

**VEL’S VIDYASHRAM SENIOR SECONDARY SCHOOL**

IP PROJECT 2020-21

Two-person Chat Messenger Application

**Name:** Anirudh.M

**Class:** XII

**Section:** B

*INSERT BONAFIDE HERE*

***Acknowledgement***

I would like to express my profound thanks to my Informatics Practices teacher Mrs. Tamilselvi, without whose valuable guidance this project could not have been made faultless. Her suggestions and her instructions have served as the major contributors to the completion of this project.

I would also like to extend my gratitude to the principal Mrs. Vasanthi Vishwanathan for providing me with all facilities required to finish this project successfully.

Finally, I would like to thank my parents and friends without whose timely advice and kind suggestions this project would be at a standstill.

***INDEX***

1. About the Project
2. Hardware and Software Requirements
3. Modules Used
4. Source Code
5. Output Screens
   1. Program Execution
   2. CSV File Modification
   3. Graphs of Resources
6. Conclusion
7. Bibliography

***ABOUT THE PROJECT***

Chat applications have been existing for a long time in this era of modern communications, ranging from the ancient Yahoo! Chat rooms to the modern applications like WhatsApp, Facebook Messenger and Snapchat.

HISTORY:

In 1961, MIT's Computation Center built the [Compatible Time Sharing System](https://en.wikipedia.org/wiki/Compatible_Time-Sharing_System) (CTSS) which facilitated communication via text message for up to 30 people. The first dedicated online chat platform is recognized as [CompuServe](https://en.wikipedia.org/wiki/CompuServe)’s CB Simulator, which was released in 1980 and required a monthly membership fee. In 1996 Israeli firm [Mirabilis](https://en.wikipedia.org/wiki/Mirabilis_(company)) released ICQ (short for “I Seek You”), the first widely used online messenger platform.

In the early 1980s, Quantum Link or Q-Link was introduced by Commodore which was an online service that enabled people to chat simultaneously along with playing games and sharing files. It further changed its name to AOL (America Online) in the year 1991 and 1997, it launched the AOL instant messenger which proved to be a huge success. It was later challenged by other competitors in the early 2000’s such as Skype and Yahoo chat room. The most famous chat application came in the late 2000s which is known as Blackberry Messenger and it revolutionized the world of modern chat applications.

Chat messengers are the perfect example of mobile communications. They are easy to use and easy to manage even on the go. They are now a big part of all communication, whether personal, professional or informational, in the world because of their various uses.

All chat systems are primarily composed of interconnected sockets. A socket primarily provides an ending point for communications between two nodes in a network, or even two different processes. Usually, a socket is identified by an IP address associated with a port (number).

In this project, a simple example of such sockets has been taken in the form of a server-client system. Here, the client will request for the data from the server and the server will respond by providing that particular information. For the communication between the client and the server, a connection needs to be established; which is performed by a socket.

The servers listen to a particular port and wait for the request from the client. The connection request by the client socket is allowed by the server and hence, the connection is made. For instance, a telnet server listens to port number 23. Similarly, the HTTP server listens to port number 80 and an FTP server listens to port number 21.

***HARDWARE AND SOFTWARE REQUIREMENTS***

This project absolutely requires:

* A working computer, with working hardware components.
* The Python software. Any development environment would suffice. The build version is to be above 3.0.1
* The python modules pandas, socket, os, time, numpy and matplotlib.
* The modules of this project to be integrated into the root directory of the system.

Optionally, this project requires:

* Python IDLE version 3.9.1 (32- or 64-bit), or above.
* Matplotlib version 3.3.4, or above.
* Numpy version 1.20.1, or above.
* Pandas version 1.2.2, or above

***The Modules Used***

1. socket:

The Python socket module is used to create and manage sockets – the end-points of any communication channel. Usually referring to client-server connections, this module is used to describe the behaviour of such created sockets. It provides a direct interface to the Berkeley Software Distribution Application Programming Interface (BSD API).

