asyncio - Asynchronous I/O:

library to write concurrent code using the async/await syntax.

used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc.

asyncio provides a set of high-level APIs to:

- run Python coroutines concurrently and have full control over their execution;
- perform network IO and IPC;
- control subprocesses;
- distribute tasks via queues;
- synchronize concurrent code;

There are low-level APIs for library and framework developers to:

- create and manage event loops, which provide asynchronous APIs for networking, running subprocesses, handling OS signals, etc;
- implement efficient protocols using transports;
- bridge callback-based libraries and code with async/await syntax.

Coroutines and tasks

Coroutines declared with the async/await syntax is the preferred way of writing asyncio applications. For example, the following snippet of code (requires Python 3.7+) prints "hello", waits 1 second, and then prints "world":

```
>>> import asyncio

>>> async def main():

...     print('hello')

...     await asyncio.sleep(1)

...     print('world')

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

Output:

```
Hello

world
```

- To actually run a coroutine, asyncio provides three main mechanisms: The asyncio.run() function to run the top-level entry point "main()" function (see the above example.) :
- Awaiting on a coroutine. The following snippet of code will print "hello" after waiting for 1 second, and then print "world" after waiting for another 2 seconds:

```
>>>import asyncio

>>>import time

async def say_after(delay, what):

...    await asyncio.sleep(delay)

...    print(what)

async def main():

...    print(f"started at {time.strftime('%X')}")

...    await say_after(1, 'hello')

...    await say_after(2, 'world')

...    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

Expected output:

```
started at 17:13:52

hello

world

finished at 17:13:55
```

Awaitables

We say that an object is an awaitable object if it can be used in an await expression. Many asyncio APIs are designed to accept awaitables.

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:

```
import asyncio

async def nested():

    return 42

async def main():

# Nothing happens if we just call "nested()".

    # A coroutine object is created but not awaited,

    # so it *won't run at all*.

    nested()

    # Let's do it differently now and await it:

    print(await nested())  # will print "42".

asyncio.run(main())
```

Tasks

Tasks are used to schedule coroutines concurrently.

When a coroutine is wrapped into a Task with functions like aysncio.create_task() the coroutine is automatically scheduled to run soon:

```
>>> import asyncio

async def nested():

...       return42

async def (main):

    # Schedule nested() to run soon concurrently    # with "main()".

...       task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or    # can simply be awaited to wait until it
is complete:

...       await taskasyncio.run(main())
```

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:

```
async def main():

...    await function_that_returns_a_future_object()

   # this is also valid:

...      await asyncio.gather(      function_that_returns_a_future_object(),
some_python_coroutine()    )
```

Running an asyncio Program

asyncio.run(coro, *, debug=False)

Execute the coroutine coro and return the result.

This function runs the passed coroutine, taking care of managing the asyncio event loop, finalizing asynchronous generators, and closing the threadpool.

This function cannot be called when another asyncio event loop is running in the same thread.

If debug is True, the event loop will be run in debug mode.

This function always creates a new event loop and closes it at the end. It should be used as a main entry point for asyncio programs, and should ideally only be called once.

Example:

```
async def main():

  await asyncio.sleep(1)

   print('hello')

asyncio.run(main())
```

Creating a task

asyncio.create_task(coro, *, name=None)

Wrap the coro coroutine into a Task and schedule its execution. Return the Task object.

If name is not None, it is set as the name of the task using Task.set_name().

The task is executed in the loop returned by get_running_loop(), RuntimeError is raised if there is no running loop in current thread.

This function has been added in Python 3.7. Prior to Python 3.7, the low-level asyncio.ensure_future() function can be used instead:

```
async def coro():

...       task = asyncio.create_task(coro())

...       task = asyncio.ensure_future(coro())...
```