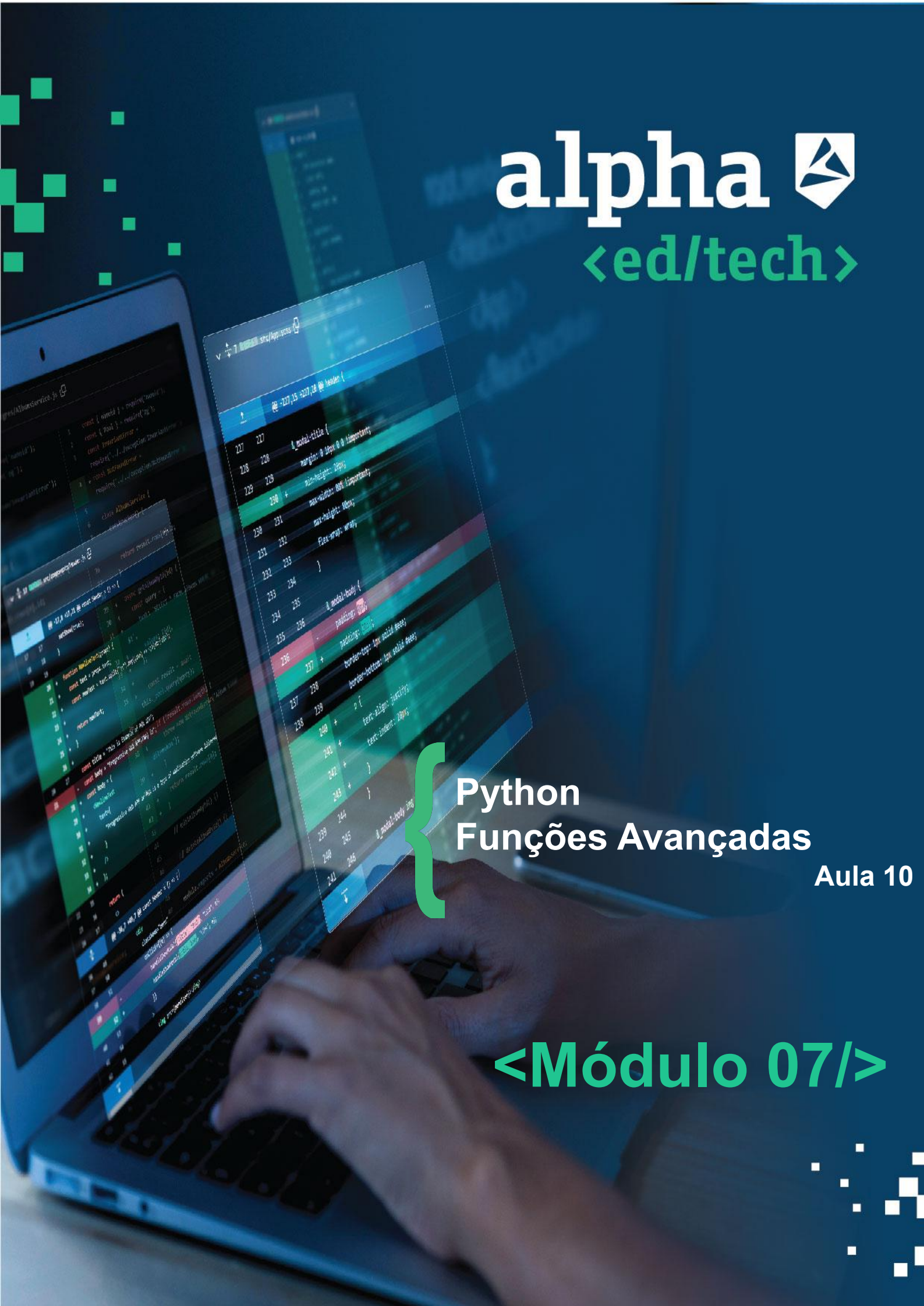

alpha
<ed/tech>
Python
Funções Avançadas
Aula 10
<Módulo 07/>

Funções Avançadas

Sumário

Funções Avançadas

Introdução

Nesta aula, vamos mergulhar em alguns dos recursos mais avançados e poderosos do Python, que elevam a programação a um novo patamar de eficiência, elegância e expressividade.

Começaremos nossa jornada explorando os **geradores**, uma ferramenta essencial para lidar com grandes volumes de dados ou sequências infinitas, permitindo processamento sob demanda e otimização de memória.

Construiremos **pipelines** com geradores, uma técnica poderosa para encadear operações de forma modular e eficiente, culminando em um exemplo prático de **processamento de arquivo de log**. Em seguida, desvendaremos o conceito de **funções de alta ordem (HOFs)**, funções que podem receber outras funções como argumentos ou retorná-las, abrindo as portas para um código mais flexível e reutilizável.

Investigaremos as aplicações práticas de HOFs, como `max()`, `sorted()`, `map()` e `filter()`, e entenderemos como elas simplificam tarefas comuns.

Abordaremos também as **funções lambda**, uma forma concisa de criar funções anônimas, perfeitas para operações simples e pontuais.

Por fim, mergulharemos no fascinante mundo dos **decorators**, que nos permitem modificar ou aprimorar funções de maneira elegante, adicionando funcionalidades extras ou simplificando a criação de classes.

Prepare-se para expandir seus horizontes e dominar técnicas avançadas que o tornarão um programador Python mais proficiente e versátil!

Geradores

Motivação

Imagine que você está trabalhando em um projeto de análise de dados onde precisa processar arquivos enormes, que não cabem na memória do seu computador de uma só vez.

Ou pense em uma situação onde você precisa realizar várias operações em uma sequência de dados, mas quer manter seu código organizado e fácil de entender.

As funções geradoras em Python são uma ferramenta poderosa para resolver esses tipos de problemas, permitindo que você trabalhe com grandes volumes de dados de forma eficiente e mantenha seu código modular e legível.

O que são geradores

Geradores são uma forma especial de iteradores em Python. Iteradores são objetos que permitem percorrer uma coleção de dados, elemento por elemento, sem precisar carregar todos os elementos na memória ao mesmo tempo.

Os geradores simplificam a criação de iteradores, permitindo que você defina uma função que "gera" valores sob demanda, utilizando a palavra-chave `yield`.

Funções Geradoras e a Função next()

Antes de mergulharmos em exemplos práticos e no uso de geradores em loops, vamos entender o conceito fundamental de uma função geradora e como podemos interagir com ela usando a função `next()`. Isso nos ajudará a compreender o mecanismo por trás dos geradores.

Geradores

Exemplo Didático: Sequência de Números

Vamos começar com o código de uma função geradora simples e, em seguida, explicaremos seu funcionamento passo a passo.

Uma função geradora é qualquer função que usa a palavra-chave `yield` em vez de `return`:

```python
# É uma função geradora porque usa "yield"
def gerador_simples(maximo):
    print("Início da função geradora")
    for n in range(maximo):
        print(f"Antes do yield: {n}")
        yield n
        print(f"Depois do yield: {n}")
    print("Fim da função geradora")

# Chamando a função geradora
gerador = gerador_simples(3)
print("Função geradora chamada, objeto gerador criado")
print(f"Tipo do objeto gerador: {type(gerador)}")
```

Ao executar esse código, a seguinte saída será produzida:

```
Função geradora chamada, objeto gerador criado
Tipo do objeto gerador: <class 'generator'>
```

Explicação:

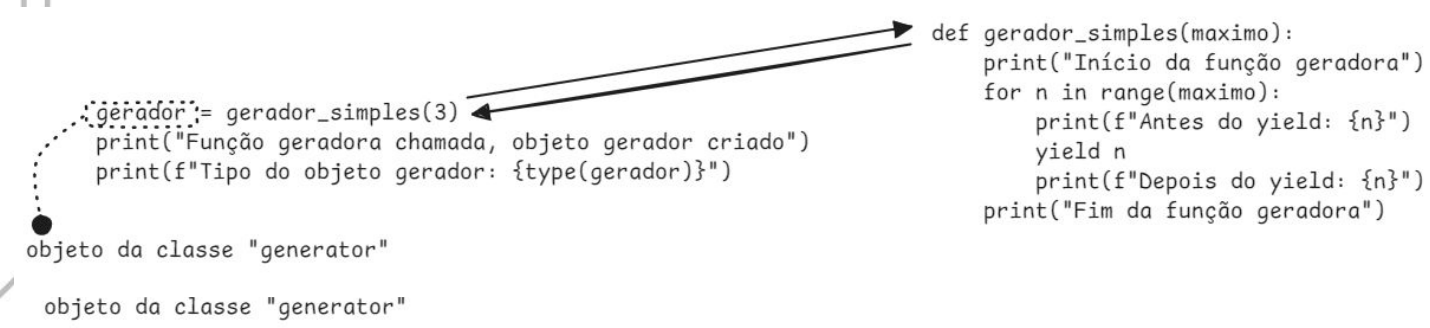
1. **Chamada da Função:**

   Quando chamamos `gerador = gerador_simples(3)`, a função `gerador_simples` **não é executada imediatamente**. Em vez disso, um objeto gerador é criado e atribuído à variável `gerador`.

   Como a função não é executada, na Saída não apareceu "Início da função geradora"

2. **Tipo do Gerador:**

   A linha `print(f"Tipo do objeto gerador: {type(gerador)}")` confirma que a variável `gerador` é um objeto do tipo `<class 'generator'>`.



