CIS 41B - Lab 5

Write a client-server application where the server can respond to a number of clients.

This is the last lab so it reviews some concepts from the first half of the class, and you have a break from GUI and OOP.

Overview

The application is in 2 files: server.py and client.py. Both use sockets for communication and both run from the command line. (When we get to low level networking, there's no GUI and no IDE, it's all on the command line.)

The server code simulates a server that's running at a remote site. The server accepts a request from a client and produces an output data set for the client. The client receives the output data and plots it.

Server tasks:

* As it comes up, the server accepts 2 command line arguments: the max number of clients that it can respond to, and the number of seconds for the timeout timer.
* Then the server waits for request for up to the max number of clients:
  + For each request, the server creates a thread to respond to the client.
  + If the number of clients < the max number of clients, then the extra listening socket times out when the number of seconds on the timer is up.
* For each client, the server accepts a command to produce an output data set with a numpy function.

Client tasks:

* Prompt the user with a menu to run 2 math functions: power function and sine function, or to quit
  + If the user chooses the power function, prompt for the exponent, min x value, and max x value
  + If the user chooses the sine function, prompt for the frequency
* Send all user input to the server
* Receives a data set from the server
* Plot the data set with matplotlib
* When the user closes the plot, loop back to prompt the user again
* End with the user chooses to quit.

Recommended steps to work on the lab

Step A

1. Write a server for 1 client and write the client.

* Use the IDE's text editor to write server.py and client.py, if you normally use an IDE, but don't run code with it.
* Write the server to service 1 client, so don't put in threading code yet
  + Check for the correct number of command line arguments, and check that they're integers. End the program with a descriptive error message if any of the command line arguments is invalid.
  + For the power function:
    - Use the min and max x values to create a numpy array of 50 data points between the min and max values. Make the 50 a global constants and use it.
    - Use numpy to create an output array of xexponent
    - Send the array back to the client
    - Return the array
  + For the sine function:
    - Create a numpy array of 50 data points between the 0 and 1. These are the x values.
    - Use the frequency and numpy to create a data set which is: frequency \* 2 \* pi \* x
    - Send the array back to the client
    - Return the array
  + Write a decorator which will print the max and min of the return values of a function
    - * Use the decorator with the power function and sine function
* Write the client to make each of the 2 requests
  + Present the user with a menu and ask for a choice of p (power function), s (sine function), or q (quit)
    - Check for valid choice or re-prompt
    - Based on the user's valid choice, prompt for all supporting input on one line, with comma as separator.
    - Check that all supporting input are integers or loop back and re-prompt
  + Format the choice and supporting input into text string. This text string is the request that's sent to the server.
  + Receive the output data set from the server and call the appropriate function to plot the data.
    - To call the appropriate function, you cannot use an if statement.
  + Use matplotlib to plot the power function or the sine function
    - Produce the same input data set (x values) that the server uses. Make the 50 (number of data points) a global constants and use it.
    - Plot the output data set from the server. Make sure to have a descriptive plot title and x axis label. (See sample output)
* Requirement for client and server code:
  + Each request for data from a client is a single command that's sent to the server.
  + Each response from the server (with output data) is a single command that's sent to the client.
  + In addition to the functions to plot the data, both client and server code should be divided into appropriate functions. Don't write one big main function.
* Recommendation for code development on the command line:
  + The IDE is mainly the text editor for this lab.
  + You need to run client.py and server.py on the command line, in 2 separate command / terminal windows, so that they are 2 different processes. They need to be 2 processes because sockets are used for processes to communicate, whether the 2 processes are on the same system (like this lab), or across the network.
  + Welcome to the world of command line debugging, which many experienced programmers choose to use because, in their mind, only wimps need an IDE.  
     In case you're new to this kind of debugging:
    - Run the code until it fails, look at the error message and the line number that fails, go to the IDE and the failed line, and fix your code. Then *don't forget to save* the file before re-running the code.
    - If you get stuck in a loop somewhere, try control-c.
    - And if control-c doesn't work because your client (or server) is waiting for the server (or client) and won't listen to you, then close the command window and start again.
    - The up arrow key to re-run the previous command will be your friend.

Step B

Make sure step A completely works before going to step 2. You don't want to add multithreading on top of partially working client-server code.

