Async IO

Synchronous Vs Asynchronous code

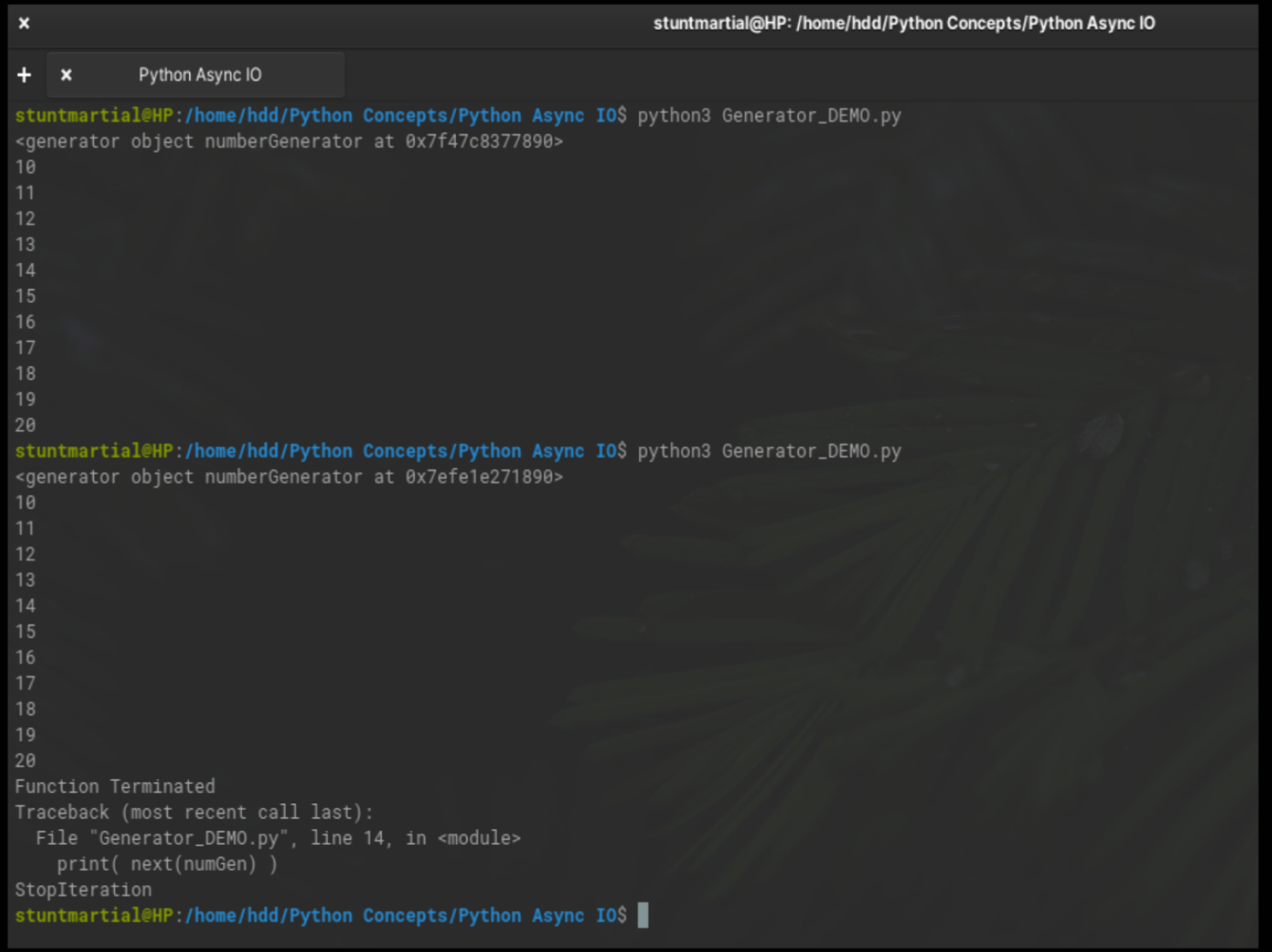
```
def add(number1, number2):
    return number1 + number2

number1 = random.randint(10, 30)
number2 = random.randint(40, 50)
sum_ = add(number1, number2)
print(sum_)
```

This code is synchronous, ie. one function or operation must be finished or executed completely before starting another operation.

```
def numberGenerator(lowerLim, upperLim):
    for number in range(lowerLim, upperLim+1):
        yield number
    print("Function Terminated")

numGen = numberGenerator(100, 200)
next(numGen) → 100
next(numGen) → 101
```

This code is Asynchronous. cuz a generator function yields (or returns) without even completing entire function. Context switch occurs, at application level. so, this is asynchronous.

