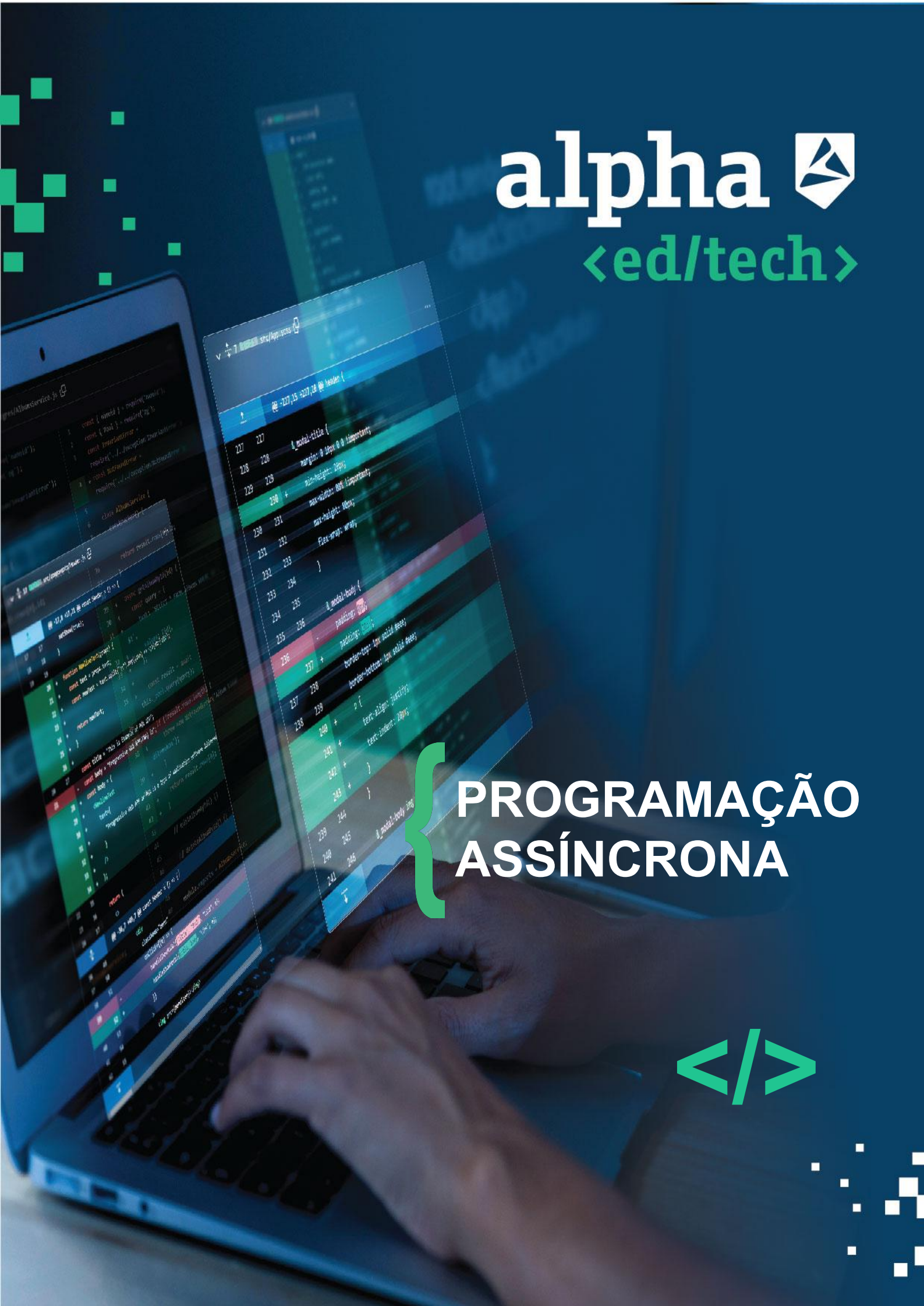
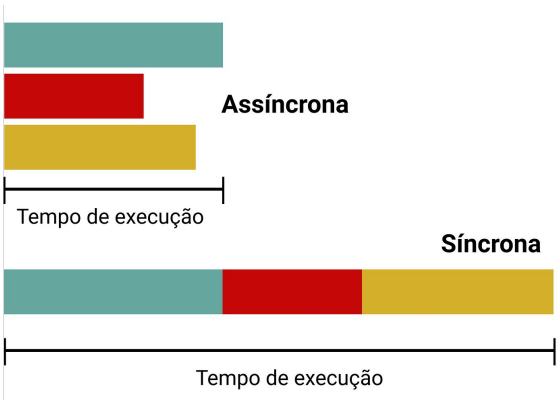