Geradores

A Função next()

A função `next()` é usada para solicitar o próximo valor de um gerador. Vamos ver o que acontece quando aplicamos `next()` ao nosso objeto gerador:

```
gerador = gerador_simples(3)
print("Função geradora chamada, objeto gerador criado")
print(f"Tipo do objeto gerador: {type(gerador)}")

# Obtendo o próximo valor do gerador
valor = next(gerador)
print(f"Valor recebido com next(): {valor}")
```

Saída:

```
Função geradora chamada, objeto gerador criado
Tipo do objeto gerador: <class 'generator'>
Início da função geradora
Antes do yield: 0
Valor recebido com next(): 0
```

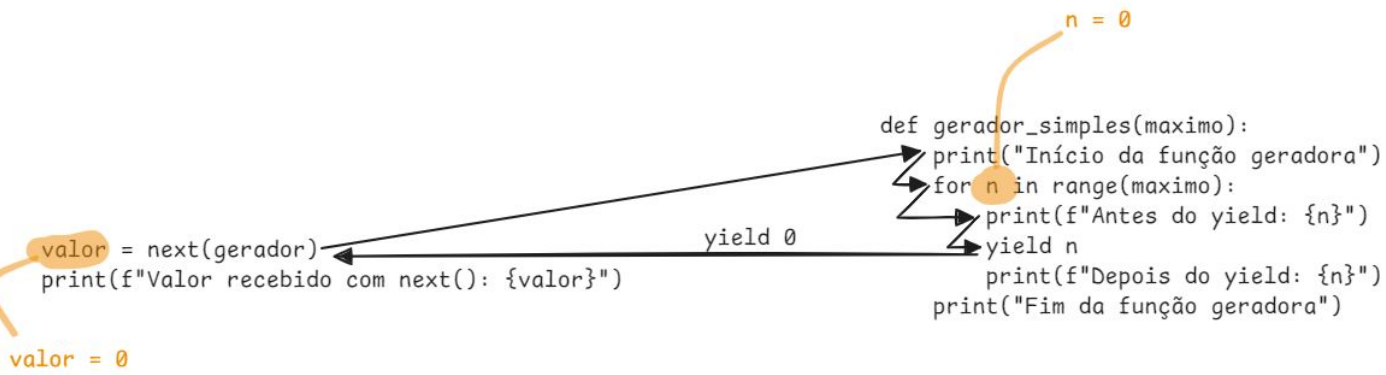
Explicação:

1. **Primeira Chamada de `next()`:** Quando chamamos `next(gerador)` pela primeira vez, a função `gerador_simples` começa a ser executada a partir do início.

2. **Execução até o `yield`:** A função executa `print("Início da função geradora")`, entra no loop `for n in range(maximo):` com `n=0`, executa `print(f"Antes do yield: {n}")` e então encontra a instrução `yield n`.

3. **Retorno e Pausa:** O `yield` retorna o valor atual de `n` (que é 0) e pausa a execução da função. O estado da função (as variáveis locais da função, incluindo o valor de `n`) é salvo.

4. **Valor Recebido:** A variável `valor` recebe o valor retornado pelo `yield` (0), e `print(f"Valor recebido com next(): {valor}")` exibe esse valor.



Geradores

Chamadas Subsequentes de `next()`:

Vamos chamar `next(gerador)` mais algumas vezes e ver o que acontece:

```
valor = next(gerador)
print(f"Valor recebido com next(): {valor}")

valor = next(gerador)
print(f"Valor recebido com next(): {valor}")
```

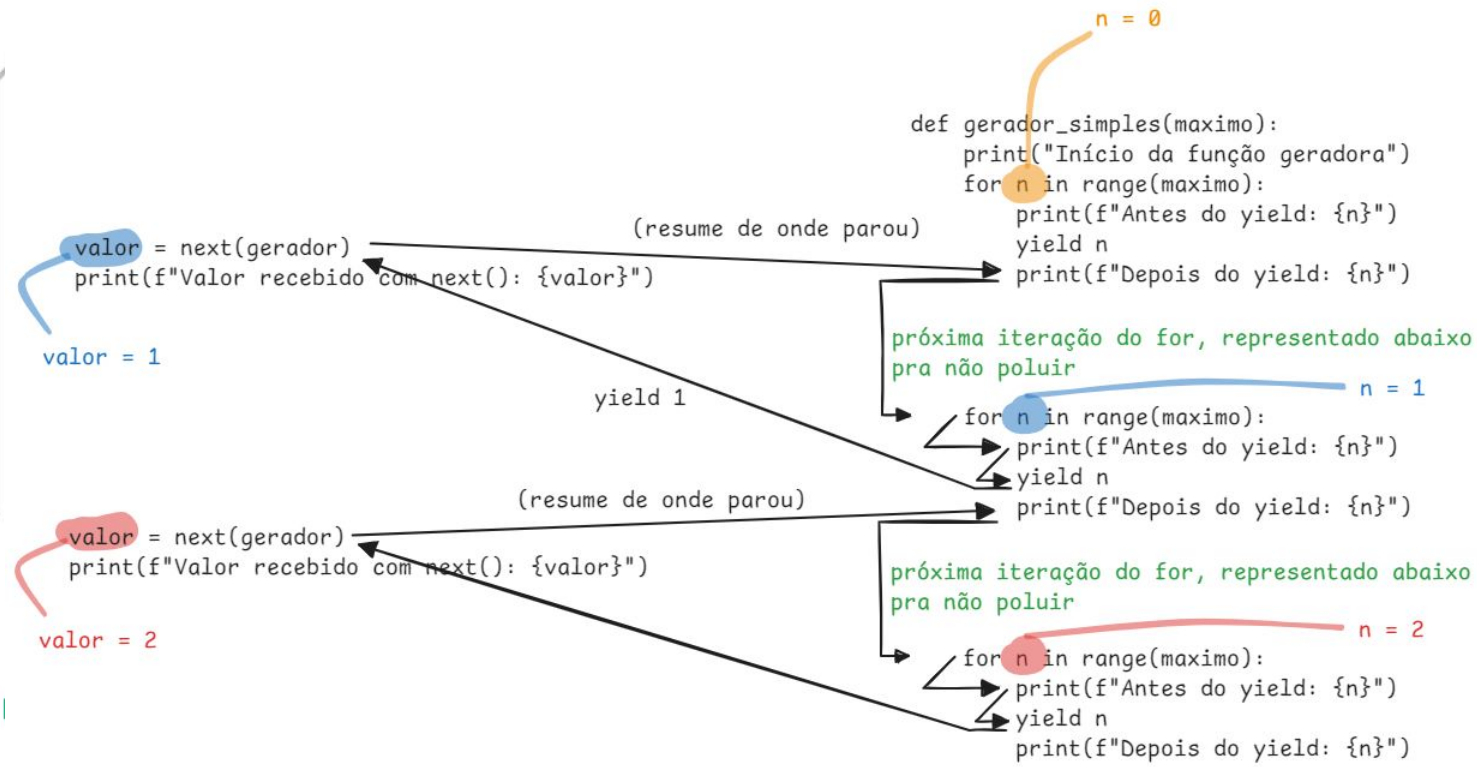
Saída:

```
Depois do yield: 0
Antes do yield: 1
Valor recebido com next(): 1
Depois do yield: 1
Antes do yield: 2
Valor recebido com next(): 2
```

Explicação:

1. **Segunda Chamada de `next()`:** A função `gerador_simples` continua de onde parou (após o `yield` anterior). Executa `print(f"Depois do yield: {n}")` (com `n` ainda sendo 0), incrementa `n` para 1, executa `print(f"Antes do yield: {1}")`, e encontra `yield n` novamente. O valor 1 é retornado e a função pausa.

2. **Terceira Chamada de `next()`:** O processo se repete. A função continua, executa `print(f"Depois do yield: {n}")` (com `n` agora sendo 1), incrementa `n` para 2, executa `print(f"Antes do yield: {2}")`, e retorna 2 através do `yield`.



Geradores

Esgotando o Gerador:

O que acontece se chamarmos `next(gerador)` mais uma vez?

```
valor = next(gerador)
print(f"Valor recebido com next(): {valor}")
```

Saída:

```
Depois do yield: 2
Fim da função geradora
Traceback (most recent call last):
  File "main.py", line 16, in <module>
    valor = next(gerador)
StopIteration
```

Explicação:

1. **Última Execução:** A função `gerador_simples` continua de onde parou (após o `yield` que retornou 2). Executa `print(f"Depois do yield: {n}")` (com `n` sendo 2), e o loop `for` termina.

2. **Fim da Função:** A função executa `print("Fim da função geradora")` e chega ao fim.

3. `StopIteration`**:** Quando um gerador chega ao fim e não tem mais valores para retornar, ele levanta uma exceção `StopIteration`. Isso sinaliza que o gerador foi exaurido.

Geradores

Resumo da Função next()

- A função `next(gerador)` solicita o próximo valor do gerador.
- Na primeira chamada, a função geradora é executada até encontrar o primeiro `yield`.
- O `yield` retorna um valor e pausa a execução da função, salvando seu estado interno (suas variáveis locais).
- Chamadas subsequentes de `next()` continuam a execução de onde a função parou.
- Quando o gerador termina, ele levanta a exceção `StopIteration`.

Geradores e Loops for

Agora que entendemos como a função `next()` funciona, podemos apreciar melhor como os geradores funcionam em conjunto com os loops `for`.

Na verdade, um loop `for` utiliza a função `next()` internamente para iterar sobre um gerador.

