Python async/await

Summary: in this tutorial, you will learn about Python coroutines and how to use the Python `async` and `await` keywords to create and pause coroutines.

Introduction to Python coroutines

A coroutine is a regular function with the ability to pause its execution when encountering an operation that may take a while to complete.

When the long-running operation completes, you can resume the paused coroutine and execute the remaining code in that coroutine.

While the coroutine is waiting for the long-running operation, you can run other code. By doing this, you can run the program asynchronously to improve its performance.

To create and pause a coroutine, you use the Python `async` and `await` keywords:

- The `async` keyword creates a coroutine.
- The `await` keyword pauses a coroutine.

Defining a coroutine with Python async keyword

The following defines a simple function that returns the square number of an integer:

```
def square(number: int) -> int:
    return number*number
```

And you can pass an integer to the `square()` function to get its square number:

```
def square(number: int) -> int:
    return number*number


result = square(10)
print(result)
```

Output:

```
100
```

When you add the `async` keyword to the function, the function becomes a coroutine:

```
async def square(number: int) -> int:
    return number*number
```

A calling coroutine returns a coroutine object that will be run later. For example:

```
async def square(number: int) -> int:
    return number*number


result = square(10)
print(result)
```

Output:

```
<coroutine object square at 0x00000185C31E7D80>
sys:1: RuntimeWarning: coroutine 'square' was neve
```

In this example, we call the `square()` coroutine, assign the returned value to the `result` variable, and print it out.

When you call a coroutine, Python doesn't execute the code inside the coroutine immediately. Instead, it returns a coroutine object.

The second line in the output also shows an error message indicating that the coroutine was never awaited. More on this in the following `await` section:

```
sys:1: RuntimeWarning: coroutine 'square' was never
```

To run a coroutine, you need to execute it on an event loop. Prior to Python 3.7, you have to manually create an event loop to execute coroutines and close the event loop.

However, since version 3.7, the `asyncio` library added some functions that simplify the event loop management.

For example, you can use the `asyncio.run()` function to automatically create an event loop, run a coroutine, and close it.

The following uses the `asyncio.run()` function to execute the `square()` coroutine and get the result:

```
import asyncio


async def square(number: int) -> int:
    return number*number

result = asyncio.run(square(10))
print(result)
```

Output:

```
100
```

It's important to note that the `asyncio.run()` is designed to be the main entry point of an `asyncio` program.

Also, the `asyncio.run()` function only executes one coroutine which may call other coroutines and functions in the program.

Pausing a coroutine with Python await keyword

The `await` keyword pauses the execution of a coroutine. The `await` keyword is followed by a call to a coroutine like this:

```
result = await my_coroutine()
```

The `await` keyword causes the `my_coroutine()` to execute, waits for the code to be completed, and returns a result.

It's important to note that `await` keyword is only valid inside a coroutine. In other words, you must use the `await` keyword inside a coroutine.

This is the reason why you saw an error message in the above example that uses the `await` keyword outside of a coroutine.

The following example shows how to use the `await` keyword to pause a coroutine:

```
import asyncio


async def square(number: int) -> int:
    return number*number


async def main() -> None:
    x = await square(10)
    print(f'x={x}')

    y = await square(5)
    print(f'y={y}')
```

```
    print(f'total={x+y}')

if __name__ == '__main__':
    asyncio.run(main())
```

Output:

```
x=100
y=25
total=125
```

How it works. (we'll focus on the `main()` function):

First, call the `square()` coroutine using the `await` keyword. The `await` keyword will pause the execution of the `main()` coroutine, wait for the `square()` coroutine to complete, and return the result:

```
x = await square(10)
print(f'x={x}')
```

Second, call the `square()` coroutine a second time using the `await` keyword:

```
y = await square(5)
print(f'y={y}')
```

Third, display the total:

```
print(f'total={x+y}')
```

The following statement uses the `run()` function to execute the `main()` coroutine and manage the event loop:

```
asyncio.run(main())
```

So far, our program executes like a synchronous program. It doesn't reveal the power of the asynchronous

programming model.

Summary

- A coroutine is a regular function with the power of pausing a long-running operation, waiting for the result, and resuming from the paused point.
- Use `async` keyword to define a coroutine.
- Use `await` keyword to pause a coroutine.
- Use `asyncio.run()` function to automatically execute a coroutine on an event loop and manage an event loop.