* In the server code, add 1 thread and have the child thread respond to the client request. The thread can start from the response socket. This means you have the same socket for all clients, and just the response part is different for each client.
* To prepare for multiple clients, add time out code for the listening socket. Since the listening socket's accept() method blocks, add a time out so that it doesn't wait forever if there's no client. In real life, the server would wait forever for the clients.
* Make sure the server with thread works just as well as when you were in step A, before starting step C.  
  Set the timer to 10 seconds and test the timeout to see that it works. It's useful to have the server print "timed out" so you know that the time out happened.

Step C

Add code to the server so that from the command line arguments, it accepts up to 4 possible clients and a timer between 10 and 30 seconds.

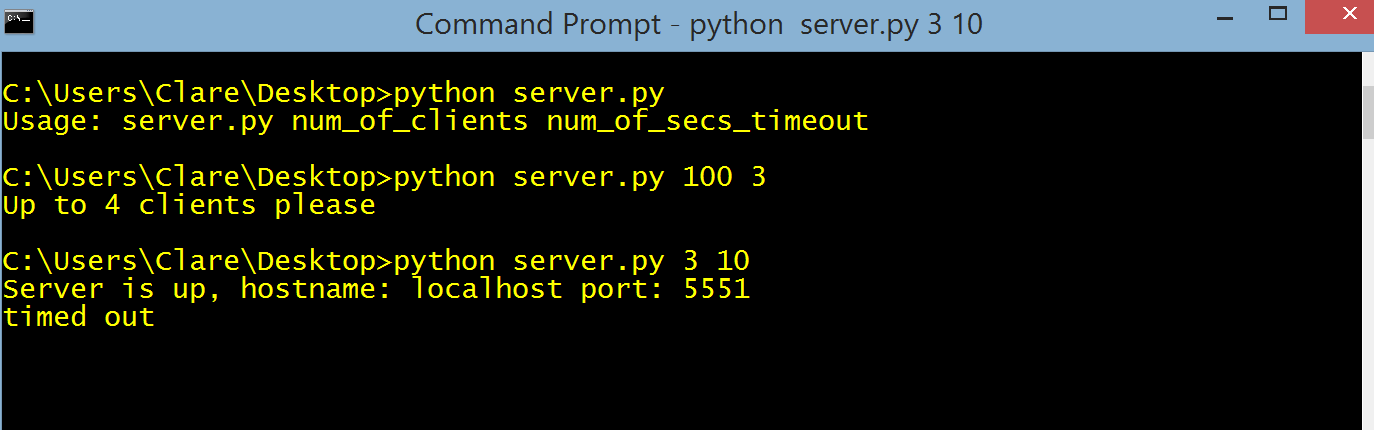
Suggestion for testing:

* Run the server code at one command / terminal window.
* Open up 2 other command / terminal windows for 2 clients. Each client is in a different window.  
  The 3rd connection should eventually time out.
* Run different requests to the server at the 2 windows to see that the 2 clients can do different things at the "same" time.