alpha
<ed/tech>
PROGRAMAÇÃO
ASSÍNCRONA
</>

Programação Assíncrona

Introdução

A assincronicidade em programação é um paradigma que se destaca pela execução de operações de forma não sequencial, permitindo que o código continue sua execução mesmo quando uma operação está em andamento. Isso é particularmente valioso em situações em que operações demoradas, como entrada/saída (I/O) ou requisições de rede, não deveriam bloquear a execução do programa.

A assincronicidade resolve esse problema, permitindo que o programa avance para outras operações enquanto espera por operações lentas. Para entender isso, é crucial compreender o conceito de "callbacks", que são funções que são chamadas após a conclusão de uma operação assíncrona. Os callbacks são uma abordagem inicial para lidar com a assincronicidade, mas podem levar a um código complicado e difícil de manter, o que destaca a necessidade de abordagens mais avançadas, como corrotinas.



Na imagem acima podemos ver a diferença no tempo de execução ao compararmos uma operação assíncrona e uma operação síncrona.

DICA!

Ao adotar a assincronicidade, os desenvolvedores podem melhorar significativamente o desempenho e a eficiência de seus programas, especialmente em cenários onde a espera por operações de entrada/saída é comum. Isso é particularmente evidente em aplicações web, servidores, automação de processos assíncronos, entre outros.

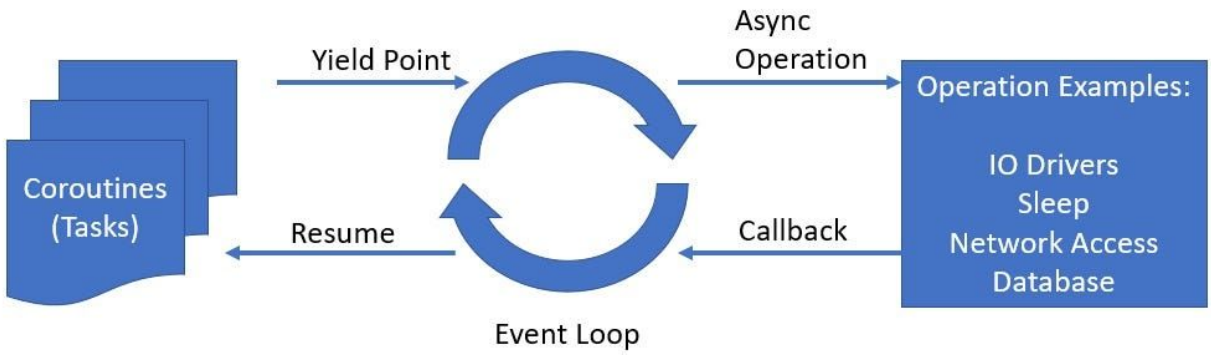
Na programação síncrona convencional, as operações são executadas de maneira sequencial, uma após a outra. Se uma operação demorada ocorre, todo o programa aguarda até que ela seja concluída. Isso pode resultar em desperdício de tempo, especialmente em situações em que outras tarefas poderiam ser realizadas enquanto aguardamos uma resposta externa.

Podemos compreender a necessidade de repensar a execução sequencial de operações, introduzindo conceitos como callbacks, promessas, corrotinas e event loops para permitir uma execução mais eficiente e responsiva de programas Python.

