```
# echo_client.py
#
# ICS 32 Winter 2014
# Code Example
#
# This is an example of the raw materials used to connect a Python
# program to another program using a socket.  There's been no attempt
# here to use good design principles -- we've written only one function,
# for example -- but this at least demonstrates how the underlying
# parts of the Python Standard Library work.  A subsequent example
# will focus on designing a better solution.


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()

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

        # Now we'll connect, using the echo_host and echo_port values
        # passed to us as parameters.  Note that the connect() method
        # on a socket takes a single parameter that indicates where to
        # connect; in this case, we're passing a single tuple containing
        # a host and a port, which is why there are two sets of parentheses
        # here.
        echo_socket.connect((echo_host, echo_port))

        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:
                # Before we can send the message to the server, we'll need
                # to encode it.  Encoding is the process by which you ask
                # a string object to turn itself into a corresponding sequence
                # of bytes (a 'bytes' object).  There are different encodings
                # that can be used, but we'll stick with a popularly-used
                # encoding called UTF-8.
                #
                # Note that we've added a '\r\n' to the end of our message,
                # which is the end-of-line sequence that is commonly used
                # by Internet servers.
```

```
                bytes_to_send = (message_to_send + '\r\n').encode(encoding='utf-
8')
                echo_socket.send(bytes_to_send)

                # Now we'll read the reply back from the server.  It will
                # come back as bytes, so we'll need to decode them back
                # into a string before we can print them out.  We also
                # strip spaces and the trailing newline off the end of the
                # resulting string.
                reply_message_bytes = echo_socket.recv(4096)
                reply_message = reply_message_bytes.decode(encoding='utf-
8').rstrip()

                print('Reply: ' + reply_message)

    finally:
        print('Closing connection')
        echo_socket.close()

    print('Goodbye!')



if __name__ == '__main__':
    conduct_echo_conversation(ECHO_HOST, ECHO_PORT)



# One problem that may potentially cause issues in this version of our
# echo client is our use of the recv() method.  We're operating under the
# assumption that we'll always get a whole message back from the server
# every time we call recv(), but there's no guarantee of that in practice.
# recv() waits until it has some data available; when it does, even if it
# is isn't as much as we asked for, it returns it.  So if the message sent
# back from the echo server is delivered in multiple pieces -- as can happen,
# especially as messages get larger, since data sent across the Internet is
# broken into smaller pieces called "packets" -- our program will fail.
#
# One solution to this problem is to use a loop to continue calling recv()
# until we get back data that contains the end-of-line sequence that we're
# expecting, gradually building up our message until the whole thing has
# arrived.  This can be a little bit tricky, though, because it's possible
# that we'll get more data than that (though not in this particular example,
# because we know we'll always receive exactly one line back for every line
# we receive), in which case we'd have to stash it somewhere for later reads.
#
# In general, a better solution to this problem is to use more appropriate
# tools.  The second version of this code example explores that.
```