In a nutshell, this module is a straightforward transliteration of the UNIX system socket call and library interface to an object-oriented programming style. This means that it is useful to create low-level networking interfaces. The most basic function in this module is ‘socket.socket()’, which creates a socket and returns a ‘socket object’. The other methods and functions defined in the module can be implemented on the socket objects thusly created.

The creation of a socket needs some parameters. These are given as follows:

* socket\_family − This can be either AF\_UNIX or AF\_INET, or the other AF\_ constants defined. Usually it is defaulted to AF\_INET, as this accounts for almost 99% of sockets in use.
* socket\_type − This can be either SOCK\_STREAM or SOCK\_DGRAM, or one of the other defined SOCK\_ constants.
* protocol − This is usually left out, defaulting to 0.
* fileno – Calls a pre-defined file for auto-detection of values. Can be overruled by providing explicit values for each keyword argument.

Thus, the syntax to create a basic socket is:

‘socket.socket(socket\_family, socket\_type, protocol = 0)’

Created sockets are always non-inheritable. This means that even if the socket contains a subset of sockets or other objects, the methods used on the socket itself need not be available or applicable to each element in the socket. This restricts reusability.

The socket module contains different methods for a server and a client application. For a server, these include, but are not restricted to:

* socket.socket.bind() – It is used to bind the created socket to an address, usually given as a (hostname, port) tuple.
* socket.socket.listen() – It is used to check for requests to connect with server. Takes allowed number of such requests as it’s argument.
* socket.socket.accept() – It is used to accept a connection request from a client. The formed connection is passive.

For the client, there exists very few socket commands. The most prominent among these is socket.socket.connect(), whose main purpose is to connect with the server socket.

There also exists many other miscellaneous methods, used in both client- and server-side applications. These include:

* socket.socket.send() – It is used to transmit an outgoing message.
* socket.socket.recv() – It is used to receive and decode an incoming encoded message.
* socket.socket.close() – It is used to close a connection. It can further be used to dissolve sockets. It makes the socket non-operational.
* socket.socket.gethostname() – It is used to get the hostname of the node on which the command is run.
* socket.socket.setdefaulttimeout() – It is used to set the default timeout in seconds for each and every socket function. Raises a timeout error if runtime exceeds the entered value.

The timeout errors arise when the default timeout is not None or if the socket is called with a timeout parameter. If such a case arises, then the socket connections created by the socket.socket.accept() and socket.socket.connect() functions also inherit that timeout, meaning that the created sockets also return a timeout error if runtime exceeds he passed argument.

The working of the socket connections is explained below.

Let us consider a host with an IP address of 146.85.10.2 and the server with an IP address of 161.25.10.8. Let the socket numbers for the host be 146.85.10.2 : 1608 and the server be 161.25.10.8 : 80. Now, to establish the communication link between the server and the host, we will use the socket.

When a client wants to send a connection request, it is assigned a port number by the host computer. Here, 1608 is the port number assigned by the host computer, Moreover, the port number assigned should be greater than 1024 as all the port numbers below it are considered to be well known for the standard service and are used for client process. At the server end, the port number is 80, which is less than 1024 as it is used by some standard service. The packets traveling from the client process to the server process are delivered appropriately based on the port number.

1. pandas:

pandas is a [Python](https://www.python.org/) package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labelled” data both easy and intuitive. Developed by Wes McKinney, it aims to be the fundamental high-level building block for doing practical, real-world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis/manipulation tool available in any language. It is already well on its way toward this goal.

pandas is well suited for many different kinds of data:

* Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet.
* Ordered and unordered (not necessarily fixed-frequency) time series data.
* Arbitrary matrix data (homogeneous or heterogeneous) with row and column labels.
* Any other form of observational / statistical data sets. The data need not be labelled at all to be placed into a pandas data structure.

