Getting started

The **asyncio** module was added to Python in version 3.4 as a provisional package. What that means is that it is possible that asyncio receives backwards incompatible changes or could even be removed in a future release of Python. According to the documentation asyncio "*provides infrastructure for writing single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives*“. This chapter is not meant to cover everything you can do with asyncio, however you will learn how to use the module and why it is useful.

If you need something like asyncio in an older version of Python, then you might want to take a look at Twisted or gevent.

Definitions

The asyncio module provides a framework that revolves around the *event loop*. An event loop basically waits for something to happen and then acts on the event. It is responsible for handling such things as I/O and system events. Asyncio actually has several loop implementations available to it. The module will default to the one most likely to be the most efficient for the operating system it is running under; however you can explicitly choose the event loop if you so desire. An event loop basically says “when event A happens, react with function B”.

Think of a server as it waits for someone to come along and ask for a resource, such as a web page. If the website isn’t very popular, the server will be idle for a long time. But when it does get a hit, then the server needs to react. This reaction is known as event handling. When a user loads the web page, the server will check for and call one or more event handlers. Once those event handlers are done, they need to give control back to the event loop. To do this in Python, asyncio uses *coroutines*.

A coroutine is a special function that can give up control to its caller without losing its state. A coroutine is a consumer and an extension of a generator. One of their big benefits over threads is that they don't use very much memory to execute. Note that when you call a coroutine function, it doesn't actually execute. Instead it will return a coroutine object that you can pass to the event loop to have it executed either immediately or later on.

One other term you will likely run across when you are using the asyncio module is *future*. A *future* is basically an object that represents the result of work that hasn't completed. Your event loop can watch future objects and wait for them to finish. When a future finishes, it is set to done. Asyncio also supports locks and semaphores.

The last piece of information I want to mention is the *Task*. A Task is a wrapper for a coroutine and a subclass of Future. You can even schedule a Task using the event loop.