Programação Assíncrona

Callbacks

Callbacks desempenham um papel fundamental na introdução à assincronicidade, sendo uma abordagem inicial para lidar com operações assíncronas em Python. Um callback é, essencialmente, uma função que é passada como argumento para outra função e é chamada após a conclusão de uma operação específica. No contexto assíncrono, os callbacks são frequentemente usados para lidar com o retorno de operações demoradas, como leituras de arquivos, requisições de rede ou consultas a bancos de dados.



Como podemos ver, após uma operação assíncrona (lado direito) é executado um callback para que os dados obtidos possam integrar o fluxo principal da aplicação.

Callbacks desempenham um papel fundamental na assincronicidade, sendo uma abordagem inicial para lidar com operações assíncronas em Python. Um callback é, essencialmente, uma função que é passada como argumento para outra função e é chamada após a conclusão de uma operação específica. No contexto assíncrono, os callbacks são frequentemente usados para lidar com o retorno de operações demoradas, como leituras de arquivos, requisições de rede ou consultas a bancos de dados.

Na programação síncrona, uma função é chamada e espera a conclusão antes de passar para a próxima instrução. Isso pode resultar em bloqueios, especialmente em operações demoradas.

Eles permitem que uma função seja registrada para ser executada quando uma operação assíncrona for concluída, permitindo que o programa continue sua execução.

Em resumo, a compreensão dos callbacks é crucial para começar a trabalhar com assincronicidade em Python. Eles proporcionam uma introdução valiosa a conceitos que serão explorados de forma mais aprofundada.

DICA!

É interessante pesquisar sobre existência padrões de projeto que envolvem callbacks, como o padrão Observer, onde objetos interessados (observadores) registram callbacks para eventos específicos.

ATENÇÃO!

Callback Hell, também conhecido como "Pyramid of Doom" ou "Callback Pyramid", é um termo utilizado para descrever uma situação em que o código assíncrono contém uma aninhamento excessivo de callbacks, resultando em um código difícil de ler, entender e manter. Esse problema é mais evidente quando se lida com uma sequência complexa de operações assíncronas que dependem uma da outra.

Programação Assíncrona

Promises e Futures: Introdução

Na nossa jornada pela assincronicidade em Python, agora vamos explorar dois conceitos essenciais: Promessas e Futures.

Essas estruturas são poderosas e oferecem uma abordagem mais organizada e legível para lidar com operações assíncronas em comparação com o uso tradicional de callbacks.

Uma promessa é uma abstração que representa o resultado de uma operação assíncrona.

Em Python, a biblioteca *concurrent.futures* oferece a classe ***Future***, que é a implementação desse conceito.

Promessas podem ser resolvidas (com sucesso) ou rejeitadas (com falha).

Os Futures em Python são frequentemente usados em conjunto com o módulo *concurrent.futures*, fornecendo uma maneira de representar valores que podem não estar disponíveis no momento da chamada.

Os métodos *add_done_callback()*, *result()*, e *exception()* são fundamentais para trabalhar com Futures.

```python
from concurrent.futures import Future

# Criação de uma instância de Future
promise = Future()

# Resolvendo a promessa com sucesso
promise.set_result("Resultado da operação")

# Rejeitando a promessa com erro
promise.set_exception(Exception("Erro na operação"))
```

```python
from concurrent.futures import Future

def async_operation():
    future_result = Future()

    future_result.set_result("Resultado da operação assíncrona")
    return future_result

def handle_result(future):
    result = future.result()
        print(f"Resultado da operação: {result}")

promise = async_operation()
promise.add_done_callback(handle_result)
```

DICA!

É interessante ver como estamos simulando uma operação assíncrona, em que em ambos os casos atribuímos o resultado da Future através da função 'set_result()'. E assim recebemos o resultado com uma função 'result()'.

Mas pensemos um pouco, este Future nos lembra de alguma estrutura que já vimos?

E se você pensou em Objetos acertou!

Future é uma classe que instanciamos, portanto, a instância dela é um objeto.