Veja o exemplo abaixo, que é equivalente ao que fizemos com chamadas explícitas a `next()`:

```
gerador = gerador_simples(3)  # Recriando o gerador

for numero in gerador:
    print(f"Número recebido no loop: {numero}")
```

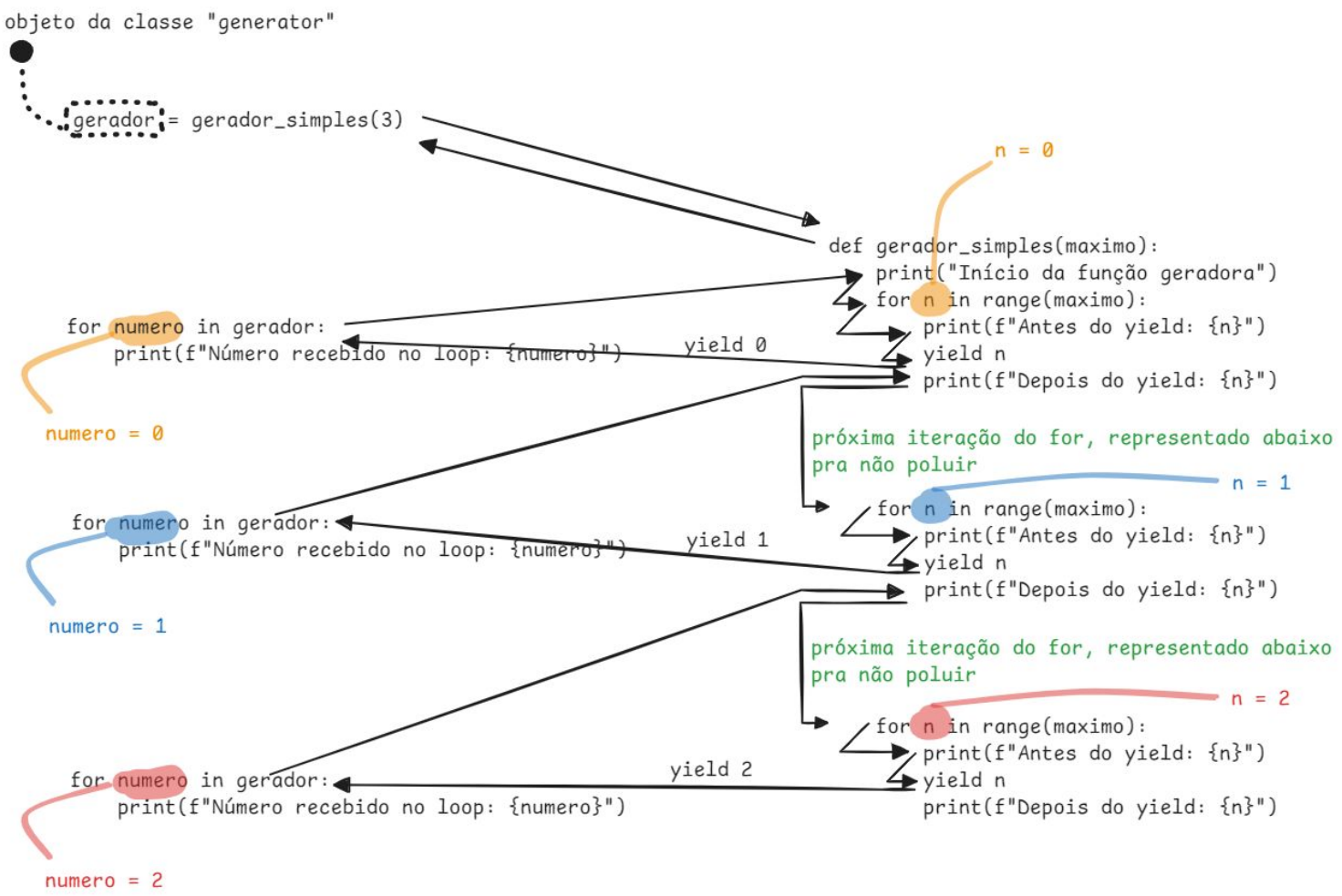
Saída:

```
Início da função geradora
Antes do yield: 0
Número recebido no loop: 0
Depois do yield: 0
Antes do yield: 1
Número recebido no loop: 1
Depois do yield: 1
Antes do yield: 2
Número recebido no loop: 2
Depois do yield: 2
Fim da função geradora
```

Explicação:

- O loop `for` chama `next(gerador)` automaticamente a cada iteração para obter o próximo valor.
- Quando `next()` retorna um valor, esse valor é atribuído à variável `numero`.
- O loop `for` continua até que a exceção `StopIteration` seja levantada, indicando o fim do gerador.

Geradores


objeto da classe "generator"
gerador = gerador_simples(3)
n = 0
def gerador_simples(maximo):
print("Início da função geradora")
for n in range(maximo):
print(f"Antes do yield: {n}")
yield n
print(f"Depois do yield: {n}")
for numero in gerador:
print(f"Número recebido no loop: {numero}")
yield 0
numero = 0
próxima iteração do for, representado abaixo
pra não poluir
n = 1
for n in range(maximo):
print(f"Antes do yield: {n}")
yield n
print(f"Depois do yield: {n}")
for numero in gerador:
print(f"Número recebido no loop: {numero}")
yield 1
numero = 1
próxima iteração do for, representado abaixo
pra não poluir
n = 2
for n in range(maximo):
print(f"Antes do yield: {n}")
yield n
print(f"Depois do yield: {n}")
for numero in gerador:
print(f"Número recebido no loop: {numero}")
yield 2
numero = 2

Geradores

Comparação com uma Função Normal

Uma função normal que retorna uma lista com a mesma sequência faria o seguinte:

```
def funcao_normal(maximo):
    print("Início da função normal")
    numeros = []
    for n in range(maximo):
        numeros.append(n)
    print("Fim da função normal")
    return numeros

lista = funcao_normal(3)
print("Função normal chamada, lista criada")
for numero in lista:
    print(f"Número da lista: {numero}")
```

Saída:

```
Início da função normal
Fim da função normal
Função normal chamada, lista criada
Número da lista: 0
Número da lista: 1
Número da lista: 2
```

Diferenças Fundamentais:

- **Execução Imediata vs. Sob Demanda:** A função normal executa todo o seu código de uma vez, criando uma lista com todos os números da sequência. A função geradora, por outro lado, executa "em pedaços", gerando um número por vez, apenas quando solicitado por `next()` (ou pelo loop `for`).
- **Retorno Único vs. Múltiplos Retornos:** A função normal retorna uma única vez, entregando a lista completa. A função geradora retorna várias vezes (através do `yield`), entregando um valor a cada vez que `next()` é chamado.
- **Uso de Memória:** A função normal precisa armazenar todos os números da sequência na memória, o que pode ser problemático para sequências muito grandes. A função geradora armazena apenas o estado atual da função, o que a torna muito mais eficiente em termos de uso de memória.

Resumo

Neste exemplo, vimos que as funções geradoras são uma forma especial de função em Python que permite gerar valores sob demanda, utilizando a palavra-chave `yield`.

Aprendemos também sobre a função `next()`, que é usada para obter o próximo valor de um gerador.

Além disso, entendemos que o loop `for` utiliza `next()` internamente para iterar sobre geradores.

Essas características tornam as funções geradoras ideais para trabalhar com grandes conjuntos de dados ou sequências infinitas, pois economizam memória e permitem processamento sob demanda.

Pipeline com geradores

Uma das aplicações mais interessantes das funções geradoras é a construção de pipelines de processamento de dados.

Um pipeline é uma sequência de operações encadeadas, onde a saída de uma operação é a entrada da próxima.

Geradores são ideais para isso porque permitem que cada etapa do pipeline processe os dados sob demanda, sem precisar armazenar resultados intermediários na memória.

Nesta seção, vamos construir um pipeline simples com duas funções geradoras para demonstrar esse conceito.

Exemplo Didático de Pipeline

Vamos criar duas funções geradoras:

- `gerar_numeros(maximo)`: Gera uma sequência de números de 0 até `maximo` - 1.
- `dobrar_numeros(numeros)`: Recebe uma sequência de números e retorna cada número multiplicado por 2.

Aqui está o código:

```
def gerar_numeros(maximo):
    print("Início do gerador de números")
    for i in range(maximo):
        print(f"Gerando número: {i}")
        yield i
    print("Fim do gerador de números")

def dobrar_numeros(numeros):
    print("Início do dobrador de números")
    for numero in numeros:
        print(f"Dobrando número: {numero}")
        yield numero * 2
    print("Fim do dobrador de números")

# Criando o pipeline
numeros = gerar_numeros(4)
numeros_dobrados = dobrar_numeros(numeros)

# Consumindo o pipeline
print("Iniciando o consumo do pipeline")
for num_dobrado in numeros_dobrados:
    print(f"Número dobrado encontrado: {num_dobrado}")
```

Pipeline com geradores

Ao executar o código, a seguinte saída será produzida:

```
Iniciando o consumo do pipeline
Início do dobrador de números
Início do gerador de números
Gerando número: 0
Dobrando número: 0
Número dobrado encontrado: 0
Gerando número: 1
Dobrando número: 1
Número dobrado encontrado: 2
Gerando número: 2
Dobrando número: 2
Número dobrado encontrado: 4
Gerando número: 3
Dobrando número: 3
Número dobrado encontrado: 6
Fim do gerador de números
Fim do dobrador de números
```

Pipeline com geradores

Explicação Detalhada

1. **Criação dos Geradores:**
   - `numeros = gerar_numeros(4)` cria um objeto gerador que, quando iterado, produzirá números de 0 a 3.
   - `numeros_dobrados = dobrar_numeros(numeros)` cria outro objeto gerador que, quando iterado, receberá números do gerador `numeros` e os retornará multiplicados por 2.
   - Importante: Nenhuma das funções geradoras (`gerar_numeros` e `dobrar_numeros`) é executada neste momento. Elas apenas retornam objetos geradores.
2. **Início do Processamento:**
   - `print("Iniciando o consumo do pipeline")` marca o início do processamento.
3. **Loop Externo:**
   - `for num_dobrado in numeros_dobrados:` inicia o loop que consome os valores gerados pelo pipeline. Internamente, o loop `for` chama a função `next()` em `numeros_dobrados` a cada iteração.
4. **Execução em Cadeia (Lazy Evaluation):**
   - Para obter o primeiro valor, `numeros_dobrados` precisa solicitar um valor a `numeros` (chamando `next(numeros)` internamente).
   - `gerar_numeros` é então executado até o primeiro `yield i` (quando `i` é 0), retornando 0 e pausando.
   - `dobrar_numeros` recebe esse valor 0, multiplica por 2 e, através do `yield numero * 2`, retorna 0, pausando também.
   - O loop externo recebe esse valor 0 e o imprime.
5. **Continuação do Processamento:**
   - O loop externo solicita o próximo valor (chamando `next(numeros_dobrados)`).
   - `dobrar_numeros` continua de onde parou, mas percebe que precisa de mais um valor de `numeros` (chamando `next(numeros)` internamente).
   - `gerar_numeros` continua, gera o próximo valor (1), e o ciclo se repete.
   - Essa "dança" entre os geradores continua até que `gerar_numeros` chegue ao fim (após gerar 3).
6. **Fim dos Geradores:**
   - Quando `gerar_numeros` termina, `dobrar_numeros` também termina, e o loop externo é finalizado.

