```
# yackety_protocol.py
#
# ICS 32 Winter 2014
# Code Example
#
# This module implements the Yackety protocol via sockets, allowing a Python
# program to connect to a Yackety server and use it to send and view Yackety
# messages.  However, it contains no user interface and is not a "program"
# that can be executed; it provides utility functions that can be used by
# programs, in the same way that modules like "os" and "socket" do in the
# Python Standard Library.  It's fair to say, actually, that this module is a
# small library.  (See?  We can build libraries, too!)

import collections
import socket



# From our work with sockets in previous examples, we discovered that we
# needed to know three things about a connection at any given time:
#
# (1) The socket across which the connection is traveling
# (2) A pseudo-file object that lets us read input from that socket as
#     though we were reading from a text file
# (3) A pseudo-file object that lets us write input to that socket as
#     though we were writing to a text file
#
# Because these three things need to be available to various functions
# in our module, it's handy to create a kind of object to store all three.
# A namedtuple is a convenient way to do that.

YacketyConnection = collections.namedtuple(
    'YacketyConnection',
    ['socket', 'socket_input', 'socket_output'])



# As we'll see, when we ask the Yackety server to send us the last N
# messages, it will be handy to separate the name of the user who sent
# the message from the message's text.  For that reason, we'll create a
# kind of object -- another namedtuple -- to store each message.

YacketyMessage = collections.namedtuple(
    'YacketyMessage',
    ['username', 'text'])



def connect(host: str, port: int) -> YacketyConnection:
    '''
    Connects to a Yackety server running on the given host and listening
    on the given port, returning a YacketyConnection object describing
    that connection if successful, or raising an exception if the attempt
    to connect fails.
    '''

    yackety_socket = socket.socket()

    yackety_socket.connect((host, port))
```

```
    yackety_socket_input = yackety_socket.makefile('r')
    yackety_socket_output = yackety_socket.makefile('w')

    return YacketyConnection(
        socket = yackety_socket,
        socket_input = yackety_socket_input,
        socket_output = yackety_socket_output)



def login(connection: YacketyConnection, username: str) -> bool:
    '''
    Logs a user into the Yackety service over a previously-made connection,
    returning True if successful and False otherwise.
    '''
    _write_line(connection, 'YACKETY_HELLO ' + username)
    return _expect_line(connection, 'YACKETY_HELLO')



def send(connection: YacketyConnection, message: str) -> bool:
    '''
    Sends a message to the Yackety server on behalf of the currently-
    logged-in user, returning True if successful and False otherwise.
    '''
    _write_line(connection, 'YACKETY_SEND ' + message)
    return _expect_line(connection, 'YACKETY_SENT')



def last(connection: YacketyConnection, how_many_messages: int) ->
[YacketyMessage]:
    '''
    Retrieves the most recent few messages from Yackety.  The how_many_messages
    parameter determines how many messages we want; the Yackety server will
    send back as many as it has, up to the number we asked for.  The result
    of this function is a list of YacketyMessage objects, one per message
    sent back from the server, in the reverse of the order they were originally
    sent to Yackety (i.e., newest message first).
    '''
    _write_line(connection, 'YACKETY_LAST {}'.format(how_many_messages))

    messages = []

    # The Yackety protocol responds to the "YACKETY_LAST x" message
    # by sending, first, a count of how many messages it is responding
    # with.  This is done by sending a line "YACKETY_MESSAGE_COUNT y".
    # We need to know what the number y is.
    message_count_line = _read_line(connection)

    if message_count_line.startswith('YACKETY_MESSAGE_COUNT '):
        # We'll look at the characters on the line starting with index 22,
        # which skips "YACKETY_MESSAGE_COUNT " and convert those characters
        # to an integer.
        number_of_messages = int(message_count_line[22:])

        for i in range(number_of_messages):
            message_line = _read_line(connection)

            # To understand each message, we'll need to carefully break up
            # the line that was sent:
```

```
            #
            # * The first word should be YACKETY_MESSAGE
            # * The second word should be a username
            # * After the second word should be a space and then the full
            #   contents of the message, which we'll want to preserve
            #   including all spaces, punctuation, etc.
            if message_line.startswith('YACKETY_MESSAGE'):
                # Break the message into words, so we can pull out the username
                message_words = message_line.split()

                # The username is the second word
                username = message_words[1]

                # The text of the message starts after "YACKETY_MESSAGE", a
                # space, the usernane, and another space.  "YACKETY_MESSAGE"
                # and the two spaces are 17 characters total; the username's
                # length will vary depending on the message.  But if we add
                # 17 and the length of the username, that will reliably tell
                # us where the message text starts.
                text_start = 17 + len(username)

                # Pull out the message text
                text = message_line[text_start:]

                # Create a YacketyMessage object and append it to a list of
                # messages that we plan to return
                messages.append(YacketyMessage(username, text))

    # Return all of the messages that came back from the server
    return messages



def goodbye(connection: YacketyConnection) -> None:
    'Exchanges YACKETY_GOODBYE messages with the server'
    _write_line(connection, 'YACKETY_GOODBYE')
    _expect_line(connection, 'YACKETY_GOODBYE')



def close(connection: YacketyConnection) -> None:
    'Closes the connection to the Yackety server'

    # To close the connection, we'll need to close the two pseudo-file
    # objects and the socket object.
    connection.socket_input.close()
    connection.socket_output.close()
    connection.socket.close()



# These are "private functions", by which I mean these are functions
# that are only going to be used within this module.  They're
# hidden implementation details.  By starting their names with an
# underscore, we're making clear to users of this module that these
# functions are intended to be private.



def _read_line(connection: YacketyConnection) -> str:
    '''
```

```
    Reads a line of text sent from the server and returns it without
    a newline on the end of it
    '''

    # The [:-1] uses the slice notation to remove the last character
    # from the string.
    return connection.socket_input.readline()[:-1]



def _expect_line(connection: YacketyConnection, line_to_expect: str) -> bool:
    '''
    Reads a line of text sent from the server, expecting it to contain
    a particular text.  Returns True if the expected text was sent,
    False otherwise.
    '''
    return _read_line(connection) == line_to_expect


def _write_line(connection: YacketyConnection, line: str) -> None:
    '''
    Writes a line of text to the server, including the appropriate
    newline sequence.
    '''
    connection.socket_output.write(line + '\r\n')
    connection.socket_output.flush()
```