

Ye Ole' Table-o-Contents

- 1. Traditional vs. `asyncio`-based source comparison**
- 2. *Flow-diagram*: Traditional sourcecode**
- 3. *Flow-diagram*: `asyncio`-based sourcecode**

Source Code:
NOT Asynchronous (*blocking*)

```
import time

def blocking_repeat_message(message, interval_seconds, max_iters=10):
    iters = 0
    while True:
        print(message)
        if iters >= max_iters:
            break
        iters += 1
        time.sleep(interval_seconds)

def main():
    # Params
    message_A = "xXxXxXxXxXxXxXxXxXxXxXxX"
    interval_A = 0.5
    message_B = "I LOVE"
    interval_B = 1
    message_C = "EXPLOSIONS!"
    interval_C = 1.5

    # Module-logic
    beginning_time = time.time()

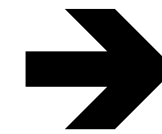
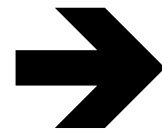
    blocking_repeat_message(message_A, interval_A)
    blocking_repeat_message(message_B, interval_B)
    blocking_repeat_message(message_C, interval_C)

    ending_time = time.time()

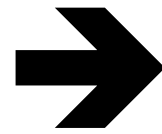
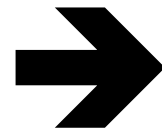
    return (beginning_time, ending_time)

if __name__ == '__main__':
    begin_time, end_time = main()
    duration = end_time - begin_time
    print("Execution time in seconds: {}".format(duration))
```

AAA OLD-SCHOOL AAA



VS.

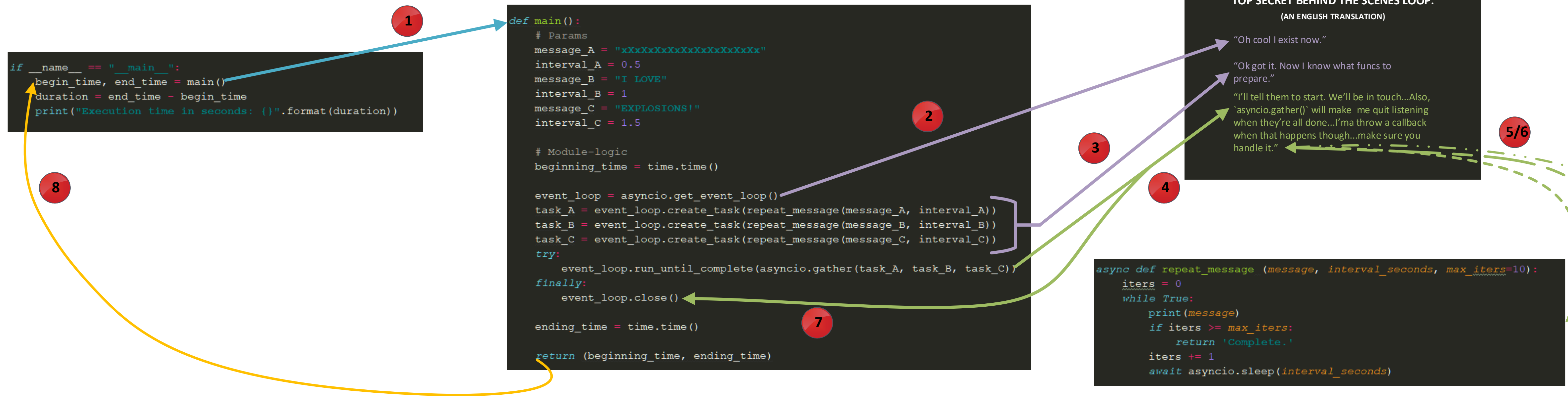


Source Code: Asynchronous

[illegible]

^^^ NEW-SCHOOL ^^^

Flow Model: Asynchronous



1 **main() is called.**

2 An "Event Loop" is created. This structure will mediate pretty much everything. Its high-level function is to TAKE control of execution flow FROM main(), and GIVE control TO our non-blocking-functions (aka `async def` functions.)

3 Here we tell the loop what functions it's supposed to oversee.

4 Note that this arrow goes both ways. Right now we're talking about going forward from `main()` to the loop, & not the other way around. We tell the loop that it's time for it to do its thing.

5 The loop starts all 3 non-blocking functions at the same time (Note: technically this isn't true, but it's helpful to conceptualize this way. See "queueing" for more details.) The 3 functions are generators, meaning they each have their own notion of state. Additionally, these functions aren't a**holes; they `await` other functions, preventing each function from getting in each other's way.

6 Each function does its thing, When one of the functions iterates, it mashes the proverbial "submit" button. The "submit" button notifies the loop of the function's successful completion of a single iteration.

7 The loop hands answers from each function to main() in the order they were received.

8 **main() completes.**

```
async def repeat_message (message, interval_seconds, max_iters=10):
    iters = 0
    while True:
        print(message)
        if iters >= max_iters:
            return 'Complete.'
        iters += 1
        await asyncio.sleep(interval_seconds)
```

```
async def repeat_message (message, interval_seconds, max_iters=10):
    iters = 0
    while True:
        print(message)
        if iters >= max_iters:
            return 'Complete.'
        iters += 1
        await asyncio.sleep(interval_seconds)
```

```
async def repeat_message (message, interval_seconds, max_iters=10):
    iters = 0
    while True:
        print(message)
        if iters >= max_iters:
            return 'Complete.'
        iters += 1
        await asyncio.sleep(interval_seconds)
```