The two primary data structures of pandas, [Series](https://pandas.pydata.org/docs/reference/api/pandas.Series.html#pandas.Series) (1-dimensional) and [DataFrame](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html" \l "pandas.DataFrame" \o "pandas.DataFrame) (2-dimensional), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering. For R users, [DataFrame](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html" \l "pandas.DataFrame" \o "pandas.DataFrame) provides everything that R’s data.frame provides and much more. pandas is built on top of [NumPy](https://www.numpy.org/) and is intended to integrate well within a scientific computing environment with many other 3rd party libraries.

It has numerous uses, some of which are:

* Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data.
* Size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects.
* Automatic and explicit data alignment: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let Series, DataFrame, etc. automatically align the data for you in computations.
* Powerful, flexible group-by functionalities to perform split-apply-combine operations on data sets, for both aggregation and transformation of data.
* Make it easy to convert ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects.
* Intelligent label-based slicing, fancy indexing, and subsetting of large data sets.
* Intuitive merging and joining data sets.
* Flexible reshaping and pivoting of data sets.
* Hierarchical labeling of axes (possible to have multiple labels per tick).
* Robust IO tools for loading data from flat files (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast HDF5 format.
* Time series-specific functionality: date range generation and frequency conversion, moving window statistics, date shifting, and lagging.

An important thing to note is that compared to other software operating in the same field, pandas is remarkably faster. But, with generalization comes a drop in performance, so faster specialized tools can and will be created.

The three available data structures in pandas are:

* Series: A one-dimensional labelled homogeneously typed array. Created by the command pandas.Series([<data>,<index>, <dtype>, <name>, <copy>])
* DataFrame: A general two-dimensional labelled, size-mutable container having potentially heterogeneous columns. Constructed using pandas.DataFrame([<data>, <index>, <columns>, <dtype>, <copy>])
* Panels: A three-dimensional container of data. Constructed using pandas.Panel([<data>, <items>, <major\_axis>, <minor\_axis>]). They are deprecated as of pandas 0.20.0, and have been replaced by MultiIndex DataFrames.

All pandas data structures are value-mutable (the values they contain can be altered) but not always size-mutable. The length of a Series cannot be changed, but, for example, columns can be inserted into a DataFrame. However, the vast majority of methods produce new objects and leave the input data untouched. In general immutability is favoured where sensible.

1. time:

Along with datetime and calendar, this module constitutes the whole of time- and date-related libraries provided by python. time provides various time-related functions, but their functionalities depends on the platform on which it is run. Some functions are not available on specific platforms, and the semantics may vary across platforms.

An explanation of some terminology and conventions is in order.

* The *epoch* is the point where the time starts, and is platform dependent. For Unix, the epoch is January 1, 1970, 00:00:00 (UTC). To find out what the epoch is on a given platform, look at time.gmtime(0).
* The phrase *seconds since the epoch* refers to the total number of elapsed seconds since the epoch, typically excluding [leap seconds](https://en.wikipedia.org/wiki/Leap_second). Leap seconds are excluded from this total on all POSIX-compliant platforms.
* The functions in this module may not handle dates and times before the epoch or far in the future. The cut-off point in the future is determined by the C library; for 32-bit systems, it is typically in 2038.
* UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT). The acronym UTC is not a mistake but a compromise between English and French.
* The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most Unix systems, the clock “ticks” only 50 or 100 times a second.
* Times are expressed as floating point numbers, [time()](https://docs.python.org/3/library/time.html#time.time) returns the most accurate time available and [sleep()](https://docs.python.org/3/library/time.html#time.sleep) will accept a time with a nonzero fraction.

The module time can be used to get the actual network time, the computer time or even the time since the epoch. It can also be used to monitor runtimes, create Exception classes of TimeoutError and measuring the efficiency of programs.

