo ciclo e assim sucessivamente, até que a condição de parada (`elemento_atual < maximo`) fosse alcançada.

Portanto, a cada iteração do loop nas **linhas 19 e 20**, nosso *generator* gera apenas **um novo número**, utilizando o passado e o atual, salvando assim, a preciosa memória da sua máquina.

Uma forma de se percebir isso, é chamando a função `next()` do próprio Python (*built-in*) diversas vezes.

Essa função traz o próximo item do objeto iterável. Por exemplo:

```
# Cria um generator de números fibonacci menor que mil
fibonacci_generator = fibonacci(1000)

next(fibonacci_generator)
next(fibonacci_generator)
next(fibonacci_generator)
next(fibonacci_generator)
```

Gera a seguinte saída:

```
0
1
1
2
```

Observação: Não tente reutilizar o *generator* após iterar sobre ele. Um *generator* só pode ser consumido uma única vez. Para utilizá-lo novamente, é necessário criá-lo ou instanciá-lo novamente.

Comparação de Memória: Generator vs Lista

A **maior vantagem** dos generators é a economia de memória. Vamos ver números reais:

```

import sys

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

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

# 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
          0.101562 KB

Lista usa 81,238x MAIS memória!

```

Conclusão: Generator usa **~100 bytes** independente do tamanho, enquanto lista cresce proporcionalmente!

Generators e Listas

Listas podem ser criadas através de objetos iteráveis.

Por exemplo, como `range` é iterável, o código: `list(range(5))` gera a seguinte saída:

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

Como um *generator* também é iterável, o código `list(fibonacci(1000))` gera (**adivinha**) a seguinte saída:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

Melhor ainda, utilizando *List Comprehensions* (assunto para um próximo *post*), podemos fazer, por exemplo:

```
fibonacci_impares = [x for x in fibonacci(100000) if x%2 != 0]
print("Número ímpares são: {}".format(fibonacci_impares))
```

```
Número ímpares são: [1, 1, 3, 5, 13, 21, 55, 89, 233, 377, 987]
```

Olha quanto poder com uma linha de código! 🎉

Casos de Uso do Mundo Real

Vamos ver onde generators brilham na prática:

1. Processar Arquivos Grandes

```
# ❌ RUIM: Carrega arquivo inteiro na memória (pode estourar RAM!)
def ler_arquivo_ruim(filename):
    with open(filename) as f:
        linhas = f.readlines() # Carrega TUDO
    return [linha.strip().upper() for linha in linhas]

# ✅ BOM: Processa linha por linha (memória constante)
def ler_arquivo_bom(filename):
    with open(filename) as f:
        for linha in f: # Generator implícito
            yield linha.strip().upper()

# Usar
for linha in ler_arquivo_bom('gigante.log'):
    processar(linha) # Processa uma de cada vez
```

2. Pipeline de Processamento de Dados

```
def ler_numeros(filename):
    """Generator: lê números de arquivo"""
    with open(filename) as f:
        for linha in f:
            yield int(linha.strip())

def filtrar_pares(numeros):
    """Generator: filtra apenas pares"""
    for num in numeros:
        if num % 2 == 0:
            yield num

def elevar_quadrado(numeros):
    """Generator: eleva ao quadrado"""
    for num in numeros:
        yield num ** 2

# Compor pipeline (lazy!)
numeros = ler_numeros('dados.txt')
pares = filtrar_pares(numeros)
quadrados = elevar_quadrado(pares)

# Processar (executa tudo de uma vez, mas economizando memória)
for resultado in quadrados:
    print(resultado)
```