Sample output

Server window

no command line argument

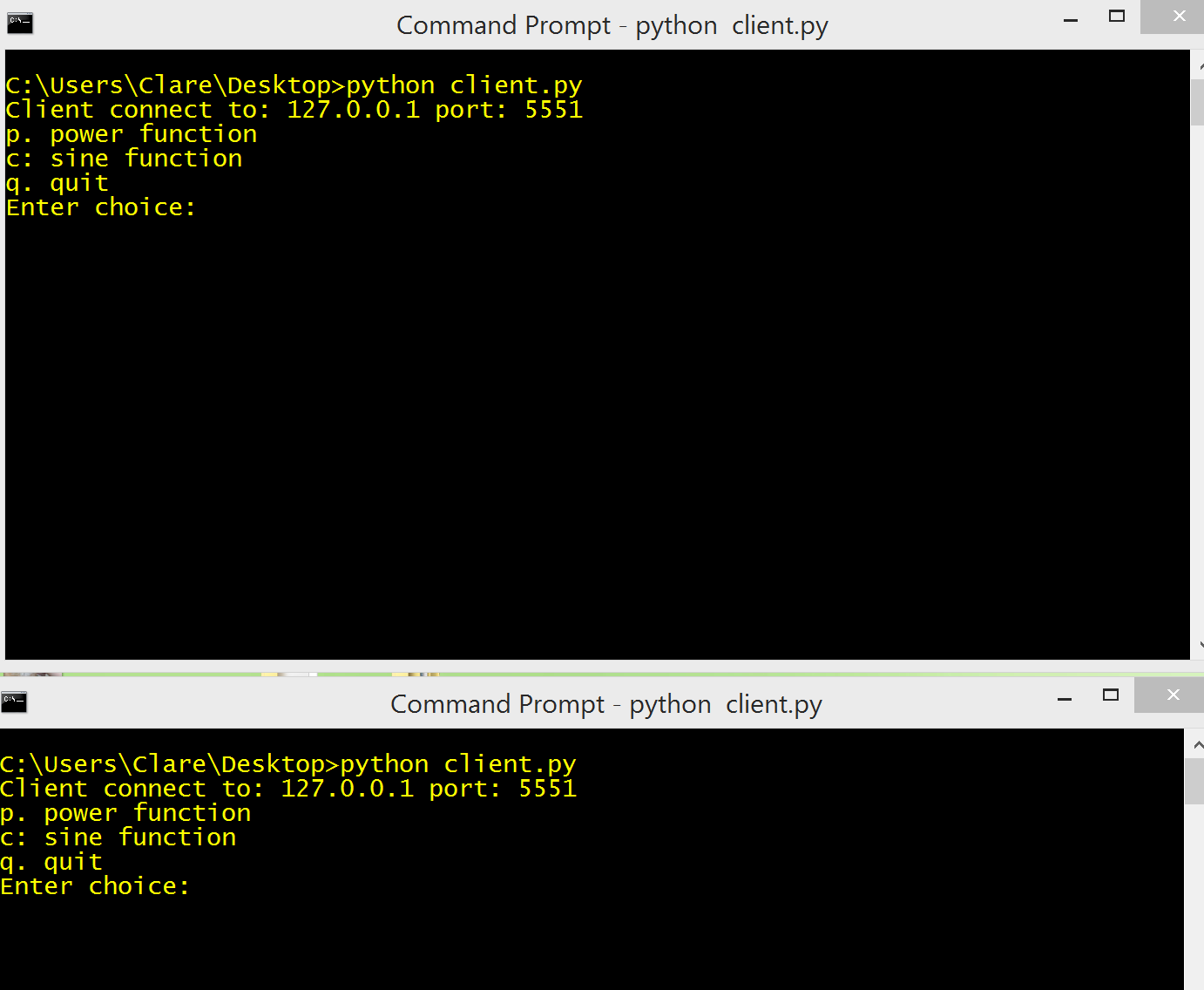


time out after 10 seconds

server identifies itself

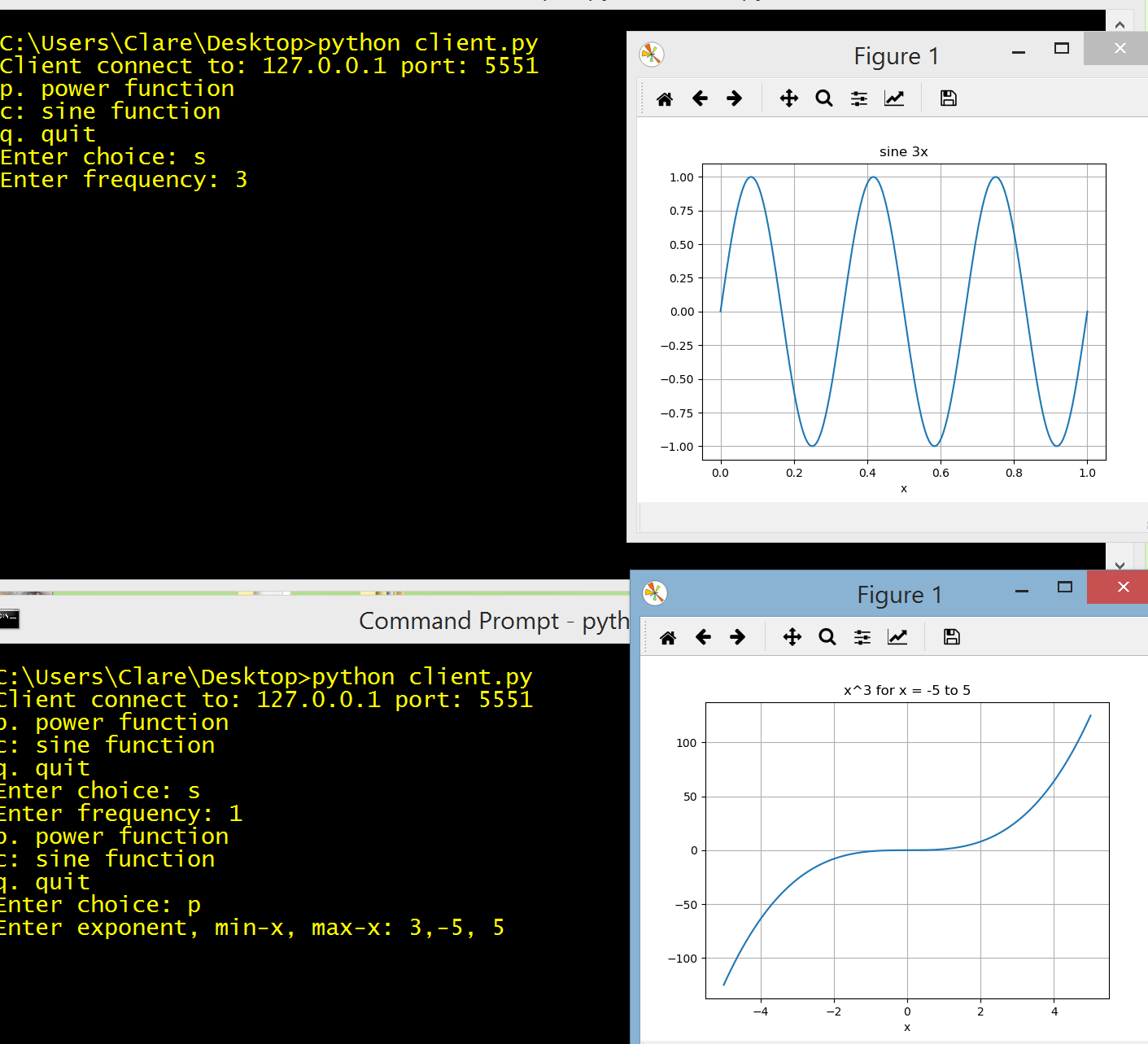
invalid argument

Client windows (2 clients)



*Hopefully the spelling on your menu is better than mine*

Out of 3 possible clients, 2 clients connect to the server. The 3rd connection times out (shown in the server window above)

Client windows (and plot result) after 1 request from a client, and 2 requests from another client 

