

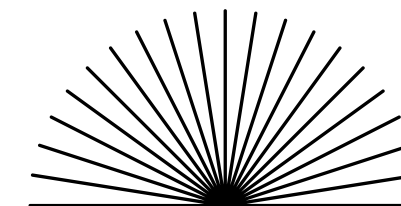
IFNMG - Campus Januária

FUNÇÕES GERADORAS & CORROTINAS ASSÍNCRONAS

Sistemas Distribuídos

Discente: Joao Marcos

Docente: Adriano



generator_function():

Funções Geradoras: São tipos especiais de funções que conseguem manter o estado atual da sua execução, mesmo após retornar um valor

```
def correr_lista():  
    lista = ['JhonnyBoy', 'Miranha', 'Badman']  
    for i in lista:  
        yield i
```

ITERADORES vs GERADORES

"Todo Gerador é um Iterador, mas nem todo Iterador é um Gerador."

Um **iterador** é uma forma manual e completa de se criar um objeto que entrega itens um por vez, exigindo uma classe com métodos `__iter__()` e `__next__()`

Um **gerador** é um atalho: uma função simples que usa `yield` para construir um iterador para você de forma automática, com um código muito mais limpo e legível.

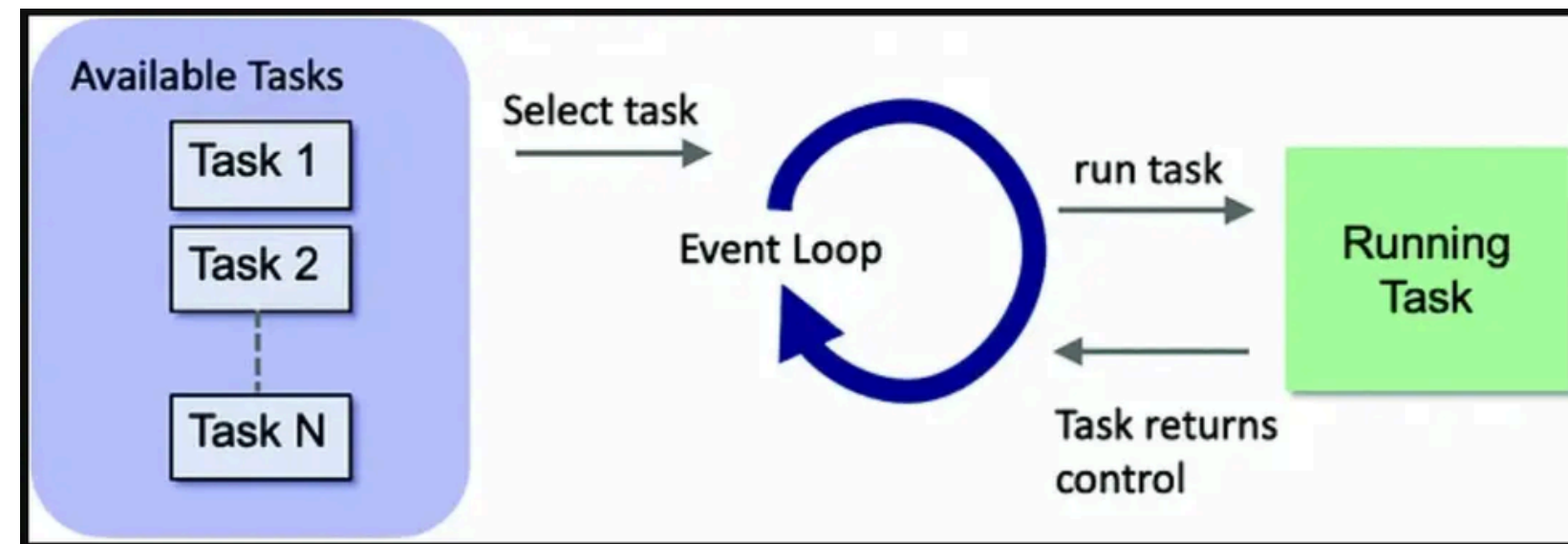
CORROTINAS ASSÍNCRONAS

Corrotinas: são funções que podem ser pausadas e retomadas, permitindo que o programa execute outras atividades enquanto uma tarefa demorada não termina.