Pipeline com geradores

Benefícios do Pipeline com Geradores

- **Eficiência de Memória:** Os números são gerados e processados um por vez. Não há necessidade de armazenar todos os números ou os resultados intermediários em memória.
- **Modularidade:** Cada função geradora tem uma responsabilidade clara e pode ser facilmente substituída ou modificada.
- **Lazy Evaluation:** O processamento só ocorre quando necessário, ou seja, quando o loop externo solicita um valor.
- **Flexibilidade:** Podemos adicionar mais etapas ao pipeline facilmente, criando processamentos mais complexos.

Resumo

Neste exemplo, vimos como as funções geradoras podem ser encadeadas para formar um pipeline de processamento de dados.

Cada gerador no pipeline atua como uma etapa de processamento, recebendo valores do gerador anterior e produzindo novos valores para o próximo.

Essa abordagem é eficiente em termos de memória, modular e flexível, tornando-a ideal para processar grandes volumes de dados ou para construir processamentos complexos passo a passo.

Aplicação prática de geradores em pipeline: Processamento de um Arquivo de Log

Vamos agora aplicar o conceito de geradores e pipelines a um problema mais próximo de situações reais: o processamento de um arquivo de log.

Arquivos de log são comumente gerados por aplicações para registrar eventos, erros, e outras informações relevantes.

Processá-los de forma eficiente é uma tarefa importante para administradores de sistemas e desenvolvedores.

Imagine que temos um arquivo de log com o seguinte formato:

```
2023-10-27 10:00:00 - INFO - Usuário X logou no sistema
2023-10-27 10:05:00 - ERRO - Falha ao processar pagamento
2023-10-27 10:10:00 - INFO - Usuário Y fez uma compra
2023-10-27 10:15:00 - ERRO - Banco de dados indisponível
2023-10-27 10:20:00 - INFO - Usuário Z alterou a senha
```

Nosso objetivo é extrair a data, hora e mensagem das linhas que contêm erros (linhas que começam com "ERRO").

Aplicação prática de geradores em pipeline: Processamento de um Arquivo de Log

Implementação com Geradores

Vamos construir um pipeline com três funções geradoras para processar o arquivo de log:

1. `ler_linhas(nome_arquivo)`: Lê o arquivo linha por linha, retornando cada linha.
2. `filtrar_erros(linhas)`: Recebe uma sequência de linhas e retorna apenas aquelas que contêm "ERRO".
3. `extrair_informacoes(linhas)`: Recebe uma sequência de linhas de erro e extrai a data, hora e mensagem, retornando-as como uma tupla.

Aqui está o código:

```
def ler_linhas(nome_arquivo):
    print(f"Abrindo arquivo: {nome_arquivo}")
    with open(nome_arquivo, 'r') as arquivo:
        for linha in arquivo:
            print("Lendo linha...")
            yield linha.strip()
    print("Fechando arquivo.")

def filtrar_erros(linhas):
    print("Iniciando filtro de erros...")
    for linha in linhas:
        print("Verificando linha para erros...")
        if "ERRO" in linha:
            print("Erro encontrado!")
            yield linha
    print("Filtro de erros concluído.")

def extrair_informacoes(linhas):
    print("Iniciando extração de informações...")
    for linha in linhas:
        partes = linha.split(' - ')
        if len(partes) >= 3:
            data_hora = partes[0] + ' ' + partes[1]
            mensagem = partes[2]
            print(f"Extraindo informações: {data_hora}, {mensagem}")
            yield (data_hora, mensagem)
    print("Extração de informações concluída.")

# Construindo o pipeline
linhas = ler_linhas('arquivo_de_log.txt')
linhas_com_erros = filtrar_erros(linhas)
informacoes = extrair_informacoes(linhas_com_erros)

print("Iniciando o processamento do log...")
for data_hora, mensagem in informacoes:
    print(f"Processando erro: Data e Hora: {data_hora}, Mensagem: {mensagem}")
print("Processamento do log concluído.")
```

Aplicação prática de geradores em pipeline: Processamento de um Arquivo de Log

Saída:

```
Iniciando o processamento do log...
Iniciando extração de informações...
Iniciando filtro de erros...
Abrindo arquivo: arquivo_de_log.txt
Lendo linha...
Verificando linha para erros...
Lendo linha...
Verificando linha para erros...
Erro encontrado!
Extraindo informações: 2023-10-27 10:05:00 ERRO, Falha ao processar pagamento
Processando erro: Data e Hora: 2023-10-27 10:05:00 ERRO, Mensagem: Falha ao processar
pagamento
Lendo linha...
Verificando linha para erros...
Lendo linha...
Verificando linha para erros...
Erro encontrado!
Extraindo informações: 2023-10-27 10:15:00 ERRO, Banco de dados indisponível
Processando erro: Data e Hora: 2023-10-27 10:15:00 ERRO, Mensagem: Banco de dados
indisponível
Lendo linha...
Verificando linha para erros...
Fechando arquivo.
Filtro de erros concluído.
Extração de informações concluída.
Processamento do log concluído.
```

Aplicação prática de geradores em pipeline: Processamento de um Arquivo de Log

Explicação Detalhada

1. `ler_linhas(nome_arquivo)`:
   - Abre o arquivo especificado em modo de leitura (`'r'`).
   - Itera sobre cada linha do arquivo usando um loop `for`.
   - `yield linha.strip()`: Retorna a linha atual, removendo espaços em branco no início e no final, e pausa a execução.
   - Quando o gerador é chamado novamente (por meio de `next()` ou de um loop `for`), a execução continua de onde parou (próxima linha do arquivo).
   - `print("Fechando arquivo.")`: É executado quando todas as linhas são lidas.
2. `filtrar_erros(linhas)`:
   - Recebe um iterável `linhas` (que pode ser o gerador `ler_linhas` ou qualquer outra sequência de linhas).
   - Itera sobre cada `linha` em `linhas`.
   - `if "ERRO" in linha:`: Verifica se a linha contém a string "ERRO".
   - `yield linha`: Se a linha contém "ERRO", ela é retornada e a execução pausa.
   - `print("Filtro de erros concluído.")`: É executado quando todas as linhas foram verificadas.
3. `extrair_informacoes(linhas)`:
   - Recebe um iterável `linhas` (que pode ser o gerador `filtrar_erros` ou qualquer outra sequência de linhas de erro).
   - `partes = linha.split(' - ')`: Divide a linha em partes usando " - " como separador.
   - `if len(partes) >= 3:`: Verifica se a linha tem pelo menos três partes (data, hora, nível, mensagem).
   - `data_hora = partes[0] + ' ' + partes[1]`: Extrai a data e a hora.
   - `mensagem = partes[2]`: Extrai a mensagem.
   - `yield (data_hora, mensagem)`: Retorna uma tupla contendo a data, hora e mensagem, e pausa a execução.
   - `print("Extração de informações concluída.")`: É executado quando todas as linhas foram processadas.
4. **Construindo o Pipeline:**
   - `linhas = ler_linhas('arquivo_de_log.txt')`: Cria um gerador para ler as linhas do arquivo.
   - `linhas_com_erros = filtrar_erros(linhas)`: Cria um gerador que filtra as linhas, retornando apenas as que contêm erros. Ele recebe o gerador `linhas` como entrada.
   - `informacoes = extrair_informacoes(linhas_com_erros)`: Cria um gerador que extrai as informações das linhas de erro. Ele recebe o gerador `linhas_com_erros` como entrada.
5. **Consumindo o Pipeline:**
   - `print("Iniciando o processamento do log...")`: Indica o início do processamento.
   - `for data_hora, mensagem in informacoes:`: Itera sobre o gerador `informacoes`. A cada iteração, o pipeline inteiro é executado passo a passo:
     - `ler_linhas` lê uma linha do arquivo (se ainda houver).
     - `filtrar_erros` verifica se a linha é um erro.
     - `extrair_informacoes` extrai as informações da linha de erro.
     - A tupla `(data_hora, mensagem)` é retornada para o loop `for`.
     - `print(f"Processando erro: Data e Hora: {data_hora}, Mensagem: {mensagem}")`: Exibe as informações extraídas.
   - `print("Processamento do log concluído.")`: É executado após o loop `for` ser finalizado.

Aplicação prática de geradores em pipeline: Processamento de um Arquivo de Log

Benefícios da Abordagem com Geradores

- **Eficiência de Memória:** O arquivo é processado linha por linha. Apenas as linhas de erro e as informações extraídas são mantidas na memória. Isso é crucial para arquivos de log grandes que não cabem na memória.
- **Modularidade:** Cada função tem uma responsabilidade bem definida: ler linhas, filtrar erros e extrair informações. Isso torna o código mais fácil de entender, manter e reutilizar.
- **Flexibilidade:** Podemos facilmente modificar o pipeline, adicionando ou removendo etapas, para adaptá-lo a diferentes formatos de log ou requisitos de processamento.
- **Lazy Evaluation:** O processamento é feito sob demanda. As linhas são lidas e processadas apenas quando o loop `for` solicita o próximo valor (através da chamada interna a `next()`). Isso melhora o desempenho, especialmente se apenas uma parte do arquivo precisar ser processada.

Resumo

Neste exemplo, aplicamos o conceito de geradores e pipelines para processar um arquivo de log de forma eficiente.