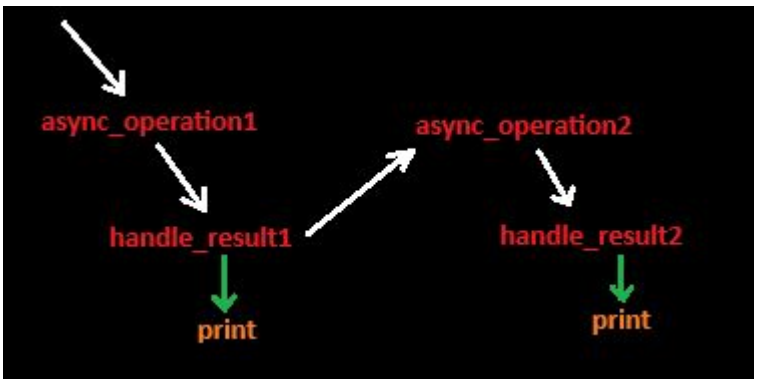
Programação Assíncrona

Promises e Futures: Encadeamento

Encadeamento de Futures permite a execução sequencial de operações assíncronas, onde o resultado de uma operação é usado como entrada para a próxima.
O método add_done_callback() é utilizado para encadear as operações.

```
from concurrent.futures  import Future

def async operation1 ():
    future result = Future()
    future result.set result( "Resultado da operação 1" )
    return future_result

def async operation2 (input data):
    future result = Future()

    future result.set result( f"Resultado da operação 2 com {input_data}" )
    return future_result

def handle result1 (future):
    result = future.result()
    print(result)

    promise2 = async operation2(result)
    promise2.add_done_callback(handle_result2)

def handle result2 (future):
    result = future.result()
    print(result)

promise1 = async operation1()
promise1.add_done_callback(handle_result1)
```

Encadeamento dessas funções usando add_done_callback() para que a segunda operação dependa do resultado da primeira.



Programação Assíncrona

Promises e Futures: Excessões

A gestão de erros é uma parte crucial ao lidar com operações assíncronas. O módulo *concurrent.futures* fornece maneiras eficientes de tratar exceções que podem ocorrer durante a execução de uma operação assíncrona representada por um *Future*.

O método *add_done_callback()* é utilizado para registrar uma função de retorno de chamada que será acionada quando o *Future* estiver concluído, seja com sucesso (*set_result*) ou com erro (*set_exception*).

Dentro da função de retorno de chamada, uma tentativa (*try*) e exceção (*except*) são usadas para capturar e tratar qualquer erro que possa ter ocorrido durante a operação assíncrona.

```
from concurrent.futures import Future

def async_operation():
    future_result = Future()

    future_result.set_exception(Exception("Erro na operação assíncrona"))
    return future_result

def handle_error(future):
    try:
        result = future.result()
        print(f"Resultado da operação: {result}")
    except Exception as e:
        print(f"Erro na operação: {e}")

promise = async_operation()
promise.add_done_callback(handle_error)
```

- *async_operation* simula uma operação assíncrona que, neste caso, é configurada para rejeitar a promessa (*set_exception*) com uma exceção personalizada.
- *handle_error* é a função de retorno de chamada registrada com *add_done_callback*.
- Dentro de *handle_error*, há um bloco *try* e *except* que tenta obter o resultado da operação assíncrona.

Se houver uma exceção, ela será capturada e tratada, imprimindo a mensagem de erro.
Essa abordagem permite que o código trate erros de forma eficaz, proporcionando um controle mais preciso sobre o fluxo do programa, mesmo em situações de operações assíncronas que podem resultar em exceções.

Programação Assíncrona

Promises e Futures: Asyncio

O módulo asyncio em Python fornece suporte nativo para programação assíncrona, permitindo que você escreva código concorrente e assíncrono de maneira mais eficiente. Vamos explorar mais detalhadamente os principais conceitos e funcionalidades do *asyncio*.

1. **Corrotinas (async def):**
   - São funções assíncronas que possibilitam a execução concorrente de tarefas sem bloqueio.
   - Utilizam *await* para pausar a execução enquanto aguardam operações assíncronas.