Lets quickly glance through Generator_DEMO.py



→ If generator is run 10 times

→ If generator is run 11th time, of course we are outside loop. So "Function Terminated" is printed. Since there is no yield in the 11th time, stop iteration is raised.

Async IO in python

Asynchrony may be achieved by multiprocessing or multithreading. In either of the cases, when a process / thread is suspended like waiting for I/O, querying for DB, we context switch to a new process / thread.

Async IO module in python is however designed for a slightly different use case. Why not execute other functions in the meantime in the same thread ?

The most common use-case is in network request response cycle. If for a request we need to do a time consuming operation like

i) Running a forward propagation through a NN model ; or
ii) Querying a DB ; or
iii) Perform some lengthy computation

our server (or current thread) is incapable of accepting other requests in the meantime. Async IO focuses on this use case and makes a thread run other portions of itself in the meantime.

some aspects in AsyncIO

coroutines → coroutines are asynchronous functions, or methods.

async → we create coroutines by just mentioning async before defining a function/ method.

Ex:
```
async def my_coroutine (parameter1, parameter2) :
    ....
    ....
    ....
    result = parameter1 + parameter2
    return result
```

await → (we will see what this means in a moment. Just mentioning here cuz async and await are used together in coroutines)

Eventloop → Eventloop is a low level python aspect, that has the responsibility of running coroutines.

Ex:

```
import asyncio

async def addnumbers(num1 = 100, num2 = 200):
    result = num1 + num2
    return result

asyncio.run(addnumbers())  # runs addnumbers coroutine
```

But where is asynchrony? Everything is synchronous; moreover we introduces some complicated "eventloop" that seems to do the same thing ie run a function.

Right, we shall now deep dive in asynchrony, and will reveal the await keyword.

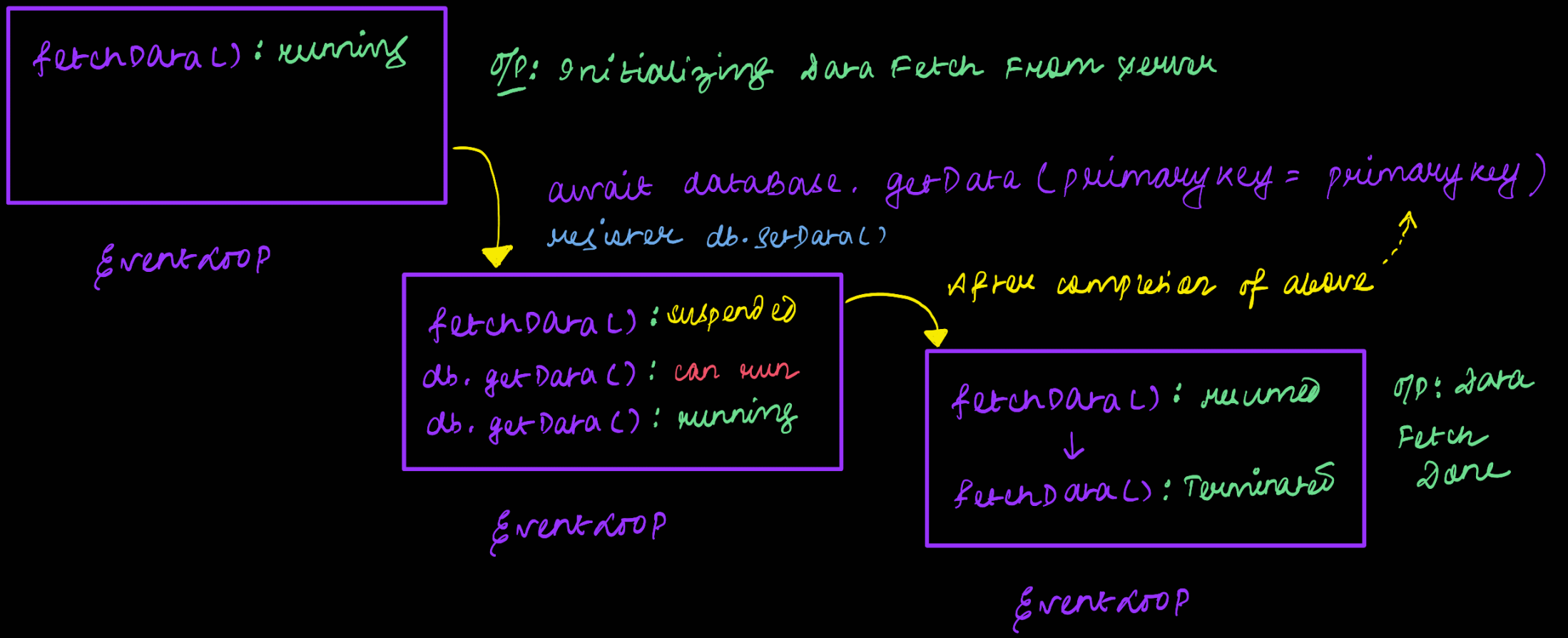
```
import asyncio

async def fetchData(primaryKey = 1):
    print("Initializing data fetch from server")
    await database.getData(primaryKey = primaryKey)
    print("Data Fetch Done")

asyncio.run(fetchData())
```

