The following scripts are written to demonstrate the use of Python Coroutines and Tasks.

Coroutines declared with async/await syntax is the preferred way of writing asyncio applications.

To actually run a coroutine, asyncio provides three main mechanisms:

> The asyncio.run() function to run the top-level entry point “main()” function.

> Awaiting on a coroutine: An object is an awaitable object if it can be used in an await expression. Many asyncio APIs are designed to accept awaitables.

> The asyncio.create\_task() function to run coroutines concurrently as asyncio Tasks.

There are three main types of awaitable objects: coroutines, Tasks, and Futures.

Coroutines: Python coroutines are awaitables and therefore can be awaited from other coroutines.

Tasks: Tasks are used to schedule coroutines concurrently. When a coroutine is wrapped into a Task with functions like asyncio.create\_task() the coroutine is automatically scheduled to run soon:

Futures: A Future is a special low-level awaitable object that represents an eventual result of an asynchronous operation. When a Future object is awaited it means that the coroutine will wait until the Future is resolved in some other place. Future objects in asyncio are needed to allow callback-based code to be used with async/await. Normally there is no need to create Future objects at the application level code. Future objects, sometimes exposed by libraries and some asyncio APIs, can be awaited:

Compiled and presented by Vakindu Philliam.