Some basic functions defined in this module are:

* time.ctime([<time\_in\_sec>]) – It converts the time value expressed in seconds since the epoch to a meaningful time string of format ‘Day Month Date HH:MM:SS Year’. If no arguments are passed, it takes it’s argument as time.time() as default.
* time.sleep(<seconds>) – It instructs the program to pause for required number of seconds as specified in the argument. Suspension time also depends on external signals and scheduling parallel processes may extend the sleep time.
* time.strftime(*format*[,*time*]) – Converts a tuple or [struct\_time](https://docs.python.org/3/library/time.html" \l "time.struct_time" \o "time.struct_time) representing a time as returned by [gmtime()](https://docs.python.org/3/library/time.html" \l "time.gmtime" \o "time.gmtime) or [localtime()](https://docs.python.org/3/library/time.html" \l "time.localtime" \o "time.localtime) to a string as specified by the *format* argument. If *time* is not provided, the current time as returned by [localtime()](https://docs.python.org/3/library/time.html" \l "time.localtime" \o "time.localtime) is used. *format* must be a string. [ValueError](https://docs.python.org/3/library/exceptions.html" \l "ValueError" \o "ValueError) is raised if any field in *time* is outside of the allowed range.
* time.time() – Returns time elapsed since epoch as a floating point number. The reference epoch depends on the platform and independent computer. It returns the system clock time.

1. os:

This module library provides a way for an interface with the operating system. In a way, it is the miscellaneous interface between the operating system and the python software.

This module library provides a portable way of using operating system dependent functionality. If you just want to read or write a file see [open()](https://docs.python.org/3/library/functions.html#open), if you want to manipulate paths, see the [os.path](https://docs.python.org/3/library/os.path.html#module-os.path) module, and if you want to read all the lines in all the files on the command line see the [fileinput](https://docs.python.org/3/library/fileinput.html#module-fileinput) module. For creating temporary files and directories see the [tempfile](https://docs.python.org/3/library/tempfile.html" \l "module-tempfile" \o "tempfile: Generate temporary files and directories.) module, and for high-level file and directory handling see the [shutil](https://docs.python.org/3/library/shutil.html#module-shutil) module.

Notes on the availability of these functions:

* The design of all built-in operating system dependent modules of Python is such that as long as the same functionality is available, it uses the same interface; for example, the function os.stat(path) returns stat information about path in the same format (which happens to have originated with the POSIX interface).
* Extensions peculiar to a particular operating system are also available through the [os](https://docs.python.org/3/library/os.html#module-os) module, but using them is of course a threat to portability.
* All functions accepting path or file names accept both bytes and string objects, and result in an object of the same type, if a path or file name is returned.
* An important thing to note is that all functions in this module raise the OSError BaseException in the case of incompatible or invalid or inaccessible file\_names and/or paths, or other arguments not accepted by the operating system.

Basically, this module is used to access the command line interface of operating systems to get data that otherwise need to be derived through difficult routes. In Windows, it is directly connected through pipes to the Command Prompt interface.

Some functions or methods in this module are:

* os.name() – Returns the name of the operating system of the computer on which it is run.
* os.open(<path>) – Opens the file at the path specified.
* os.pipe() – Creates a pipe and returns a tuple (r, w), where r is readable and w is writable.
* os.popen (*cmd, mode = ‘r’, buffering = 1) –* Opens a pipe to the command prompt on Windows. The ‘mode’ argument determines if the pipe is readable (‘r’) or writable (‘w’). The ‘buffering’ argument accepts only integers and will set the buffering policy. It determines the maximum bytes that can be stored in temporary memory until the file/command finishes loading.

1. matplotlib:

Developed by John D. Hunter, matplotlib is the primary plotting library for the Python programming language. It is a comprehensive library for creating static, animated or even interactive visualizations. It is built on the NumPy library, and uses general-purpose GUIs like Tkinter, Qt or GTK+ to display plots or visualizations.

This module is widely preferred as the end results of most of its functions and methods can be used or manipulated easily without using complex methods. It has the distinct advantage of supporting most types of output formats, and having a cross-platform approach. It also can be used with different modules without any problems arising, the most notable of such instances being its usage in the pandas module for visualization of tabular data.

This library is generally used to create visualizations of two types: *Static* and *Animated*. For Static visualizations, the module *pyplot* is preferred while the module *animation* is used to create visualizations that are updated in real-time.