await keyword basically says "stop execution of current coroutine till `database.getData()` coroutine has completed its execution."

Here say, database.getData(primaryKey = primaryKey) takes 2s.

Lets understand the execution.

asyncio.run(fetchData()) : Creates an eventloop and says register and run fetchData() coroutine. Eventloop basically handles which coroutine to run at any time.



Yes till now, no asynchrony. fetchData() and database.getData() runs synchronously, cuz fetchData() depends on db.getData() for its execution.

But, let's understand some key points.

i) We cannot run a coroutine by just calling it like a regular function.

why? Because, we do not know from exactly where we need to start executing the coroutine. So, this job is given to the event loop.

ii) If we recall, calling a generator function, returned a generator object and we ran the generator object using next(). Similarly,

calling coroutine function returns coroutine object

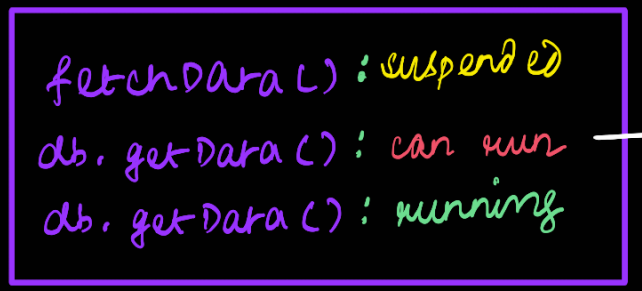
using,

asyncio.run(coroutine object) → registers coroutine object in event loop & starts running it

await coroutine object → registers coroutine object in event loop

iii) When does an event loop run a coroutine?

Let's say an event loop has multiple coroutines in suspended state and may be 4 of them may be resumed. Which one should get resumed? Probably event loop follows some strategy (like FCFS) to break ties in these scenarios.



→ only 1 coroutine can be run. So, run

iv) Why use await, when we can use asyncio.run() ?

asyncio.run() creates an event loop. Each thread can have only one event loop.

Q) Okay, then why not use await only, cuz the extra thing asyncio.run() does is it starts running the coroutine?

await can only be written inside of an async function. We can't obviously say await to a synchronous function.

But Python code in itself is synchronous. So, to start off this chain of coroutines, we cannot use await.

asyncio.run() → entry point of asynchronous operations in synchronous python code.

await → wait execution of current coroutine until execution of mentioned coroutine is done; in this way, mentioned coroutine is pushed in event loop.

