# Asynchronous Concurrency Concepts

Most programs using other concurrency models are written linearly, and rely on the underlying threading or process management of the language runtime or operating system to change context as appropriate. An application based on `asyncio` requires the application code to explicitly handle context changes, and using the techniques for doing that correctly depends on understanding several inter-related concepts.

The framework provided by `asyncio` is centered on an *event loop*, a first class object responsible for efficiently handling I/O events, system events, and application context changes. Several loop implementations are provided, to take advantage of operating system capabilities efficiently. While a reasonable default is usually selected automatically, it is also possible to pick a particular event loop implementation from within the application. This is useful under Windows, for example, where some loop classes add support for external processes in a way that may trade some efficiencies in network I/O.

An application interacts with the event loop explicitly to register code to be run, and lets the event loop make the necessary calls into application code when resources are available. For example, a network server opens sockets and then registers them to be told when input events occur on them. The event loop alerts the server code when there is a new incoming connection or when there is data to read. Application code is expected to yield control again after a short period of time when no more work can be done in the current context. For example, if there is no more data to read from a socket the server should yield control back to the event loop.

The mechanism for yielding control back to the event loop depends on Python's *coroutines*, special functions that give up control to the caller without losing their state. Coroutines are similar to generator functions, and in fact generators can be used to implement coroutines in versions of Python earlier than 3.5 without native support for coroutine objects. `asyncio` also provides a class-based abstraction layer for *protocols* and *transports* for writing code using callbacks instead of writing coroutines directly. In both the class-based and coroutine models, explicitly changing context by re-entering the event loop takes the place of implicit context changes in Python's threading implementation.

A *future* is a data structure representing the result of work that has not been completed yet. The event loop can watch for a `Future` object to be set to done, allowing one part of an application to wait for another part to finish some work. Besides futures, `asyncio` includes other concurrency primitives such as locks and semaphores.

A `Task` is a subclass of `Future` that knows how to wrap and manage the execution for a coroutine. Tasks can be scheduled with the event loop to run when the resources they need are available, and to produce a result that can be consumed by other coroutines.

asyncio — Asynchronous I/O, event loop, and concurrency tools

Cooperative Multitasking with Coroutines

*This page was last updated 2016-12-18.*

**Navigation**

asyncio — Asynchronous I/O, event loop, and concurrency tools
Cooperative Multitasking with Coroutines

*The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.*

*Looking for examples for Python 2?*

**This Site**

- Module Index
- Index

**Other Writing**

- Blog
- The Python Standard Library By Example