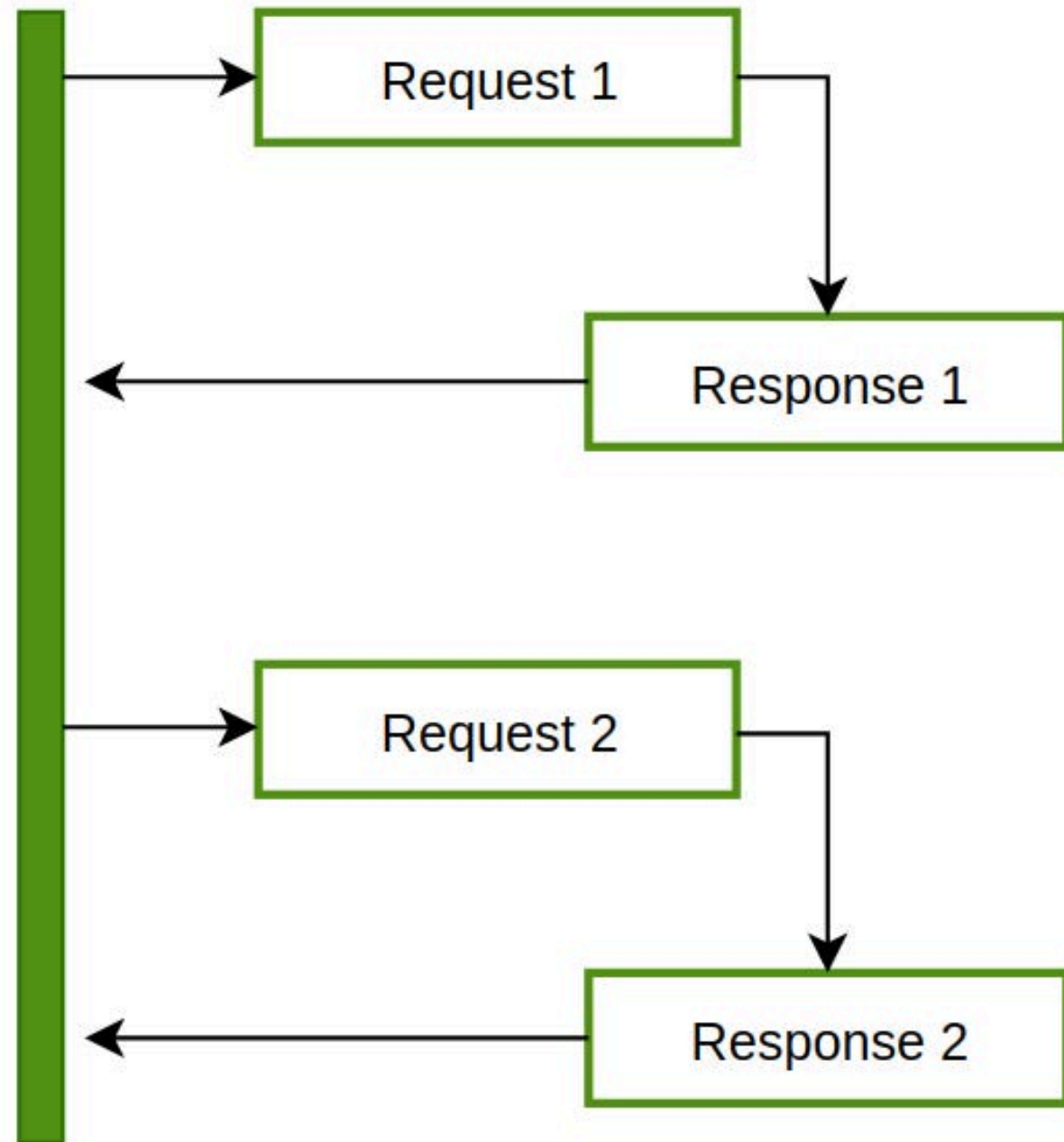
Ponto-chave: Loop de Eventos

Loop de Eventos

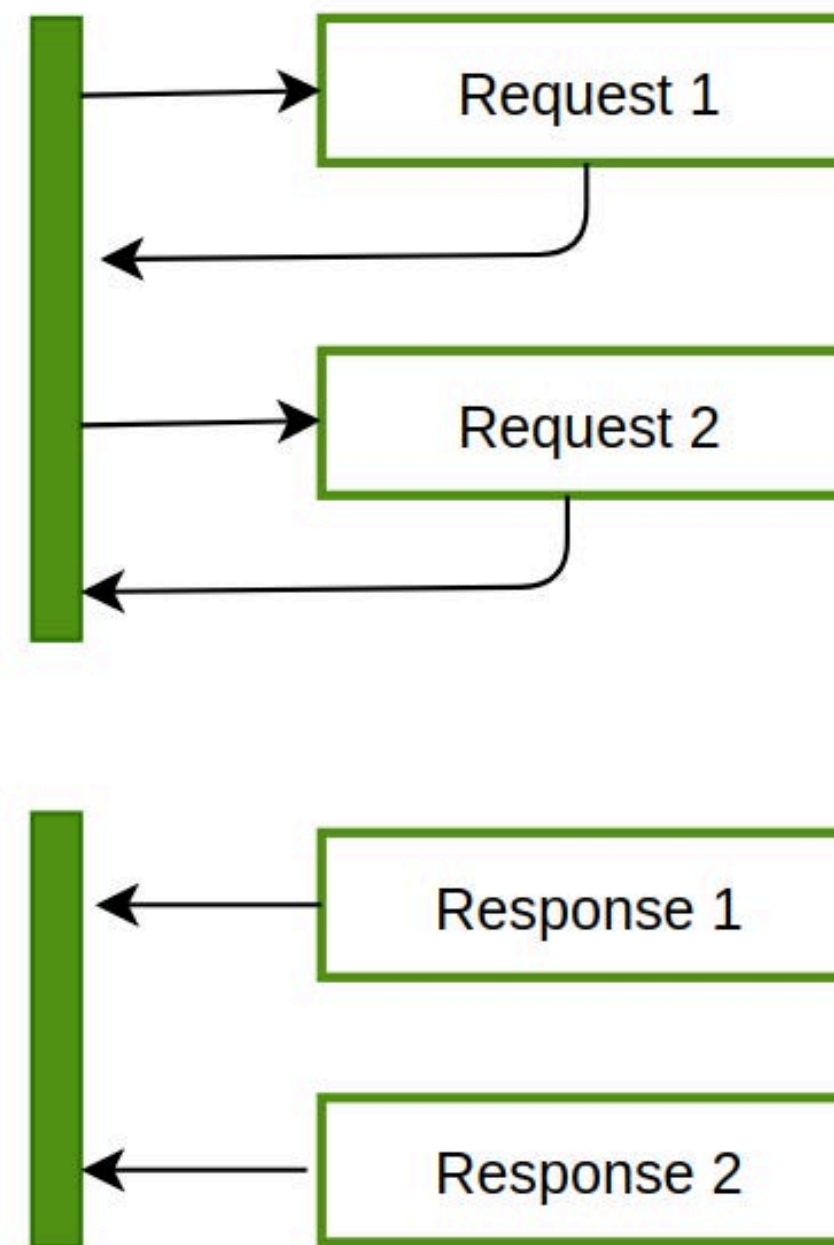
Loop de Eventos: é o gerenciador que de fato executa as corrotinas. Ele aproveita o exato momento em que uma função pausa para, imediatamente, dar a vez para outra que já esteja pronta para rodar, mantendo o programa sempre em progresso.