The module pyplot provides the different supporting methods needed to create the needed visualizations for the given set of data, whether it be a bar graph, a histogram, a pie chart or even a simple line graph. Along with it, it also houses the necessary tools to create, adjust and destroy the *artists* – the graphical output screens on which the visualizations are displayed.

Some well-known methods of pyplot are listed below:

* pyplot.plot(data, \*\*kwargs) – The command needed to create a basic line plot. Plots y versus x as lines and/or markers. Accepts two ways of argument listing. One is to pass a list of arguments along with their keywords. The other is to pass argument lists positionally, which also allows to pass different sets of data through a single line. Some widely used keyword arguments are linestyle (ls), linewidth (lw), color (c), and others.
* pyplot.bar(data, heights, \*\*kwargs) – The command used to create bar graphs. Bars are positioned on the x-axis according to the *alignment* specified. Most keyword arguments can take both single-valued inputs that are copied to all bars or a list-like of inputs corresponding to each bar in order. Popular keyword arguments are width, align and bottom.
* pyplot.hist(data, \*\*kwargs) – The command used to create histograms. The data passed is grouped into *bins* as per specified in the function call. If *bins* is not called explicitly, it defaults to 10. If data is passed in the form of sequences, it is not necessary that every sequence should have the same length. Some keyword arguments are bins, weights, histtype.
* pyplot.subplot(\*args, \*\*kwargs) – The command to create a subplot in the artist. Needs at least three arguments specifying the number of subplots being created, their orientation and the subplot to be modified. Some keywords used here are label and polar.
* pyplot.show() – Used to display the artist. Usually called only at the end of the script, as calling multiple times results in outputs differing across platforms. Called artist is not mutable.

The module animation provides a supporting framework around which the animation functionality is built. There are two main interfaces to achieve this:

* FuncAnimation – The most convenient way to build animations. Works by repeatedly calling an already defined function.
* ArtistAnimation – A way to create an animated image by using a fixed set of artist objects.

Though it has so many pros, it also has numerous shortcomings. Some are listed below.

* Matplotlib has an imperative API which is often overly verbose.
* Sometimes poor stylistic defaults.
* Poor support for web and interactive graphs.
* Often slow for large & complicated data.

1. NumPy:

NumPy is the fundamental package for scientific computing in Python. Introduced in early 2005 by Travis Oliphant, it is a Python library that provides multidimensional array objects, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the *ndarray* object. This encapsulates *n*-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

* NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an *ndarray* will create a new array and delete the original.
* The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
* NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python’s built-in sequences.

Why is NumPy so fast? It is due to two reasons: *vectorization* and *broadcasting*. Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just “behind the scenes” in optimized, pre-compiled C code. Vectorized code has many advantages, among which are:

* Vectorized code is more concise and easier to read.
* Fewer lines of code generally means fewer bugs.
* The code more closely resembles standard mathematical notation (making it easier, typically, to correctly code mathematical constructs).
* Vectorization results in more “Pythonic” code. Without vectorization, our code would be littered with inefficient and difficult to read for loops.

Broadcasting is the term used to describe the implicit element-by-element behaviour of operations; generally speaking, in NumPy all operations, not just arithmetic operations, but logical, bit-wise, functional, etc., behave in this implicit element-by-element fashion, i.e., they broadcast. Moreover, in the example above, a and b could be multidimensional arrays of the same shape, or a scalar and an array, or even two arrays of with different shapes, provided that the smaller array is “expandable” to the shape of the larger in such a way that the resulting broadcast is unambiguous.

Some functions and methods provided by NumPy are:

* numpy.array(<data\_sequence>) – Creates an array. The passed data sequence should be a list-like or an array-like object.
* numpy.arange(start, stop[, step]) – Creates a ndarray with values spaced at regular intervals as defined by the *step* value. The *start* and *stop* values define from which numbers the array should start and end. If *step* is not passed explicitly, by default, it assumes the value of 1.
* numpy.array.reshape(<size\_params>) – The command used to create a new array of needed size from an old array, conserving the data contained inside. The size passed should have same number of containers as the original array, or else it will return a ValueError. Does not change the original array.

