Execute Single and Multiple Tasks

This lesson will teach you cooperative multitasking to execute single and multiple tasks using python.

WE'LL COVER THE FOLLOWING

In **asynchronous programming**, the execution of a function is usually non-blocking. In other words, each time you call a function it returns immediately. However, that function does not necessarily get executed right away. Instead, there is usually a mechanism (called the "scheduler") which is responsible for the future execution of the function.

The problem with asynchronous programming is that a program may end before any asynchronous function starts. A common solution for this is for asynchronous functions to return "futures" or "promises". These are objects that represent the state of execution of an async function. Finally, asynchronous programming frameworks typically have mechanisms to block or wait for those async functions to end based on those "future" objects.

Co-operative Multitasking

Since Python 3.6, the `asyncio` module combined with the `async` and `await` keyword allows us to implement what is called *co-operative multitasking programs*. In this type of programming, a coroutine function voluntarily yields control to another coroutine function when idle or when waiting for some input.

Asynchronous functions are declared with `async def`.

```
import asyncio
```

```
async def functionName():
  await asyncio.sleep(1)
  return
```

Execute a Single Task

To call an asynchronous function once, do the following:-

1. Create an event loop
2. Run async function and wait for completion
3. Close the loop

```
# Create an event loop
loop = asyncio.get_event_loop()

# Run async function and wait for completion
results = loop.run_until_complete(functionName())

# Close the loop
loop.close()
```

Consider the following asynchronous function that squares a number and sleeps for one second before returning. (Ignore the `await` keyword for now.)

```
import asyncio

async def square(x):
    print('Square', x)
    await asyncio.sleep(1)
    print('End square', x)
    return x * x

# Create event loop
loop = asyncio.get_event_loop()

# Run async function and wait for completion
results = loop.run_until_complete(square(1))
print(results)

# Close the loop
loop.close()
```

The event loop for asynchronous Programming is, among other things, the Python mechanism that schedules the execution of asynchronous functions. We use the loop to run the function until completion. This is a synchronizing mechanism that makes sure the next print statement doesn't execute until we have some results.

have some results.

Execute Multiple Tasks

The previous example is not a good example of asynchronous programming because we don't need that much complexity to execute only one function. However, imagine that you would need to execute the `square(x)` function three times, like this:

```
square(1)
square(2)
square(3)
```

To call an asynchronous function to execute multiple tasks, do the following:- 1.Create an event loop 2.Run async function and wait for completion

```
# Create event loop
loop = asyncio.get_event_loop()

# Run async function and wait for completion
results = loop.run_until_complete(asyncio.gather(
functionName()
functionName()
.
.
.
functionName())

# Close the loop
loop.close()
```

Since the `square()` function has a sleep function inside, the total execution time of this program would be 3 seconds. However, given that the computer is going to be idle for a full second each time the function is executed, why can't we start the next call while the previous is sleeping? Here's how we do it:

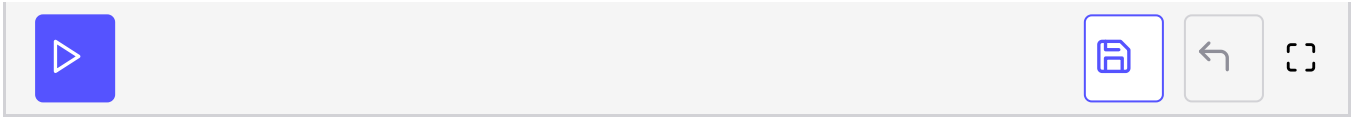
```
import asyncio

async def square(x):
    print('Square', x)
    await asyncio.sleep(1)
    print('End square', x)
    return x * x

# Create event loop
loop = asyncio.get_event_loop()
```

```
# Run async function and wait for completion
results = loop.run_until_complete(asyncio.gather(
    square(1),
    square(2),
    square(3)
))
print(results)

# Close the loop
loop.close()
```



Basically, we use `asyncio.gather(*tasks)` to inform the loop to wait for all tasks to finish. Since the coroutines will start at almost the same time, the program will run for only 1 second. Asyncio **gather()** won't necessarily run the coroutines by order, although it will return an ordered list of results.

Sometimes results may be needed as soon as they are available. For that, we can use a second coroutine that deals with each result using `asyncio.as_completed()`:

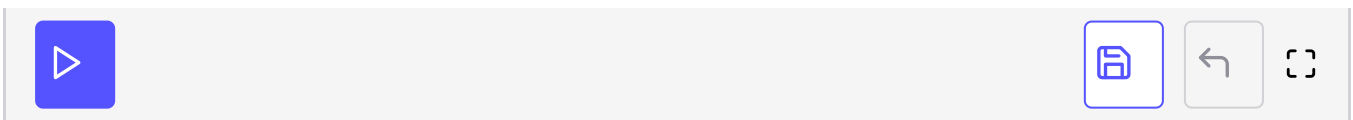
```
import asyncio

async def square(x):
    print('Square', x)
    await asyncio.sleep(1)
    print('End square', x)
    return x * x

# Create event loop
loop = asyncio.get_event_loop()

async def when_done(tasks):
    for res in asyncio.as_completed(tasks):
        print('Result:', await res)

loop = asyncio.get_event_loop()
loop.run_until_complete(when_done([
    square(1),
    square(2),
    square(3)
]))
```



Finally, async coroutines can call **other async coroutine functions** with the **await** keyword:

```
import asyncio

async def compute_square(x):
    await asyncio.sleep(1)
    return x * x

async def square(x):
    print('Square', x)
    res = await compute_square(x)
    print('End square', x)
    return res

# Create event loop
loop = asyncio.get_event_loop()

async def when_done(tasks):
    for res in asyncio.as_completed(tasks):
        print('Result:', await res)

loop = asyncio.get_event_loop()
loop.run_until_complete(when_done([
    square(1),
    square(2),
    square(3)
]))
```

Now that you have a clear concept of asynchronous programming, let's test your knowledge in the upcoming exercises.