Construímos um pipeline composto por três funções geradoras, cada uma responsável por uma etapa específica do processamento: ler linhas, filtrar erros e extrair informações.

Essa abordagem demonstrou os benefícios do uso de geradores em termos de eficiência de memória, modularidade, flexibilidade e lazy evaluation.

Quando Usar Geradores

Geradores são especialmente úteis nas seguintes situações:

- **Trabalhando com Grandes Conjuntos de Dados:** Quando você precisa processar arquivos ou conjuntos de dados que não cabem na memória do computador, os geradores permitem que você carregue e processe os dados em pedaços, sob demanda.
- **Pipelines de Processamento:** Geradores são ideais para construir pipelines, onde a saída de uma etapa é a entrada da próxima. Isso torna o código mais modular, legível e fácil de manter.
- **Sequências Infinitas:** Geradores podem representar sequências infinitas de dados, pois geram valores apenas quando solicitados.
- **Avaliação Preguiçosa (Lazy Evaluation):** Se você precisa de um processamento sob demanda, onde os valores são gerados apenas quando necessários, os geradores são a escolha certa.

Funções de Alta Ordem (HOF)

O que são Funções de Alta Ordem?

Motivação:

Imagine que você está construindo um sistema modular onde certas operações podem variar, mas a estrutura geral permanece a mesma. Por exemplo, você pode querer aplicar diferentes operações a uma lista de números, como somar, multiplicar ou qualquer outra transformação. Como você faria isso de forma flexível e sem duplicar código? Funções de alta ordem oferecem uma solução elegante para esse tipo de problema.

Explicação do Conceito:

Uma função de alta ordem (Higher-Order Function, HOF) é uma função que pode receber uma ou mais funções como argumentos e/ou retornar uma função como resultado. Em Python, funções são tratadas como objetos de primeira classe, o que significa que elas podem ser atribuídas a variáveis, passadas como argumentos e retornadas por outras funções, assim como qualquer outro tipo de dado (como inteiros, strings ou listas).

Nesta seção inicial, vamos focar em exemplos didáticos, sem aplicações práticas imediatas, para ilustrar o mecanismo das funções de alta ordem. Posteriormente abordaremos aplicações práticas e como essas funções são comumente usadas em construções do Python.

Exemplo Didático 1: Passando uma função como parâmetro (sem retorno de valor)

Vamos começar com um exemplo simples para ilustrar como uma função pode ser passada como argumento para outra função.

```python
def funcao_a():
    print("Executando função A")

def funcao_b():
    print("Executando função B")

def funcao_de_alta_ordem(func):
    print("Início da função de alta ordem")
    func()
    print("Fim da função de alta ordem")

funcao_de_alta_ordem(funcao_a)
# Saída:
# Início da função de alta ordem
# Executando função A
# Fim da função de alta ordem

funcao_de_alta_ordem(funcao_b)
# Saída:
# Início da função de alta ordem
# Executando função B
# Fim da função de alta ordem
```

Funções de Alta Ordem (HOF)

Explicação:

- Definimos duas funções simples, `funcao_a` e `funcao_b`, que apenas imprimem uma mensagem.
- Definimos uma `funcao_de_alta_ordem` que recebe uma função `func` como parâmetro.
    - **O ponto crucial aqui é entender que `func` não é uma variável comum, mas sim uma referência para uma função.**

      Quando passamos `funcao_a` ou `funcao_b` como argumento, estamos essencialmente dizendo para a `funcao_de_alta_ordem`: "Use o código dentro de `funcao_a` (ou `funcao_b`) onde quer que você veja `func()`."
- Dentro da `funcao_de_alta_ordem`, chamamos a função `func()`. Isso significa que, dependendo da função que passamos como argumento, o código dentro de `funcao_a` ou `funcao_b` será executado.

  É como se estivéssemos inserindo o código de `funcao_a` ou `funcao_b` dentro da `funcao_de_alta_ordem` no local onde `func()` é chamado.
- Chamamos `funcao_de_alta_ordem` passando `funcao_a` e depois `funcao_b` como argumentos, demonstrando que o comportamento interno da HOF muda com base na função passada.
- Observe que, ao passar `funcao_a` ou `funcao_b` como argumento, **não usamos parênteses após o nome da função**. Isso é importante porque estamos passando a *função em si*, e não o *resultado* da chamada da função.

Funções de Alta Ordem (HOF)

Exemplo Didático 2: Passando uma função como parâmetro (com retorno de valor)

Agora que já vimos como passar uma função como parâmetro e como a função de alta ordem executa o código dessa função passada, vamos dar um passo adiante.

No exemplo anterior, as funções que passamos como parâmetro (`funcao_a` e `funcao_b`) não retornavam nenhum valor.

Neste exemplo, vamos explorar o que acontece quando a função passada como parâmetro **retorna um valor** e como a função de alta ordem pode **manipular esse valor de retorno**

```
def funcao_c():
    print("Executando função C")
    return "Resultado da função C"

def funcao_d():
    print("Executando função D")
    return "Resultado da função D"

def funcao_de_alta_ordem_retorno(func):
    print("Início da função de alta ordem com retorno")
    resultado = func()
    print(f"O resultado da função passada é: {resultado}")
    print("Fim da função de alta ordem com retorno")

funcao_de_alta_ordem_retorno(funcao_c)
# Saída:
# Início da função de alta ordem com retorno
# Executando função C
# O resultado da função passada é: Resultado da função C
# Fim da função de alta ordem com retorno

funcao_de_alta_ordem_retorno(funcao_d)
# Saída:
# Início da função de alta ordem com retorno
# Executando função D
# O resultado da função passada é: Resultado da função D
# Fim da função de alta ordem com retorno
```

Explicação:

- `funcao_c` e `funcao_d` agora retornam uma string, além de imprimir uma mensagem.
- `funcao_de_alta_ordem_retorno` chama a função passada como parâmetro e armazena o resultado em uma variável `resultado`.
    - Novamente, `func()` é substituído pelo código da função que passamos como argumento. Se passarmos `funcao_c`, `resultado` receberá o valor retornado por `funcao_c`.
- O resultado é então impresso, mostrando como a HOF pode lidar com funções que retornam valores.

Funções de Alta Ordem (HOF)

Exemplo Didático 3: Funções com parâmetros

Nos exemplos anteriores, as funções que passamos como parâmetro não recebiam nenhum argumento.

Neste exemplo, vamos explorar o cenário onde a função passada como parâmetro também precisa receber um ou mais argumentos.

Veremos como a função de alta ordem pode passar esses argumentos para a função que ela está executando, adicionando uma camada extra de flexibilidade ao nosso código.

```
def funcao_e(x):
    print(f"Executando função E com x = {x}")
    return x * 2

def funcao_f(y):
    print(f"Executando função F com y = {y}")
    return y + 5

def funcao_de_alta_ordem_parametros(func):
    print("Início da função de alta ordem com parâmetros")
    resultado1 = func(10)
    print(f"Resultado da primeira chamada: {resultado1}")
    resultado2 = func(20)
    print(f"Resultado da segunda chamada: {resultado2}")
    print("Fim da função de alta ordem com parâmetros")

funcao_de_alta_ordem_parametros(funcao_e)
# Saída:
# Início da função de alta ordem com parâmetros
# Executando função E com x = 10
# Resultado da primeira chamada: 20
# Executando função E com x = 20
# Resultado da segunda chamada: 40
# Fim da função de alta ordem com parâmetros

funcao_de_alta_ordem_parametros(funcao_f)
# Saída:
# Início da função de alta ordem com parâmetros
# Executando função F com y = 10
# # Resultado da primeira chamada: 15
# Executando função F com y = 20
# Resultado da segunda chamada: 25
# Fim da função de alta ordem com parâmetros
```

Explicação:

- `funcao_e` e `funcao_f` agora recebem um parâmetro e retornam um valor baseado nesse parâmetro.
- `funcao_de_alta_ordem_parametros` recebe uma função `func` como parâmetro.
- Dentro da HOF, chamamos `func` duas vezes, passando valores diferentes (10 e 20) como argumentos.

  Isso ilustra que a função de alta ordem pode chamar a função recebida como parâmetro passando outros parâmetros para ela. Os valores 10 e 20 são apenas exemplos; a HOF poderia passar qualquer valor, dependendo da lógica desejada.
- Este exemplo demonstra como uma HOF pode passar argumentos para a função que ela recebe, permitindo ainda mais flexibilidade.

Funções de Alta Ordem (HOF)

Resumo

Nesta seção, introduzimos o conceito de funções de alta ordem e demonstramos, através de exemplos sem aplicação prática imediata, como elas funcionam.

Vimos como passar funções como parâmetros, como lidar com funções que retornam valores e como passar argumentos para essas funções dentro da HOF.

Agora, vamos explorar algumas aplicações práticas desse conceito.

Aplicações Práticas de Funções de Alta Ordem

Nesta seção, vamos ver como as funções de alta ordem são usadas em construções comuns do Python. Nosso objetivo é mostrar as principais utilidades dessas funções na prática.

max()

Motivação:

Vamos começar com um problema simples: encontrar o maior número em uma lista. Você pode usar a função `max()` do Python para isso.

Mas e se a lista contiver objetos mais complexos, como dicionários? Como você encontraria o maior elemento com base em um critério específico, como o maior preço de um produto em uma lista de dicionários de produtos?