2. **Loop de Eventos (asyncio.get_event_loop()):**
   - Orquestra a execução de corrotinas, decidindo quando cada uma deve ser pausada e retomada.
   - Gerencia a concorrência, permitindo que várias tarefas assíncronas sejam executadas eficientemente.

3. **Tarefas (asyncio.create_task()):**
   - Representam a execução concorrente de corrotinas.
   - Criadas usando *asyncio.create_task()* para permitir a execução paralela e a gestão flexível de várias tarefas.

4. **asyncio.sleep():**
   - Corotina que pausa a execução assíncrona por um período especificado.
   - Útil para simular a espera ou para introduzir atrasos controlados durante a execução.

5. **asyncio.gather():**
   - Corotina que agendia a execução simultânea de várias corrotinas.
   - Melhora a eficiência ao permitir que várias tarefas assíncronas sejam processadas ao mesmo tempo.

6. **Tratamento de Eventos (asyncio.Event):**
   - Facilita a sincronização entre corrotinas, permitindo a comunicação e coordenação eficientes.
   - *O asyncio.Event* fornece métodos como *set()* para sinalizar e *wait()* para esperar a sinalização.

Programação Assíncrona

Promises e Futures: Asyncio

Você deve ter se deparado com muitos conceitos novos, mas tranquilize-se. Eles podem parecer um pouco intangíveis agora, mas pra que você veja o quão simples são, vamos aos exemplos práticos:

1. **Corrotinas (async/await)**:
    - *asyncio* é baseado no conceito de corrotinas, que são funções assíncronas marcadas com os decoradores *async def*.
    - Corrotinas utilizam *await* para pausar a execução enquanto esperam por operações assíncronas, permitindo que outras tarefas executem durante esse tempo.

```
import asyncio

async def exemplo_corrotina():
    print("Início da corrotina")
    await asyncio.sleep(2)  # Simulando uma operação assíncrona
    print("Fim da corrotina")

asyncio.run(exemplo_corrotina())
```

2. **Loop de Eventos (*asyncio.run()* e *asyncio.get_event_loop()*)**:
    - O loop de eventos é o coração do *asyncio*. Ele gerencia e coordena a execução de tarefas assíncronas.
    - *asyncio.run()* é utilizado para executar uma corrotina principal, simplificando a inicialização e finalização do loop de eventos.
    - *asyncio.get_event_loop()* obtém a instância do loop de eventos, permitindo o controle mais avançado sobre sua execução.

```
import asyncio

async def corrotina_principal():
    # ... código da corrotina principal ...

asyncio.run(corrotina_principal())
```

É possível ver que corrotinas e o loop de eventos trabalham em conjunto para fornecer uma abordagem eficaz e intuitiva para programação assíncrona em Python.

As corrotinas são as unidades de trabalho assíncrono, enquanto o loop de eventos é responsável por orquestrar a execução concorrente dessas corrotinas, garantindo eficiência e manutenção da ordem de execução.

Programação Assíncrona

Promises e Futures: Asyncio

3. **Tarefas (*asyncio.create_task()*)**:

   - *asyncio.create_task()* é usado para criar tarefas assíncronas que são executadas concorrentemente no loop de eventos.
   - Tarefas são frequentemente usadas para representar a execução paralela de múltiplas corrotinas.

```
import asyncio

async def corrotina1():
    # ... código da corrotina 1 ...

async def corrotina2():
    # ... código da corrotina 2 ...

async def corrotina principal():
    task1 = asyncio.create_task(corrotina1())
    task2 = asyncio.create task(corrotina2())
    await asyncio.gather(task1, task2)

asyncio.run(corrotina_principal())
```

4. ***asyncio.sleep()* e *asyncio.gather()***:

   - *asyncio.sleep()* é uma corrotina que suspende a execução por um determinado número de segundos, simulando uma espera assíncrona.
   - *asyncio.gather()* é usado para agendar a execução simultânea de várias corrotinas.

```
import asyncio

async def corrotina1():
    print("Corrotina 1 iniciada")
    await asyncio.sleep(2)
    print("Corrotina 1 concluída")

async def corrotina2():
    print("Corrotina 2 iniciada")
    await asyncio.sleep(1)
    print("Corrotina 2 concluída")

async def corrotina principal():
    await asyncio.gather(corrotina1(), corrotina2())

asyncio.run(corrotina_principal())
```

