Ye Ole' Table-o-Contents

Source Code: **NOT Asynchronous** *(blocking)*

```python
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 = "xXxXxXxXxXxXxXxXxXxXx"
    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))
```

^^^ **OLD-SCHOOL** ^^^

vs.

Source Code: **Asynchronous**

```python
import asyncio
import time


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)

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

    # Module-logic
    beginning_time = time.time()

    event_loop = asyncio.get_event_loop()
    task_A = event_loop.create_task(repeat_message(message_A, interval_A))
    task_B = event_loop.create_task(repeat_message(message_B, interval_B))
    task_C = event_loop.create_task(repeat_message(message_C, interval_C))
    try:
        event_loop.run_until_complete(asyncio.gather(task_A, task_B, task_C))
    finally:
        event_loop.close()

    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))
```

^^^ **NEW-SCHOOL** ^^^

Flow Model:
NOT Asynchronous

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

```python
def main():
    # Params
    message_A = "xXxXxXxXxXxXxXxXxXxXx"
    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)
```

```python
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)
```

```python
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)
```

```python
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)
```

1. main() is called.
2. We start calling functions.
3. One is run at a time. When each finishes, we move on to the next.
4. Functions are complete. Going back to main().
5. main() completes.

Flow Model: Asynchronous

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

1

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

    # Module-logic
    beginning_time = time.time()

    event_loop = asyncio.get_event_loop()
    task_A = event_loop.create_task(repeat_message(message_A, interval_A))
    task_B = event_loop.create_task(repeat_message(message_B, interval_B))
    task_C = event_loop.create_task(repeat_message(message_C, interval_C))
    try:
        event_loop.run_until_complete(asyncio.gather(task_A, task_B, task_C))
    finally:
        event_loop.close()

    ending_time = time.time()

    return (beginning_time, ending_time)
```

2

TOP SECRET BEHIND THE SCENES LOOP:

(AN ENGLISH TRANSLATION)

"Oh cool I exist now."

"Ok got it. Now I know what funcs to prepare."

"I'll tell them to start. We'll be in touch...Also, `asyncio.gather()` will make me quit listening when they're all done...I'ma throw a callback when that happens though...make sure you handle it."

3

4

5/6

```python
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)
```

```python
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)
```

```python
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)
```

7

8

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 it's 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.**