```
# echo_client.py
#
# ICS 32 Winter 2014
# Code Example
#
# As we saw in lecture, the send() and recv() methods that send and
# receive bytes can be cumbersome when our goal is to send and receive
# text.  It should always be our goal to find a better tool for a job
# when there is one -- or to build ourselves a better tool when there
# isn't.  Fortunately, a better tool exists: we can ask a socket object
# give us a "pseudo-file object," an object that behaves just like a
# file object, except that it reads or writes to the socket's underlying
# streams, instead of to a file.  We end up needing two of them: one that
# reads from the socket's input stream and another that writes to its
# output stream.  Once we have them, we can treat our socket a lot like
# a text file, using techniques we already know.
#
# As in the previous example, no attempt has been made to find a clean
# design for our code; this is an example of how the underlying raw
# materials are used.  Once we've got these under our belts, a further
# example that focuses on design will follow.
import socket


# NOTE: You will need to replace this address with the correct address
# where the echo server is running.
ECHO_HOST = '127.0.0.1'
ECHO_PORT = 5151



def conduct_echo_conversation(echo_host: str, echo_port: int) -> None:
    # First, we'll create a socket, which we'll need in order to connect
    # to the echo server.
    echo_socket = socket.socket()

    # We'll also create two variables that we'll use to store the
    # pseudo-file objects (one to read from the socket, the other to
    # write to it).  We won't be able to create these until after the
    # socket is connected, but we'll need the variables to exist, so we
    # can refer to them in the "finally" block later.
    echo_socket_in = None
    echo_socket_out = None

    try:
        print('Connecting to echo server...')

        # Now we'll connect, using the echo_host and echo_port parameters
        # defined above.
        echo_socket.connect((echo_host, echo_port))

        # Next, now that we've connected successfully, we'll ask the
        # socket for the pseudo-file objects.  makefile('r') asks for
        # a pseudo-file object we can read from; it reads from the socket's
        # input stream.  makefile('w') asks for a pseudo-file object we
        # can write to; it writes to the socket's output stream.
        echo_socket_in = echo_socket.makefile('r')
        echo_socket_out = echo_socket.makefile('w')

        print('Connected successfully!')
        # Now that we're connected, we'll ask the user to type a message,
        # sending it to the echo server and printing out the echo server's
        # response.  This will continue until the user specifies an empty
        # message to send.
        while True:
            message_to_send = input('Message: ')

            if len(message_to_send) == 0:
                break
            else:
                # Sending the message to the echo server is much easier
                # than it was in the previous example, because we can
                # simply call the write() method on the pseudo-file object
                # that represents the socket's output.  This acts just like
                # writing to a text file, so we can just pass it a string,
                # and it'll handle the encoding to bytes for us.
                echo_socket_out.write(message_to_send + '\r\n')

                # One detail exists that we have to get right.  File objects
                # do what's called "buffering" (they store data in memory
                # temporarily and then write it once in a while when the
                # buffer runs out of space).  But when you're talking to
                # another program via a socket, it's usually important
                # that a message is sent *now*, because the other program
                # won't know what to do until it receives it.  So it becomes
                # important to tell the file object "Take whatever is in your
                # buffer and send it now!"  That's what "flushing" does.
                echo_socket_out.flush()

                # Receiving a line of text back from the echo server is
                # just like reading a line from a text file.  We call
                # readline() on the pseudo-file object representing the
                # socket's input.
                reply_message = echo_socket_in.readline()

                print('Reply: ' + reply_message)

    finally:
        # We'll end up in this "finally" block in one of two circumstances.
        #
        # (1) Everything went fine, meaning that we connected to the echo
        #     server, sent messages to it, and received messages from it
        #     with nothing going wrong.
        #
        # (2) A problem occurred while we tried to connect, or later when
        #     we were trying to send messages or receive replies.
        #
        # Either way, we want to be sure that the socket object and the
        # pseudo-file objects get closed if they were opened successfully.

        print('Closing connection')

        if echo_socket_in != None:
            echo_socket_in.close()

        if echo_socket_out != None:
            echo_socket_out.close()

        echo_socket.close()
    print('Goodbye!')



if __name__ == '__main__':
    conduct_echo_conversation()
```