Synchronous



Asynchronous



import asyncio

```
async def task():  
    print("Fala galera! ")  
    await asyncio.sleep(1)  
    print("Aulinha de SD hoje... ")
```

Em Python, **async/await** é a sintaxe moderna utilizada para escrever código concorrente

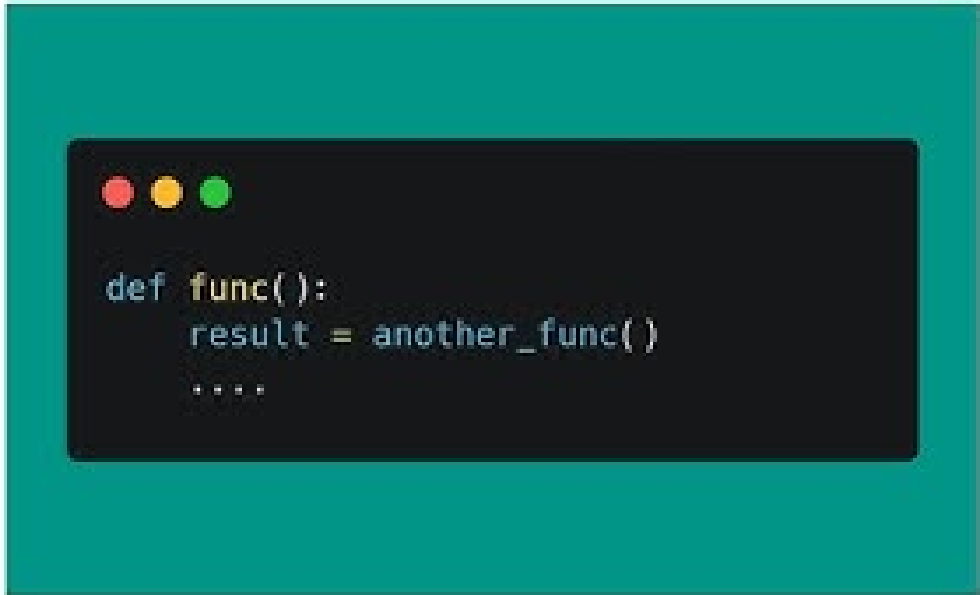
A palavra-chave **async** antes de **def** transforma uma função normal em uma corrotina. Uma corrotina é como uma "receita" para uma tarefa que pode ser pausada.

A palavra-chave **await** só pode ser usada dentro de uma função `async def`, ela faz duas coisas importantes:


- Pausa a execução da corrotina atual;
- Devolve o controle ao gerenciador de eventos.

Por que a necessidade de usar funções assíncronas?

Para evitar que o programa "trave" enquanto espera por operações lentas, como uma resposta de rede ou acesso a um arquivo.



```
def func():  
    result = another_func()  
    ....
```



```
async def func():  
    result = await another_func()  
    ...
```


Vantagens do uso de Corrotinas

- Eficiência extrema para tarefas de I/O (Entrada/Saída)
- Baixo consumo de recursos
- Código mais legível e limpo
- Alta escalabilidade e responsividade

Limitações no uso de Corrotinas

- Ineficaz para tarefas ligadas à CPU (CPU-Bound)
- Não há paralelismo real: As tarefas se alternam, mas não são executadas ao mesmo tempo
- Dependências de bibliotecas compatíveis: Para que os benefícios funcionem, todo o fluxo de I/O precisa ser assíncrono.

Obrigado