3. Stream Infinito (Fibonacci Infinito)

```
def fibonacci_infinito():
    """Generator infinito - nunca para!"""
    a, b = 0, 1
    while True: # Loop infinito
        yield a
        a, b = b, a + b

# Pegar apenas os primeiros 10
from itertools import islice
fib_10 = list(islice(fibonacci_infinito(), 10))
print(fib_10) # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Quando Usar Cada Abordagem?

Use Generator quando:

- ✓ **Dataset grande ou infinito** - Arquivos de GB/TB - Streams de dados - Sequências matemáticas infinitas
- ✓ **Iterar apenas uma vez** - Processamento em pipeline - ETL (Extract, Transform, Load)
- ✓ **Memória limitada** - Sistemas embarcados - Servidores com múltiplos processos

Use Lista quando:

- ✓ **Precisa acessar múltiplas vezes** - Cache de resultados - Reutilizar dados
- ✓ **Precisa de indexação** - `lista[5], lista[-1]` - Random access

 **Precisa de métodos de lista** - `len()`, `sort()`, `reverse()` - Modificação in-place

 **Dataset pequeno** - < 10.000 itens - Memória não é problema

Use Iterator (classe) quando:

 **Lógica complexa de iteração** - Múltiplos estados internos - Métodos auxiliares necessários

 **Reutilização com reset** - Implementar método `reset()` customizado

 **Regra de Ouro:** Se você vai iterar **apenas uma vez** e o dataset é **grande**, use **generator**. Caso contrário, lista é mais simples.

Conclusão

Neste guia completo sobre **Iterators e Generators**, você aprendeu:

-  **Protocolo Iterator** - `__iter__` e `__next__`
-  **Generators com yield** - Sintaxe simplificada
-  **Lazy evaluation** - Processamento sob demanda
-  **Economia de memória** - Até **81.000x menos memória!**
-  **Casos de uso reais** - Arquivos grandes, pipelines, streams
-  **Quando usar cada um** - Generator vs Lista vs Iterator

Principais lições: - Generators economizam **memória massivamente** (~100 bytes vs GB) - Use generators para **arquivos grandes e processamento em stream** - Use listas quando precisa **reutilizar** ou **indexar** - Generators são **consumidos apenas uma vez** - `yield` cria generator automaticamente, sem precisar de classe

Próximos passos: - Revise seus códigos com loops imensos carregando listas grandes - Experimente substituir por generators - Explore `itertools` para generators avançados - Combine generators para criar pipelines de dados

Agora... *Que tal dar uma revisitada nos seus códigos, focando naqueles loops imensos?*

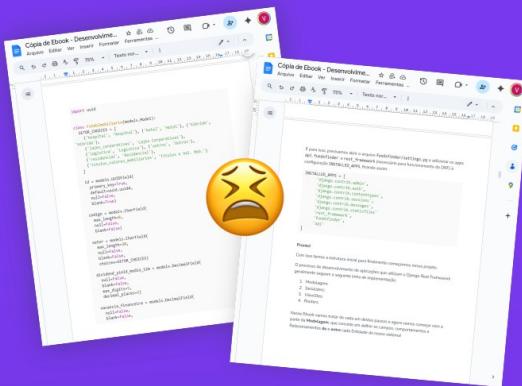
Talvez ali haja uma boa oportunidade de botar em prática a teoria que você leu aqui!

Bom desenvolvimento!



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



Arquitetura de Software Moderna

```
import python
import python

class Arquitetura_de_Software_Moderna:
    ...
    def share(self):
        pass
    ...
    return "Arquitetura de Mod", "arquitetura_mod"
}

def __init__(self):
    if user.username == self.username:
        self.username = self.username + self.username
        self.password = self.password + self.password
        self.name = self.name + self.name
    ...
    return self.username
}

resource saabell0
```

AI-generated system

A arquitetura com prolívia algoritmo software amadeirado de fusões modernas. Sesemtos tímicoscausus concretiza modulaçao estruturada externa. Chaveio e aonex dialektos AI-generated sistema si generated system oplemonia copiente enemot.

```
graph TD
    UserInput[User input] --> DataProcessor[Data processor]
    DataProcessor --> Agents[Agents]
    Agents --> Archestrator[Archestrator]
    Agents --> Cache[Cache]
    Agents --> Orchestrator[Orchestrator]
    SystemOutput[System output] --> DataProcessor
    Archestrator --> SystemOutput
```

Clean layout

Gentilmente Alia maticot en turbacit evicticos that alion ossibid to coenize Inugra with opegrath en oncees dibos. Net layout in gremarios formatacione exzma um dñivormour exzistem foa miltibid diginucleus, poiso ee dñor alour fumilat.



</> 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