Though it has a wide variety of uses, it also has various limitations. Inserting or appending entries to an array is not as trivially possible as it is with Python's lists. The np.pad(...) or np.append(…) routines to extend arrays actually creates new arrays of the desired shape and padding values, copies the given array into the new

one and returns it. NumPy's np.concatenate([a1,a2]) operation does not actually link the two arrays but returns a new one, filled with the entries from both given arrays in sequence. Reshaping the dimensions of an array with np.reshape(...) is only possible as long as the number of elements in the array does not change.

Algorithms that are not expressible as a vectorized operation will typically run slowly because they must be implemented in "pure Python", while vectorization may increase memory complexity of some operations from constant to linear, because temporary arrays must be created that are as large as the inputs.

***SOURCE CODE***

1. The Background Module misc:

import time

import pandas as pd

import os

import matplotlib.pyplot as plt

def iterate(x):

for i in x: print(i)

def read(y):

if len(y)>0:

for i in range(0, len(y)):

if y.isna()[i] == False:

print(y.pop(i))

def check(x, ser, ser1=None):

try:

if x==0:

global temp

temp=ser.copy().dropna()

else:

try:

global sert

sert=ser1.copy()

x1=len(temp)

x2=len(sert)

if x1>=x2:

i=pd.RangeIndex(0, x1)

sert=sert.reindex(i)

else:

i=pd.RangeIndex(0, x2)

temp=temp.reindex(i)

temp1=temp==sert

return temp1

except: pass

except: pass

def rem\_na(\*sers):

for ser in sers:

ser=ser.dropna().reset\_index().drop(columns='index')

class raw\_data:

def \_\_init\_\_(self):

self.data = []

def CPU():

CPU\_Pct = str(os.popen('wmic cpu get loadpercentage').read()).strip()[-3:].strip()

try:

CPU\_Pct = int(CPU\_Pct)

except:

CPU\_Pct = 0

return CPU\_Pct

def RAM(mode = None):

if mode == None:

RAM\_kbytes = float(os.popen('wmic os get freephysicalmemory').read().strip()[-9:].strip())

return RAM\_kbytes

elif mode in ('MB', 'mb', 'mB', 'Mb'):

RAM\_kbytes = float(os.popen('wmic os get freephysicalmemory').read().strip()[-9:].strip())

RAM\_Mb = RAM\_kbytes/1024

return RAM\_Mb

else:

raise AttributeError

Cpu\_data = []

Ram\_data = []

RamM\_data = []

time\_arr = []

def graph\_create\_plot(col\_size):

fig, axs = plt.subplots(1, col\_size)

plt.subplots\_adjust(wspace=1, hspace=1)

return fig, axs

class graphing:

def \_\_init\_\_(self):

self.data = []

def graph\_plot(i, axs=graph\_create\_plot(3)[1]):

tm = time.strftime('%H:%M:%S')

global Cpu\_data

global Ram\_data

global RamM\_data

global time\_arr

time\_arr.append(tm)

try:

Cpu\_data.append(raw\_data.CPU())

axs[0].set\_xlabel('Time')

axs[0].set\_ylabel('CPU Usage')

axs[0].set\_title('CPU vs Time plot')

try:

axs[0].plot(time\_arr, Cpu\_data, color = 'blue')

except Exception as e: print(e)

except Exception as e: print(e, 'THIS IS ERROR!')

try:

Ram\_data.append(raw\_data.RAM())

axs[1].set\_xlabel('Time')

axs[1].set\_ylabel('RAM availability (in kilobytes)')

axs[1].set\_title('RAM(kb) vs Time plot')

try:

axs[1].plot(time\_arr, Ram\_data, color = 'red')

except Exception as e: print(e)

except: print('This is Error2')

try:

RamM\_data.append(raw\_data.RAM(mode = 'MB'))

axs[2].set\_xlabel('Time')

axs[2].set\_ylabel('RAM availability (in Megabytes)')

axs[2].set\_title('RAM(Mb) vs Time plot')

try:

axs[2].plot(time\_arr, RamM\_data, color = 'magenta')

except Exception as e: print(e)

except: print('This is Error 3')

for i in range(3):

for tick in axs[i].get\_xticklabels():

tick.set\_rotation(75)

try:

Cpu\_data = Cpu\_data[-10:]

Ram\_data = Ram\_data[-10:]

RamM\_data = RamM\_data[-10:]

time\_arr = time\_arr[-10:]

axs[0].set\_xlim(left = time\_arr[-10], right = time\_arr[-1])

axs[1].set\_xlim(left = time\_arr[-10], right = time\_arr[-1])

axs[2].set\_xlim(left = time\_arr[-10], right = time\_arr[-1])

except: pass

1. The Server-side Program:

import socket as sc

import time

import pandas as pd

import numpy as np

from misc import \*

try:

x=pd.read\_csv(r'C:\Users\Aniru\OneDrive\ Documents\Temp.csv')

except:

file\_path=input(str('Enter CSV File Path: '))

x=pd.read\_csv(file\_path)

CMsgs1=x.CMsgs.dropna()

read(x.CMsgs)

x\_copy=CMsgs1.copy()

x.drop(columns=['CMsgs'], inplace=True)

x.SMsgs=x.SMsgs.dropna().reset\_index().drop(columns='index')

check(0, x\_copy)

i=input(str("Wait to read? (y/n): "))

if i=='y':

time.sleep(20)

else:

pass

sc.setdefaulttimeout(10)

s = sc.socket()

host = sc.gethostname()

print(f"Server starts on host: {host}")

port = 8080

s.bind((host, port))

print("")

print("Server bound to host and port successfully")

print("")

print("Server is waiting for incoming connection")

print("")

try:

s.listen(1)

conn, addr = s.accept()

sc.setdefaulttimeout(None)

print(f"{addr} has connected to the server and is now online")

print("")

while 1:

message = input(str('>>'))

if message not in np.array(('/e', '/h', '/a', '/c', '/v')):

message = message.encode()

conn.send(message)

print("Message has been sent")

print("")

inc\_msg = conn.recv(1024)

inc\_msg = inc\_msg.decode()

print(f"Client: {inc\_msg}")

print()

else:

if message == '/e':

conf = input(str('y/n: '))

if conf=='y':

exit()

elif message == '/h':

print("Welcome to the Server side Chat Monitor! This utilises csv and socket modules to provide a smooth experience of a private chat! Don't mind the kinks... ")

print("There are just 5 commands in our command line. They are as follows:")

print("/e -- Exits the application")

print("/h -- The command used to get here. It's the command for help, if you didnt get it yet")

print("/a -- Gives a formatted automatic reply")

print("/c -- Clears the cache/backup chats")

print("/v -- To view the backup chats again")

print("")

elif message == '/a':

msg = 'Server is very busy! Will reply shortly!'

msg = msg.encode()

conn.send(msg)

inc\_msg = conn.recv(1024)

inc\_msg = inc\_msg.decode()

print(f"Client: {inc\_msg}")

print()

elif messsage == '/c':

conf = input('Clear ALL backup messages?(y/n): ')

if conf == 'y':

x.drop(columns=['CMsgs', 'SMsgs'], inplace=True)

try:

x.to\_csv(r'C:\Users\Aniru\OneDrive\ Documents\Temp.csv')

except:

x.to\_csv(file\_path)

elif message == '/v':

C\_cache = x.CMsgs.copy()

