Event Loop

This lesson introduces the concept of the event loop and the central role it plays in asynchronous programming models.

Event Loop

The event loop is a programming construct that waits for events to happen and then dispatches them to an event handler. An event can be a user clicking on a UI button or a process initiating a file download. At the core of asynchronous programming, sits the event loop. The concept isn't novel to Python. In fact, many programming languages enable asynchronous programming with event loops. In Python, event loops run asynchronous tasks and callbacks, perform network IO operations, run subprocesses and delegate costly function calls to pool of threads.

One of the most common use cases you'll find in the wild is of webservers implemented using asynchronous design. A webserver waits for an HTTP request to arrive and returns the matching resource. Folks familiar with JavaScript would recall NodeJS works on the same principle. It is a webserver that runs an event loop to receive web requests in a single thread. Contrast that to webservers which create a new thread or worse fork a new process, to handle each web request. In some benchmarks, the asynchronous event loop based webservers outperformed multithreaded ones, which may seem counterintuitive.

Rationale for Event Loops

In order to truly appreciate event loops, we'll present Ryan Dahl's motivation for creating NodeJS which also runs an event loop. Ryan classifies disk and network I/O operations as blocking operations and presents (https://www.youtube.com/watch?v=ztspvPYybIY) the following table to put the latency for various operations in perspective.

Device	CPU Cycles	Humanified
L1 cache	3	1 seconds
L2 cache	14	4.6 seconds
RAM	250	83 seconds
Disk	41000000	158 days
Network	240000000	2.5 years

The third column *humanizes* the CPU cycles by assuming if 3 CPU cycles equaled one second then a network I/O would feel equivalent of 2.5 years. This may dawn upon you how slow can disk and network I/O be in comparison to access times for other devices. The long waiting times suggest that if these blocking calls are synchronous then we end up wasting a lot of CPU cycles. The CPU can be better utilized especially in environments with a lot of I/O.

Synchronous blocking I/O calls can be made non-blocking by either one of the following two methods:

- Use threads to make blocking calls

- Use threads to make blocking calls.

- Convert blocking calls to nonblocking asynchronous calls.

Threads don't come cheap. Creating, maintaining and tearing down threads takes CPU cycles in addition to memory. In fact, this difference becomes more visible in webservers which use threads to handle HTTP web requests vs which use an event loop. Apache is an example of the former and NGINX of the latter. NGINX outshines Apache in memory usage under high load.

Taking the discussion back to Python, you'll rarely need to access the event loop directly unless you are working on low level libraries or functionality. The below code snippet retrieves the event loop object and prints it.

```
import asyncio

# Retrieving the event loop in asyncio
loop = asyncio.get_event_loop()
print(str(loop))
```

Back
... continued

Next
... continued

✓ Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/event-loop__asyncio__python-concurrency-for-senior-engineering-interviews)