Explicação do Conceito:

- `max(iteravel, key=funcao)`: Retorna o maior item do `iteravel` com base no valor retornado pela `funcao` aplicada a cada item.
- `max` **chama a função passada em** `key` **várias vezes, uma vez para cada elemento do iterável. O elemento para o qual a função** `key` **retornar o maior valor será considerado o maior elemento.**

Exemplo 1: Usando max() com uma lista de números

```python
numeros = [-1, -5, 2, 3, -2]

maior_numero = max(numeros)
print(f"Maior número: {maior_numero}")
# Saída:
# Maior número: 3

def absoluto(x):
  return abs(x)

maior_absoluto = max(numeros, key=absoluto)
print(f"Maior valor absoluto: {maior_absoluto}")
# Saída:
# Maior valor absoluto: -5
```

Explicação:

- `max(numeros)` retorna o maior número da lista, que é 3.
- `max(numeros, key=absoluto)` retorna o número com o maior valor absoluto (absoluto é o valor sem sinal)
    - `max` chama `absoluto` para cada número na lista: `absoluto(-1)` retorna 1, `absoluto(-5)` retorna 5, `absoluto(2)` retorna 2, `absoluto(3)` retorna 3, `absoluto(-2)` retorna 2.
    - O maior valor retornado por `absoluto` é 5, que corresponde ao número -5. Portanto, -5 é retornado como o elemento que tem maior valor absoluto.

Aplicações Práticas de Funções de Alta Ordem

Exemplo 2: Usando max() com uma lista de dicionários

```
produtos = [
    {"nome": "Camiseta", "preco": 50, "estoque": 100},
    {"nome": "Calça", "preco": 100, "estoque": 50},
    {"nome": "Tênis", "preco": 150, "estoque": 20},
    {"nome": "Meia", "preco": 10, "estoque": 200}
]

def obter_preco(produto):
    return produto["preco"]

produto_mais_caro = max(produtos, key=obter_preco)
print(f"Produto mais caro: {produto_mais_caro}")
# Saída:
# Produto mais caro: {'nome': 'Tênis', 'preco': 150, 'estoque': 20}
```

Explicação:

- Temos uma lista de `produtos`, onde cada produto é um dicionário.
- Definimos uma função `obter_preco` que retorna o preço de um produto.
- `max(produtos, key=obter_preco)` encontra o produto mais caro.
  - `max` chama `obter_preco` para cada produto na lista. O valor retornado (o preço) é usado para comparar os produtos.
  - O produto com o maior preço retornado por `obter_preco` é considerado o produto mais caro.

Aplicações Práticas de Funções de Alta Ordem

Explicação do `meu_max()`:

1. **Inicialização:** Inicializamos `maior_elemento` e `maior_chave` como `None`.
2. **Iteração:** Iteramos sobre cada `elemento` no `iteravel`.
3. **Chave de Comparação:** Para cada `elemento`, chamamos a função `key` para obter a `chave` de comparação.
    - Aqui é onde a função de alta ordem entra em ação: `key(elemento)` executa a função que foi passada como argumento, permitindo que o usuário personalize o critério de comparação.
4. **Comparação:** Comparamos a `chave` atual com a `maior_chave` que vimos até agora. Se a `chave` atual for maior, atualizamos `maior_elemento` e `maior_chave`.
5. **Retorno:** Após iterar por todos os elementos, retornamos o `maior_elemento`.

Conexão com a Teoria de HOFs:

- `meu_max()` é uma função de alta ordem porque aceita uma função (`key`) como argumento.
- A função `key` é chamada dentro do loop, demonstrando como a HOF pode executar o código da função passada como parâmetro.
- O comportamento de `meu_max()` muda com base na função `key` fornecida, mostrando a flexibilidade das HOFs.

Aplicações Práticas de Funções de Alta Ordem

sorted()

Motivação:

Vamos explorar outra função de alta ordem comum: `sorted()`. Ela é usada para ordenar sequências, como listas. Mas como ela se comportaria com sequências de objetos mais complexos, como dicionários?

Explicação do Conceito:

A função `sorted()` em Python é uma função de alta ordem que pode receber uma função como argumento para o parâmetro `key`. Essa função `key` é aplicada a cada elemento da lista antes da ordenação, permitindo que você especifique um critério de ordenação personalizado.

Exemplo 1: Ordenando uma lista de números

```
precos = [10, 5, 20, 15, 25]

precos_ordenados = sorted(precos)
print(f"Preços ordenados em ordem crescente: {precos_ordenados}")
# Saída:
# Preços ordenados em ordem crescente: [5, 10, 15, 20, 25]

precos_ordenados_decrescente = sorted(precos, reverse=True)
print(f"Preços ordenados em ordem decrescente: {precos_ordenados_decrescente}")
# Saída:
# Preços ordenados em ordem decrescente: [25, 20, 15, 10, 5]
```

Explicação:

- Temos uma lista simples de `precos`.
- `sorted(precos)` ordena a lista em ordem crescente por padrão.
- `sorted(precos, reverse=True)` ordena a lista em ordem decrescente.

Aplicações Práticas de Funções de Alta Ordem

Exemplo 2: Ordenando uma lista de dicionários

```python
produtos = [
    {"nome": "Camiseta", "preco": 50, "estoque": 100},
    {"nome": "Calça", "preco": 100, "estoque": 50},
    {"nome": "Tênis", "preco": 150, "estoque": 20},
    {"nome": "Meia", "preco": 10, "estoque": 200}
]

def obter_preco(produto):
    return produto["preco"]

produtos_ordenados_por_preco = sorted(produtos, key=obter_preco)
print("Produtos ordenados por preço:")
print(produtos_ordenados_por_preco)
# Saída:
# Produtos ordenados por preço
# [
#   {'nome': 'Meia', 'preco': 10, 'estoque': 200},
#   {'nome': 'Camiseta', 'preco': 50, 'estoque': 100},
#   {'nome': 'Calça', 'preco': 100, 'estoque': 50},
#   {'nome': 'Tênis', 'preco': 150, 'estoque': 20}
# ]
```

Explicação:

- Agora temos uma lista de `produtos`, onde cada produto é um dicionário com `nome`, `preco` e `estoque`.
  - **Se tentássemos usar `sorted(produtos)` diretamente, o Python não saberia como comparar os dicionários, pois não há uma ordem padrão definida para eles.**
- Definimos uma função `obter_preco` que recebe um `produto` (um dicionário) e retorna o valor da chave "preco".
- Usamos `sorted(produtos, key=obter_preco)` para ordenar a lista de produtos por preço.
  - **Aqui, `sorted` chama `obter_preco` para cada produto na lista. O valor retornado por `obter_preco` (o preço) é usado como chave para a ordenação.**
  - O Python usa esses valores retornados para comparar os produtos entre si e determinar a ordem correta.

Aplicações Práticas de Funções de Alta Ordem

map()

Motivação:

Imagine que você tem uma lista de números e quer aplicar uma operação a cada um deles, como elevar ao quadrado. Como você faria isso de forma eficiente?

Explicação do Conceito:

- `map(funcao, iteravel)`: Aplica a `funcao` a cada item do `iteravel` e retorna um iterador com os resultados.
    - `map` **chama a função fornecida uma vez para cada elemento do iterável, passando o elemento como argumento para a função. O valor de retorno da função é adicionado ao iterador de resultados.**

Uma ressalva importante:

Você pode estar pensando que esse problema poderia ser facilmente resolvido com compreensão de lista, e você estaria certo! Por exemplo, para elevar todos os números de uma lista ao quadrado, poderíamos usar a seguinte compreensão de lista:

```
numeros = [1, 2, 3, 4, 5]
quadrados = [x ** 2 for x in numeros]
print(f"Quadrados: {quadrados}")
# Saída:
# [1, 3, 9, 16, 25]
```

Isso produziria o mesmo resultado que a função `map`. **No entanto, o objetivo aqui é demonstrar que existe uma função de alta ordem,** `map()`**, que oferece uma alternativa para realizar essa mesma tarefa.** Compreender o `map()` é importante para entender o conceito mais amplo de funções de alta ordem e como elas são usadas em diferentes contextos. Além disso, você pode encontrar o `map()` em códigos existentes e em outras linguagens de programação funcionais.

Aplicações Práticas de Funções de Alta Ordem

Exemplo: Usando map() para elevar números ao quadrado

```python
numeros = [1, 2, 3, 4, 5]

def quadrado(x):
    return x ** 2

quadrados = list(map(quadrado, numeros))
print(f"Quadrados: {quadrados}")
# Saída:
# Quadrados: [1, 3, 9, 16, 25]
```

Explicação:

- Temos uma lista de `numeros`.
- Definimos uma função `quadrado` que eleva um número ao quadrado.
- `map(quadrado, numeros)` aplica a função `quadrado` a cada número na lista `numeros`.
    - `map` chama `quadrado` para cada número, passando o número como argumento (`quadrado(1)`, `quadrado(2)`, etc.). O resultado de cada chamada (o quadrado do número) é adicionado ao iterador de resultados.
- Convertemos o iterador retornado por `map` para uma lista usando `list()`.

Aplicações Práticas de Funções de Alta Ordem

filter()

Motivação:

E se você quisesse filtrar uma lista, mantendo apenas os elementos que satisfazem uma determinada condição?

Explicação do Conceito:

- `filter(funcao, iteravel)`: Retorna um iterador com os itens do `iteravel` para os quais a `funcao` retorna `True`.
    - `filter` **chama a função fornecida uma vez para cada elemento do iterável. Se a função retornar** `True` **para um determinado elemento, esse elemento é adicionado ao iterador de resultados. Caso contrário, o elemento é ignorado.**
    - **A função passada para** `filter` **deve retornar um valor booleano (**`True` **ou** `False`**) ou um valor que possa ser interpretado como booleano (por exemplo, 0 é considerado** `False`**, e qualquer outro número é considerado** `True`**).**

Ressalva sobre Compreensão de Lista:

Assim como no caso do `map()`, você poderia usar compreensão de lista para obter o mesmo resultado do `filter()`. Por exemplo, para filtrar números pares de uma lista, poderíamos usar a seguinte compreensão de lista:

```
numeros = [1, 2, 3, 4, 5, 6]
pares = [x for x in numeros if x % 2 == 0]
print(f"Números pares: {pares}")
# Saída:
# Números pares: [2, 4, 6]
```

Isso produziria o mesmo resultado que a função `filter()`. **No entanto, nosso objetivo aqui é demonstrar a existência e o funcionamento de outra função de alta ordem, o** `filter()`**.** Entender o `filter()` é valioso para uma compreensão mais profunda de funções de alta ordem e seus usos variados. Além disso, você pode se deparar com `filter()` em códigos existentes ou em outras linguagens que suportam programação funcional.

