selectors — I/O Multiplexing Abstractions

Purpose: Provide platform-independent abstractions for I/O multiplexing based on the `select` module.

The `selectors` module provides a platform-independent abstraction layer on top of the platform-specific I/O monitoring functions in `select`.

Operating Model

The APIs in `selectors` are event-based, similar to `poll()` from `select`. There are several implementations and the module automatically sets the alias `DefaultSelector` to refer to the most efficient one for the current system configuration.

A selector object provides methods for specifying what events to look for on a socket, and then lets the caller wait for events in a platform-independent way. Registering interest in an event creates a `SelectorKey`, which holds the socket, information about the events of interest, and optional application data. The owner of the selector calls its `select()` method to learn about events. The return value is a sequence of key objects and a bitmask indicating what events have occurred. A program using a selector should repeatedly call `select()`, then handle the events appropriately.

Echo Server

The echo server example below uses the application data in the `SelectorKey` to register a callback function to be invoked on the new event. The main loop gets the callback from the key and passes the socket and event mask to it. As the server starts, it registers the `accept()` function to be called for read events on the main server socket. Accepting the connection produces a new socket, which is then registered with the `read()` function as a callback for read events.

```
# selectors_echo_server.py

import selectors
import socket

mysel = selectors.DefaultSelector()
keep_running = True


def read(connection, mask):
    "Callback for read events"
    global keep_running

    client_address = connection.getpeername()
    print('read({})'.format(client_address))
    data = connection.recv(1024)
    if data:
        # A readable client socket has data
        print('  received {!r}'.format(data))
        connection.sendall(data)
    else:
        # Interpret empty result as closed connection
        print('  closing')
        mysel.unregister(connection)
        connection.close()
        # Tell the main loop to stop
        keep_running = False


def accept(sock, mask):
    "Callback for new connections"
    new_connection, addr = sock.accept()
    print('accept({})'.format(addr))
    new_connection.setblocking(False)
    mysel.register(new_connection, selectors.EVENT_READ, read)


server_address = ('localhost', 10000)
```

```
print('starting up on {} port {}'.format(*server_address))
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(False)
server.bind(server_address)
server.listen(5)

mysel.register(server, selectors.EVENT_READ, accept)

while keep_running:
    print('waiting for I/O')
    for key, mask in mysel.select(timeout=1):
        callback = key.data
        callback(key.fileobj, mask)

print('shutting down')
mysel.close()
```

When read() receives no data from the socket, it interprets the read event as the other side of the connection being closed instead of sending data. It removes the socket from the selector and closes it. In order to avoid an infinite loop, this server also shuts itself down after it has finished communicating with a single client.

Echo Client

The echo client example below processes all of the I/O events in the main loop, instead of using callbacks. It sets up the selector to report read events on the socket, and to report when the socket is ready to send data. Because it is looking at two types of events, the client must check which occurred by examining the mask value. After all of its outgoing data has been sent, it changes the selector configuration to only report when there is data to read.

```
# selectors_echo_client.py

import selectors
import socket

mysel = selectors.DefaultSelector()
keep_running = True
outgoing = [
    b'It will be repeated.',
    b'This is the message.  ',
]
bytes_sent = 0
bytes_received = 0

# Connecting is a blocking operation, so call setblocking()
# after it returns.
server_address = ('localhost', 10000)
print('connecting to {} port {}'.format(*server_address))
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(server_address)
sock.setblocking(False)

# Set up the selector to watch for when the socket is ready
# to send data as well as when there is data to read.
mysel.register(
    sock,
    selectors.EVENT_READ | selectors.EVENT_WRITE,
)

while keep_running:
    print('waiting for I/O')
    for key, mask in mysel.select(timeout=1):
        connection = key.fileobj
        client_address = connection.getpeername()
        print('client({})'.format(client_address))

        if mask & selectors.EVENT_READ:
            print('  ready to read')
            data = connection.recv(1024)
            if data:
                # A readable client socket has data
                print('  received {!r}'.format(data))
                bytes_received += len(data)
            # Interpret empty result as closed connection,
            # and also close when we have received a copy
            # of all of the data sent.
            keep_running = not (
                data or
                (bytes_received and
                 (bytes_received == bytes_sent))
            )

        if mask & selectors.EVENT_WRITE:
            print('  ready to write')
            if not outgoing:
                # We are out of messages, so we no longer need to
                # write anything. Change our registration to let
                # us keep reading responses from the server.
                print('  switching to read-only')
                mysel.modify(sock, selectors.EVENT_READ)
            else:
                # Send the next message.
                next_msg = outgoing.pop()
                print('  sending {!r}'.format(next_msg))
                sock.sendall(next_msg)
                bytes_sent += len(next_msg)

print('shutting down')
mysel.unregister(connection)
connection.close()
mysel.close()
```

The client tracks the amount of data it has sent, and the amount it has received. When those values match and are non-zero, the client exits the processing loop and cleanly shuts down by removing the socket from the selector and closing both the socket and the selector.

Server and Client Together

The client and server should be run in separate terminal windows, so they can communicate with each other. The server output shows the incoming connection and data, as well as the response sent back to the client.

```
$ python3 source/selectors/selectors_echo_server.py
starting up on localhost port 10000
waiting for I/O
waiting for I/O
accept(('127.0.0.1', 59850))
waiting for I/O
read(('127.0.0.1', 59850))
  received b'This is the message.  It will be repeated.'
waiting for I/O
read(('127.0.0.1', 59850))
  closing
shutting down
```

The client output shows the outgoing message and the response from the server.

```
$ python3 source/selectors/selectors_echo_client.py
connecting to localhost port 10000
waiting for I/O
client(('127.0.0.1', 10000))
  ready to write
  sending b'This is the message.  '
waiting for I/O
client(('127.0.0.1', 10000))
  ready to write
  sending b'It will be repeated.'
waiting for I/O
client(('127.0.0.1', 10000))
  ready to write
  switching to read-only
waiting for I/O
```

```
client(('127.0.0.1', 10000))
  ready to read
  received b'This is the message.  It will be repeated.'
shutting down
```

See also

- Standard library documentation for selectors
- select – Lower-level APIs for handling I/O efficiently.

Non-blocking Communication and Timeouts   select — Wait for I/O Efficiently

This page was last updated 2016-12-31.

Navigation
Non-blocking Communication and Timeouts
select — Wait for I/O Efficiently

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?