Till now, we have been ignoring 2 things :

1) The real asynchrony : we have been using asyncio.run and await to run 1 single chain of coroutines, that runs synchronously

```
async def coroutine():
    print("Hello")

async def coroutine1():
    .......
    .......
    await coroutine()
    ....
    ....

async def coroutine2():
    ....
    ....
    await coroutine1()
    ....
    ....

asyncio.run(coroutine2())
```

Execution FLOW :

coroutine 2
↓
coroutine 1
↓
coroutine
↓
coroutine 1
↓
coroutine 2

During print() or more generally during I/O, CB & very current thread remains idle & thus one single chain of coroutines are not properly asynchronous.

ii) Handling returns from coroutines

These aspects are carried under 2 concepts :

i) Asynchrony → Tasks

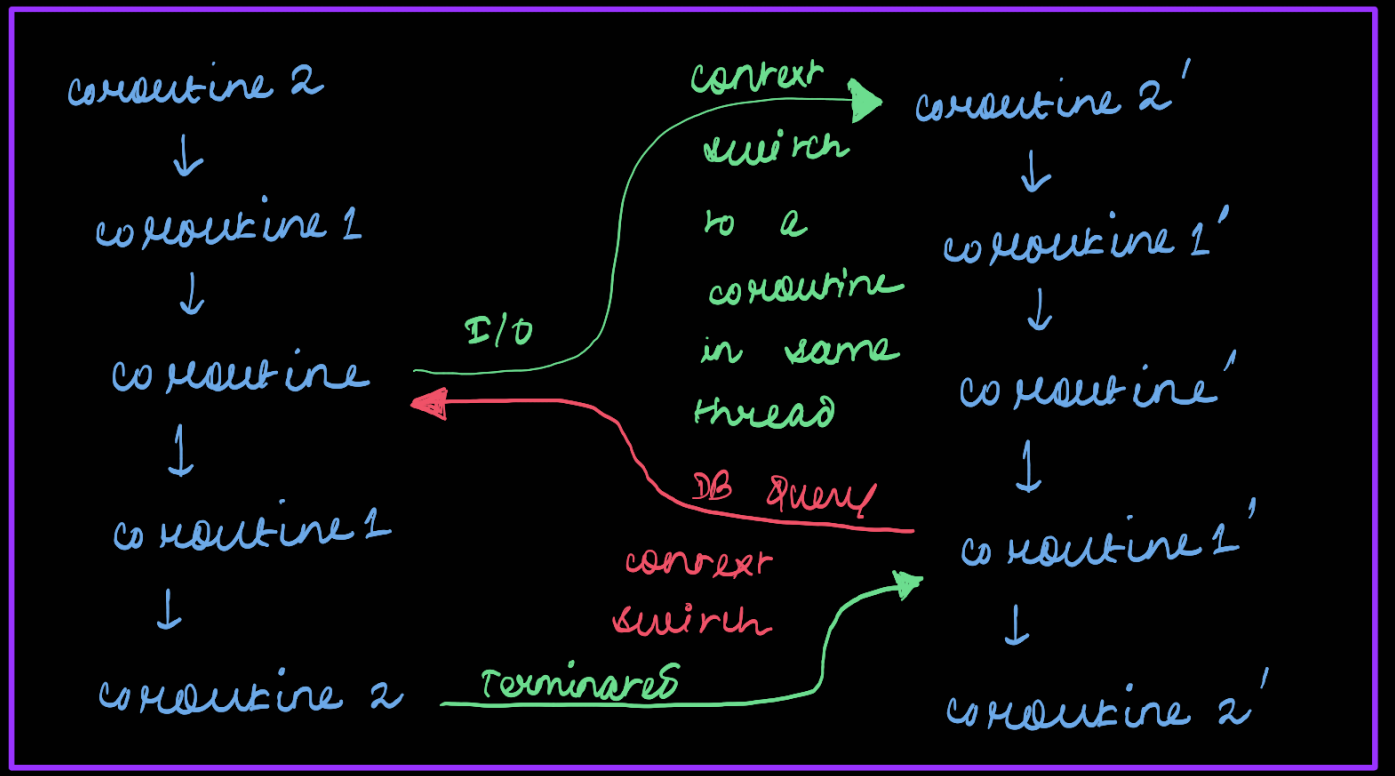
ii) Handling returns from coroutines → Futures

Lets focus on the concepts first and then we will properly understand the execution flow via an example.