Aplicações Práticas de Funções de Alta Ordem

Exemplo: Usando filter() para obter números pares

```python
numeros = [1, 2, 3, 4, 5, 6]

def eh_par(x):
    # Retorna True se x for par, False caso contrário
    return x % 2 == 0

pares = list(filter(eh_par, numeros))
print(f"Números pares: {pares}")
print(f"Números pares: {pares}")
```

Explicação:

- Temos uma lista de `numeros`.
- Definimos uma função `eh_par` que retorna `True` se um número for par e `False` caso contrário.
- `filter(eh_par, numeros)` filtra a lista `numeros`, mantendo apenas os números para os quais `eh_par` retorna `True`.
    - `filter` chama `eh_par` para cada número, passando o número como argumento (`eh_par(1)`, `eh_par(2)`, etc.). Se `eh_par` retornar `True` (ou seja, se o número for par), o número é adicionado ao iterador de resultados.
- Convertemos o iterador retornado por `filter` para uma lista usando `list()`.

Resumo

Partimos para a **aplicação prática** de HOFs. Exploramos como funções de alta ordem são a espinha dorsal de funções comuns da biblioteca padrão do Python, como `max()`, `sorted()`, `map()` e `filter()`. Através de exemplos concretos, aprendemos como:

- Usar `max()` para encontrar o maior elemento de uma lista com base em **critérios personalizados**, definidos por uma função passada como argumento.
- Ordenar listas de forma flexível usando `sorted()`, especificando **chaves de ordenação complexas** através de funções.
- Transformar elementos de uma lista de forma eficiente com `map()`, aplicando uma função a cada elemento.
- Filtrar elementos indesejados de uma lista usando `filter()`, mantendo apenas aqueles que satisfazem uma condição definida por uma função.

Para consolidar nosso entendimento, **mergulhamos nas entranhas do** `max()`, implementando uma versão simplificada chamada `meu_max()`. Essa experiência prática nos permitiu visualizar como uma função de alta ordem opera internamente, reforçando a conexão entre a teoria e a prática. Vimos como a função `key`, passada como argumento, é utilizada para determinar o maior elemento, personalizando o comportamento do `max()` de acordo com nossas necessidades.

Funções lambda

O que são Funções Lambda?

Motivação:

Às vezes, precisamos de funções simples que serão usadas apenas uma vez, como no parâmetro `key` de `sorted()` ou `max()`. Definir uma função nomeada para isso pode ser desnecessário e verboso. Como podemos criar funções simples e descartáveis de forma rápida e concisa?

Explicação do Conceito:

Funções lambda, também conhecidas como funções anônimas, são funções pequenas e sem nome definido. Elas são definidas usando a palavra-chave `lambda`, seguida pelos argumentos, dois pontos e a expressão que será retornada.

Funções lambda

Exemplo: Usando funções lambda com sorted(), map(), e as outras HOFs

```
produtos = [
    {"nome": "Camiseta", "preco": 50, "estoque": 100},
    {"nome": "Calça", "preco": 100, "estoque": 50},
    {"nome": "Tênis", "preco": 150, "estoque": 20},
    {"nome": "Meia", "preco": 10, "estoque": 200}
]

# Ordenando produtos por estoque usando lambda
produtos_ordenados_por_estoque = sorted(produtos, key=lambda produto:
produto["estoque"])
print("Produtos ordenados por estoque:")
print(produtos_ordenados_por_estoque)
# Saída:
# Produtos ordenados por estoque:
# [
#   {"nome": "Tênis", "preco": 150, "estoque": 20},
#   {"nome": "Calça", "preco": 100, "estoque": 50},
#   {"nome": "Camiseta", "preco": 50, "estoque": 100},
#   {"nome": "Meia", "preco": 10, "estoque": 200}
# ]

# Usando map com lambda para obter os preços dos produtos
precos = list(map(lambda produto: produto["preco"], produtos))
print(f"Preços dos produtos: {precos}")
# Saída:
# Preços dos produtos: [50, 100, 150, 10]

# Usando filter com lambda para obter produtos com estoque maior que 50
produtos_em_estoque = list(filter(lambda produto: produto["estoque"] > 50, produtos))
print("Produtos com estoque maior que 50:")
print(produtos_em_estoque)
# Saída:
# Produtos com estoque maior que 50:
# [
#   {"nome": "Camiseta", "preco": 50, "estoque": 100},
#   {"nome": "Meia", "preco": 10, "estoque": 200}
# ]


# Usando max com lambda para encontrar o produto mais caro
produto_mais_caro = max(produtos, key=lambda produto: produto["preco"])
print(f"Produto mais caro: {produto_mais_caro}")
# Saída:
# Produto mais caro: {"nome": "Tênis", "preco": 150, "estoque": 20}

# Usando min com lambda para encontrar o produto com menor estoque
produto_menor_estoque = min(produtos, key=lambda produto: produto["estoque"])
print(f"Produto com menor estoque: {produto_menor_estoque}")
# Saída:
# Produto com menor estoque: {"nome": "Tênis", "preco": 150, "estoque": 20}
```

Funções lambda

Explicação:

- Em vez de definir funções nomeadas, usamos funções lambda diretamente no parâmetro `key` de `sorted()`, `max()` e `min()`, e nas funções `map()` e `filter()`.
- A função lambda `lambda produto: produto["estoque"]` retorna o valor da chave "estoque" para cada produto.
- Da mesma forma, usamos funções lambda com `map`, `filter`, `max` e `min` para simplificar a definição de funções simples.

Comparações

Comparando com funções nomeadas, as funções lambda são mais concisas e convenientes para casos de uso simples e únicos. Elas tornam o código mais limpo e evitam a poluição do namespace com nomes de funções que não serão reutilizadas.

Próximos Passos:

- Pratique a criação de suas próprias funções de alta ordem para resolver problemas específicos.
- Explore outras funções da biblioteca padrão do Python que utilizam funções de alta ordem.
- Estude sobre decoradores em Python, que são uma aplicação avançada de funções de alta ordem.

Resumo

Funções lambda são uma forma concisa e elegante de definir funções simples, especialmente úteis quando precisamos de uma função descartável para passar como argumento para uma função de alta ordem. Vimos como as funções lambda podem simplificar nosso código, tornando-o mais legível e eficiente, principalmente em conjunto com `max()`, `min()`, `sorted()`, `map()` e `filter()`.

Em resumo, você agora possui um sólido conhecimento sobre:

- O que são funções de alta ordem e como elas funcionam.
- Como passar funções como argumentos para outras funções.
- Como funções de alta ordem são usadas na prática em funções comuns do Python.
- O funcionamento interno de uma função de alta ordem, exemplificado pelo `max()`.
- O que são funções lambda e como utilizá-las para simplificar seu código.

Com esse conhecimento, você está pronto para escrever código Python mais limpo, modular, flexível e expressivo. Você será capaz de resolver problemas complexos de forma elegante, utilizando o poder das funções de alta ordem para manipular dados e controlar o fluxo do seu programa de maneira eficiente.

Decorators: Fazendo Mágica com Funções

Decorators são uma ferramenta poderosa e expressiva em Python que permite modificar ou aprimorar funções de uma maneira elegante e legível. Eles são frequentemente descritos como "anotações especiais" que alteram o comportamento de uma função.

Nesta aula, não vamos nos aprofundar em como criar decorators do zero, pois isso envolve conceitos mais avançados. Em vez disso, vamos focar em entender e utilizar alguns decorators comuns que já existem no ecossistema Python. Ao final da aula, você terá uma compreensão clara de como esses decorators funcionam e como eles podem tornar seu código mais eficiente e legível.

Se você estiver interessado em aprender mais sobre a criação de decorators personalizados, aqui estão alguns recursos úteis:

- Real Python - Primer on Python Decorators
- PEP 318 – Decorators for Functions and Methods

Vamos começar nossa jornada pelos decorators com um exemplo prático.

@cache: Memorizando Resultados para Otimização

Motivação: Funções Demoradas e a Necessidade de Otimização

Imagine que você está trabalhando em um projeto que envolve cálculos matemáticos complexos ou consultas a um banco de dados. Você tem uma função que, embora crucial para o seu sistema, é bastante demorada para executar.

Vamos criar um exemplo fictício para ilustrar isso. Suponha que você tenha uma função que calcula a soma de todos os números inteiros de 1 até um número `n` fornecido como parâmetro.

```python
def soma_demorada(n):
    """Calcula a soma de todos os números de 1 até n."""
    soma = 0
    for i in range(1, n + 1):
        soma += i
    print(f"Soma calculada para n = {n}")
    return soma

# Exemplo de uso
print("Iniciando o programa...")
resultado = soma_demorada(100000000)
print(f"Resultado: {resultado}")
```

Se você executar este código, notará que ele leva alguns segundos para ser concluído. Isso ocorre porque o loop `for` dentro da função `soma_demorada` é executado muitas vezes (100 milhões de vezes), o que leva tempo.

Agora, imagine que seu sistema precisa chamar essa função várias vezes, muitas vezes com os mesmos parâmetros. Seria ineficiente recalcular o resultado toda vez que a função é chamada com um valor de `n` que já foi processado anteriormente.

Decorators: Fazendo Mágica com Funções

Solução: Memorizando Resultados com @cache

É aqui que o decorator `@cache` (disponível no módulo `functools` a partir do Python 3.9, ou `lru_cache` em versões anteriores) entra em cena. Ele permite que você memorize (ou "cacheie") os resultados de uma função. Quando a função é chamada novamente com os mesmos argumentos, o `@cache` intercepta a chamada e retorna o resultado memorizado, evitando a reexecução da função.