Programação Assíncrona

Promises e Futures: Asyncio

5. **Tratamento de Eventos (*asyncio.Event*)**:

   - *asyncio* oferece objetos *Event* para coordenar a execução entre corrotinas. Um evento é sinalizado (*set()*) para indicar que algo aconteceu.
   - Corrotinas podem esperar (*wait()*) até que um evento seja sinalizado.

```
import asyncio

async def corrotina1(event):
    print("Corrotina 1 esperando evento")
    await event.wait()
    print("Corrotina 1 continua após o evento")

async def corrotina2(event):
    print("Corrotina 2 sinalizando o evento")
    event.set()

async def corrotina_principal():
    event = asyncio.Event()
    task1 = asyncio.create_task(corrotina1(event))
    task2 = asyncio.create_task(corrotina2(event))
    await asyncio.gather(task1, task2)

asyncio.run(corrotina_principal())
```

Resumo das Diferenças entre *asyncio.create_task* e *asyncio.gather*:

- asyncio.create_task() é utilizado quando você deseja iniciar uma única corrotina como uma tarefa independente.
- asyncio.gather() é usado quando você deseja executar várias corrotinas simultaneamente e aguardar a conclusão de todas.
- asyncio.gather() retorna uma corrotina, enquanto asyncio.create_task() retorna um objeto asyncio.Task.
- asyncio.gather() é mais adequado para coordenar a execução paralela de múltiplas corrotinas,
- enquanto asyncio.create_task() é mais útil quando você precisa de uma referência à tarefa específica para monitoramento ou cancelamento.

Resumindo, os conceitos do asyncio trabalham de forma integrada para criar um ambiente assíncrono eficiente em Python. Corrotinas possibilitam a execução concorrente e são coordenadas pelo loop de eventos, enquanto tarefas são gerenciadas para execução paralela. O asyncio.sleep() controla pausas, asyncio.gather() facilita a execução simultânea, e asyncio.Event permite a comunicação eficaz entre tarefas assíncronas. Em conjunto, o asyncio fornece uma base sólida para desenvolver de forma eficiente, legível e escalável em projetos assíncronos. A compreensão desses conceitos é crucial para tirar o máximo proveito dessa biblioteca em programação assíncrona em Python.

Programação Assíncrona

Exercícios Práticos: Asyncio

</>

1. **Entendendo Corrotinas:**
   - Crie uma corrotina simples que imprima uma mensagem antes e depois de uma pausa de 2 segundos usando asyncio.sleep().
   - Execute a corrotina usando asyncio.run().

2. **Criando e Executando Tarefas:**
   - Crie duas corrotinas que simulam tarefas assíncronas simples.
   - Use asyncio.create_task() para executar essas corrotinas concorrentemente.
   - Adicione uma pausa entre as corrotinas para observar a execução simultânea.

3. **Coordenando Corrotinas:**
   - Escreva três corrotinas que realizam operações assíncronas distintas.
   - Utilize asyncio.gather() para executar as corrotinas simultaneamente.
   - Observe como a execução paralela melhora a eficiência.

4. **Sincronização entre Corrotinas:**
   - Crie duas corrotinas que se comunicam usando um objeto asyncio.Event.
   - Uma corrotina deve esperar até que o evento seja sinalizado antes de continuar.
   - A outra corrotina deve sinalizar o evento após uma pausa.

5. **Combinação de Conceitos:**
   - Crie um sistema de simulação de processamento assíncrono onde várias corrotinas representam tarefas complexas.
   - Utilize asyncio.create_task() para gerenciar a execução concorrente dessas corrotinas.
   - Integre o uso de asyncio.sleep(), asyncio.gather(), e, se necessário, asyncio.Event para criar um fluxo de controle assíncrono eficiente.