Non-blocking Communication and Timeouts

By default, a `socket` is configured so that sending or receiving data *blocks*, stopping program execution until the socket is ready. Calls to `send()` wait for buffer space to be available for the outgoing data, and calls to `recv()` wait for the other program to send data that can be read. This form of I/O operation is easy to understand, but can lead to inefficient operation and even deadlocks, if both programs end up waiting for the other to send or receive data.

There are a few ways to work around this situation. One is to use a separate thread for communicating with each socket. This can introduce other complexities, though, with communication between the threads. Another option is to change the socket to not block at all, and return immediately if it is not ready to handle the operation. Use the `setblocking()` method to change the blocking flag for a socket. The default value is 1, which means to block. Passing a value of 0 turns off blocking. If the socket is has blocking turned off and it is not ready for the operation, then `socket.error` is raised.

A compromise solution is to set a timeout value for socket operations. Use `settimeout()` to change the timeout of a `socket` to a floating point value representing the number of seconds to block before deciding the socket is not ready for the operation. When the timeout expires, a `timeout` exception is raised.

This page was last updated 2016-12-18.
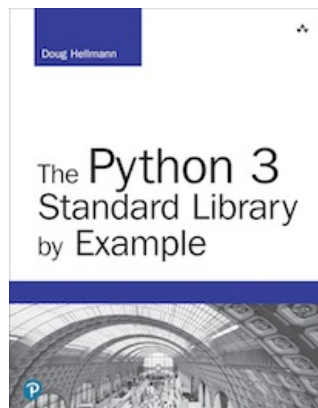
[Get the book](#)

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](#)?

Other Writing

✏️ Blog

📕 The Python Standard Library By Example