Vamos aplicar o `@cache` à nossa função `soma_demorada`:

```
from functools import cache

@cache
def soma_demorada(n):
    """Calcula a soma de todos os números de 1 até n."""
    soma = 0
    for i in range(1, n + 1):
        soma += i
    print(f"Soma calculada para n = {n}")
    return soma

# Exemplo de uso
print("Iniciando o programa...")
print(f"Primeira chamada (n=100 milhões): {soma_demorada(100000000)}")
print(f"Segunda chamada (n=50 milhões): {soma_demorada(50000000)}")
print(f"Terceira chamada (n=100 milhões): {soma_demorada(100000000)}")
print(f"Quarta chamada (n=50 milhões): {soma_demorada(50000000)}")
```

Saída:

```
Iniciando o programa...
Soma calculada para n = 100000000
Primeira chamada (n=100 milhões): 5000000050000000
Soma calculada para n = 50000000
Segunda chamada (n=50 milhões): 1250000025000000
Terceira chamada (n=100 milhões): 5000000050000000
Quarta chamada (n=50 milhões): 1250000025000000
```

Ao executar este código, você observará o seguinte comportamento:

1. A primeira chamada `soma_demorada(100000000)` executa a função normalmente, exibe "Soma calculada para n = 100 milhões" e retorna o resultado.
2. A segunda chamada `soma_demorada(50000000)` também executa a função normalmente.
3. A terceira chamada `soma_demorada(100000000)` **não** exibe "Soma calculada para n = 100 milhões". Em vez disso, ela retorna o resultado memorizado da primeira chamada, sem reexecutar a função.
4. A quarta chamada `soma_demorada(50000000)` também retorna o resultado memorizado da segunda chamada sem reexecutar a função.

Isso demonstra claramente como o `@cache` evita que a função seja chamada desnecessariamente, economizando tempo de processamento.

Decorators: Fazendo Mágica com Funções

Comparação: Com e Sem @cache

Sem o `@cache`, cada chamada à função `soma_demorada` resultaria na reexecução de todo o loop, independentemente de os argumentos serem os mesmos de chamadas anteriores. Com o `@cache`, apenas a primeira chamada com um determinado conjunto de argumentos executa a função; as chamadas subsequentes com os mesmos argumentos retornam o resultado memorizado.

Vantagens do `@cache`:

- **Otimização de desempenho:** Reduz significativamente o tempo de execução em casos onde a função é chamada repetidamente com os mesmos argumentos.
- **Simplicidade:** Fácil de aplicar, exigindo apenas a adição de uma linha de código acima da definição da função.

Limitações do `@cache`:

- **Uso de memória:** Armazena os resultados em memória, o que pode ser problemático para funções com um grande número de combinações possíveis de argumentos ou para resultados muito grandes.
- **Não é adequado para funções com efeitos colaterais:** Funções que modificam variáveis globais ou dependem de fatores externos (como a hora atual) não devem ser cacheadas, pois o resultado memorizado pode não refletir o estado atual do sistema.
- **Só funciona para funções com argumentos que sejam "hasheáveis"**: strings, números, tuplas, mas não listas, dicionários ou outros tipos mutáveis.

Resumo

O decorator `@cache` é uma ferramenta poderosa para otimizar funções que são computacionalmente intensivas e são chamadas repetidamente com os mesmos argumentos. Ele memoriza os resultados, evitando a reexecução desnecessária da função e economizando tempo de processamento. No entanto, é importante estar ciente de suas limitações e usá-lo apenas quando apropriado.

Decorators: Fazendo Mágica com Funções

@dataclass: Simplificando a Criação de Classes de Dados

2.1 Motivação: A Tarefa Repetitiva de Criar Classes para Armazenar Dados

Em muitos programas, é comum criarmos classes simples cujo principal objetivo é armazenar dados. Por exemplo, considere as seguintes classes:

```
class Cliente:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

class Produto:
    def __init__(self, nome, preco):
        self.nome = nome
        self.preco = preco

class Pedido:
    def __init__(self, cliente, produtos):
        self.cliente = cliente
        self.produtos = produtos
```

Essas classes são bastante simples, mas observe que precisamos escrever um método `__init__` para cada uma delas, apenas para inicializar os atributos. Isso pode se tornar repetitivo e tedioso, especialmente quando temos muitas classes semelhantes ou quando uma classe tem muitos atributos.

Além disso, se quisermos adicionar outros métodos comuns, como uma representação textual (`__repr__`) ou métodos de comparação (`__eq__`, `__lt__`, etc.), teremos que escrevê-los manualmente para cada classe, o que aumenta ainda mais a quantidade de código repetitivo.

Decorators: Fazendo Mágica com Funções

Solução: @dataclass para a Criação Automática de Métodos Comuns

O decorator `@dataclass` (introduzido no Python 3.7 no módulo `dataclasses`) foi criado para resolver exatamente esse problema. Ele automatiza a geração de métodos especiais, como `__init__`, `__repr__`, `__eq__`, e outros, para classes que são principalmente usadas para armazenar dados.

Vamos refatorar as classes `Cliente`, `Produto` e `Pedido` usando `@dataclass`:

```
from dataclasses import dataclass

@dataclass
class Cliente:
    nome: str
    idade: int

@dataclass
class Produto:
    nome: str
    preco: float

@dataclass
class Pedido:
    cliente: Cliente
    produtos: list
```

Observe como o código ficou mais conciso. Não precisamos mais escrever o método `__init__` explicitamente. O `@dataclass` se encarrega de gerar esse método automaticamente, com base nos atributos que declaramos.

Além disso, podemos instanciar e usar essas classes da mesma forma que antes:

```
cliente = Cliente("Alice", 30)
produto1 = Produto("Camiseta", 29.90)
produto2 = Produto("Calça", 99.90)
pedido = Pedido(cliente, [produto1, produto2])

print(cliente.nome)
print(produto1.preco)
print(produto2.nome)

print(cliente)
print(produto1)
print(pedido)
```

A saída será algo como:

```
Alice
29.9
Calça
Cliente(nome='Alice', idade=30)
Produto(nome='Camiseta', preco=29.9)
Pedido(cliente=Cliente(nome='Alice', idade=30), produtos=[Produto(nome='Camiseta',
preco=29.9), Produto(nome='Calça', preco=99.9)])
```

Veja que o `@dataclass` também gerou automaticamente uma representação textual (`__repr__`) para nossas classes.

Decorators: Fazendo Mágica com Funções

Comparação: Com e Sem @dataclass

Sem o `@dataclass`, teríamos que escrever manualmente o método `__init__` (e potencialmente outros métodos especiais) para cada classe. Isso não apenas aumenta a quantidade de código, mas também torna a definição da classe menos clara, pois a parte importante (os atributos) fica misturada com o código repetitivo dos métodos especiais.

Com o `@dataclass`, a definição da classe fica mais focada nos atributos que ela armazena, tornando o código mais legível e fácil de manter.

Vantagens do `@dataclass`:

- **Redução de código repetitivo:** Elimina a necessidade de escrever manualmente métodos comuns como `__init__` e `__repr__`.
- **Maior clareza:** A definição da classe fica mais focada nos atributos que ela armazena.
- **Facilidade de manutenção:** Menos código para escrever e manter.
- **Geração de código útil:** Além do `__init__` e do `__repr__`, o `@dataclass` também pode gerar automaticamente métodos de comparação e outros métodos úteis.

Limitações do `@dataclass`:

- **Menor controle:** Como os métodos são gerados automaticamente, temos menos controle sobre sua implementação específica. No entanto, ainda é possível sobrescrever os métodos gerados pelo `@dataclass` se precisarmos de um comportamento personalizado.
- **Disponibilidade:** Só está disponível a partir do Python 3.7.

Resumo

O decorator `@dataclass` é uma ferramenta valiosa para simplificar a criação de classes que são principalmente usadas para armazenar dados. Ele automatiza a geração de métodos comuns, reduzindo a quantidade de código repetitivo e tornando a definição da classe mais clara e fácil de manter.

Decorators: Fazendo Mágica com Funções

Conclusão e Outros Decorators Comuns

Decorators são uma forma poderosa de modificar ou aprimorar funções em Python.

Eles são aplicados usando a sintaxe `@nome_do_decorator` acima da definição da função ou da classe.

Decorators podem adicionar funcionalidades, modificar o comportamento ou simplificar a criação de funções e classes.

Exploramos dois exemplos práticos de como os decorators podem simplificar e otimizar nosso código Python:

1. `@cache`: Memoriza os resultados de uma função para evitar recálculos desnecessários, melhorando o desempenho.
2. `@dataclass`: Automatiza a criação de métodos comuns em classes de dados, reduzindo código repetitivo e aumentando a clareza.

Estes são apenas dois exemplos de um vasto leque de possibilidades. Existem muitos outros decorators úteis, tanto na biblioteca padrão do Python quanto em bibliotecas de terceiros. Alguns exemplos notáveis incluem:

- `@property`: Permite que um método seja acessado como um atributo, possibilitando a criação de atributos calculados ou a realização de validações antes de definir ou retornar um valor.
- `@staticmethod` **e** `@classmethod`: Definem métodos que estão associados à classe em si, e não a instâncias da classe.
- `@abstractmethod` **(do módulo** `abc`**)**: Define métodos abstratos em classes abstratas, garantindo que as subclasses implementem esses métodos.
- `@app.route` **(do framework Flask)**: Associa uma função a uma rota específica em uma aplicação web, definindo como a aplicação deve responder a requisições HTTP para aquela rota.
- `@login_required` **(do framework Django)**: Garante que apenas usuários autenticados possam acessar uma determinada view em uma aplicação web.

Para aprofundar seus conhecimentos sobre decorators, recomendamos a leitura dos seguintes recursos:

- Real Python - Primer on Python Decorators
- PEP 318 – Decorators for Functions and Methods
- Documentação do Python sobre Decorators