Tasks

If we have a single chain of coroutines, they are not sort of properly asynchronous, cuz our current thread still remains idle on I/O operations, DB query etc.

But what if we had multiple chains of coroutines in our EventLoop.



whenever there are multiple chains of coroutines registered in an eventloop, if await occurs, and eventloop if finds another chain of coroutines, it has for the opportunity to shift execution to a coroutine in some other chain, cuz this chain is blocked.

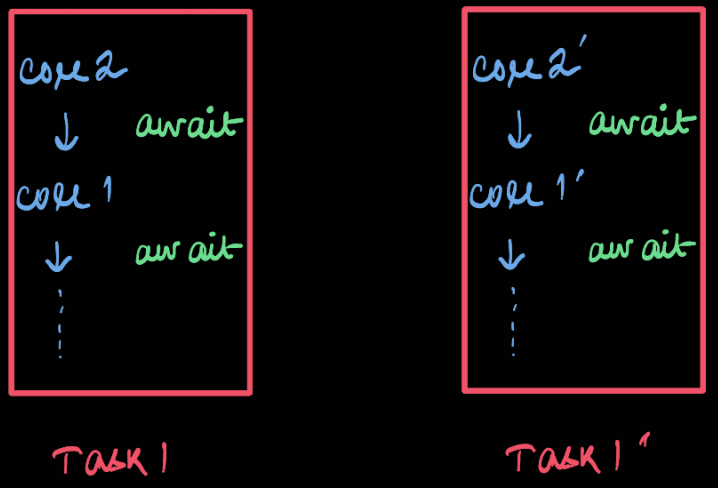
But now can we register multiple chains in our eventloop ?

Till now, we have learnt to register the first coroutine of a chain using asyncio.run() and then use await to expand that chain (intuitively that is only happening).

Formally, one chain of coroutine is called Task.

Tasks basically define the sequence in which coroutines execute.

We can encapsulate each of the chains into tasks.




```
task1 = asyncio.create_task(cor2())
task2 = asyncio.create_task(cor2')
```

create_task basically creates task and registers that into event loop.

Futures

Futures are low level python aspects. What we do need to know about is Futures represent the ultimate result of a coroutine.

Where exactly does a coroutine function return after its complete execution?

Obviously either at asyncio.run() or at await. If we write

```
async def coroutine2():
    .....
    * await coroutine1()
    ....
    ...
* asyncio.run(coroutine2())
```

```
var1 = await coroutine1()  or

var2 = asyncio.run(coroutine2())
```

var1 and var2 can store whatever is returned by the coroutines. Hence, we can think var1 and var2 as "Future".

Key points while using AsyncIO

i) async and await are python keywords, and not part of asyncio package.

ii) coroutines, tasks and futures can be awaited and hence are collectively called as Awaitables or Awaitable Objects.

iii) We need not worry about Futures, as Futures are low level awaitable objects and we will rarely use them explicitely at an application level. All we need to know is how and where to get the returned value or object from coroutine.

Ex: if coroutine1() returns 24 after its complete execution,

```
var1 = 24
```

iv) General pattern of writing Asynchronous Program while using asyncio

Create a coroutine, where we will register all tasks in event loop and use that coroutine as entry point to our asynchronous code.

Ex:

```
async def registerTasks():
    task1 = asyncio.create_task(coro1())
    task2 = asyncio.create_task(coro2())

    await task1
    await task2

asyncio.run(registerTasks())
```

No context switch occurs here, cuz registerTask() did not give up its execution. Only task1 and task2 got registered in event loop etc

* We must await registerTask() till all tasks are complete. Otherwise control will come out of registerTask() back to synchronous code. Hence, event loop won't get executed for task1 and task2.