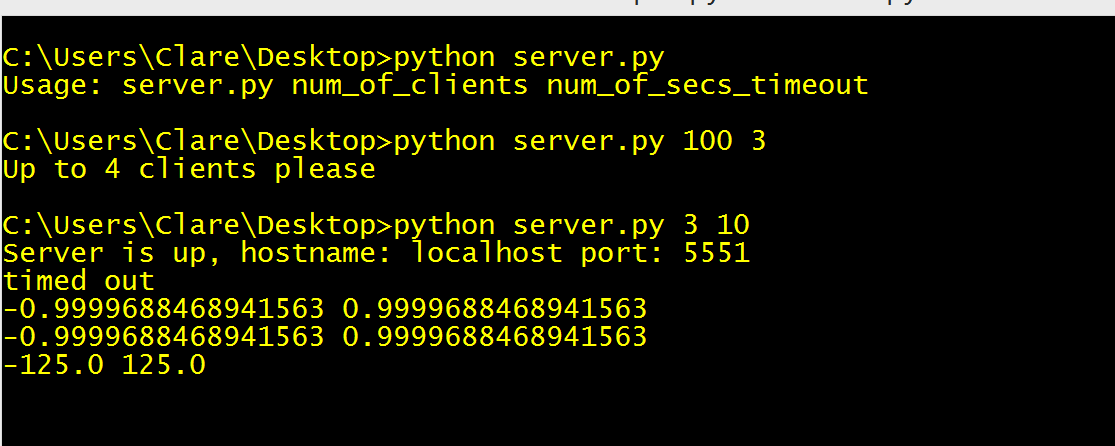
Note the plot titles which are specific to the data that the user entered

In the example above, the first client requests a sine function.

The second client requests a sine function first, and then requests the power function.

There are 3 requests total.

This is the corresponding server window:



There are 3 lines of min and max values of the plot data, one line per client request.

The print out comes from the decorator

When done, submit client.py and server.py.