C\_cache.dropna(inplace=True)

iterate(C\_cache)

except:

cont = input("Save messages offline?(y/n): ")

temp\_ser = pd.Series(dtype = 'object')

temp\_df = pd.DataFrame(columns=['SMsgs', 'CMsgs'])

while cont == 'y':

message = pd.Series(input(str('>>')))

temp\_ser = temp\_ser.append(message)

cont=input(str('Continue? (y/n): '))

temp\_df['SMsgs'] = temp\_df['SMsgs'].append(temp\_ser, ignore\_index=True)

x = x.append(temp\_df, ignore\_index=True)

try:

y=pd.read\_csv(r'C:\Users\Aniru\OneDrive\ Documents\Temp.csv')

except:

y=pd.read\_csv(file\_path)

a=check(1, x\_copy, ser1=y.CMsgs)

c=y.CMsgs.copy()

for i in range(len(c)):

if a[i]==True:

c.drop(i, inplace=True)

else: continue

y.CMsgs=c.dropna().reset\_index().drop(columns='index')

x.CMsgs=y.CMsgs.dropna().reset\_index().drop(columns='index')

try:

x.SMsgs=y.SMsgs.append(temp\_df['SMsgs'], ignore\_index=True)

except:

x.SMsgs=y.SMsgs.dropna().reset\_index(). drop(columns='index')

rem\_na(x.CMsgs, x.SMsgs)

x.dropna(thresh=1, inplace=True)

x.set\_index('SMsgs', inplace = True)

try:

x.to\_csv(r'C:\Users\Aniru\OneDrive\ Documents\Temp.csv')

except:

x.to\_csv(file\_path)

try:

del temp\_df, temp\_ser

except: pass

s.close()

exit()

1. The Client-side Program:

import socket as sc

import time

import pandas as pd

import numpy as np

from misc import \*

try:

x=pd.read\_csv(r'C:\Users\Aniru\OneDrive\ Documents\Temp.csv')

except:

file\_path=input(str('Enter CSV File Path: '))

x=pd.read\_csv(file\_path)

SMsgs1=x.SMsgs.dropna()

read(x.SMsgs)

x\_copy=SMsgs1.copy()

x.drop(columns='SMsgs', inplace=True)

x.CMsgs=x.CMsgs.dropna().reset\_index().drop(columns='index')

check(0, x\_copy)

s = sc.socket()

host = input(str("Please enter hostname of the server: "))

try:

port = 8080

s.connect((host, port))

print("Connected to chat server")

while 1:

inc\_msg = s.recv(1024)

inc\_msg = inc\_msg.decode()

print(f"Server: {inc\_msg}")

print("")

msg = input(str(">>"))

if msg not in np.array(('/e', '/h', '/a', '/cs', '/cc', '/v')):

msg = msg.encode()

s.send(msg)

print("Message has been sent")

print("")

else:

if msg == '/e':

conf = input(str('y/n: '))

if conf == 'y':

exit()

elif msg == '/h':

print("Welcome to the Client side Chat Monitor! This utilises csv and socket modules to provide a smooth experience of a private chat! Don't mind the kinks... ")

print("There are just 6 commands in our command line. They are as follows:")

print("/e -- Exits the application")

print("/h -- The command used to get here. It's the command for help, if you didn’t get it yet")

print("/a -- Gives a formatted automatic reply")

print("/cs -- Clears the Server side cached messages")

print("/cc -- Clears the Client side cached messages")

print("/v -- To view the cached messages for the client")

msg = input('>>')

s.send(msg.encode())

print("Message has been sent")

print("")

elif msg == '/a':

msg = 'Client is not looking at his computer! Please repeat the message!'

msg = msg.encode()

s.send(msg)

elif msg == '/cs':

x.SMsgs = x.SMsgs.drop([i for i in range(0, len(x.SMsgs))])

try:

x.to\_csv(r'C:\Users\Aniru\OneDrive\ Documents\Temp.csv')

except:

x.to\_csv(file\_path)

print("All Server side message cache cleared")

msg=input('>>')

s.send(msg.encode())

print("Message has been sent")

print("")

elif msg == '/cc':

x.CMsgs = x.CMsgs.drop([i for i in range(0, len(x.CMsgs))])

try:

x.to\_csv(r'C:\Users\Aniru\OneDrive\ Documents\Temp.csv')

except:

x.to\_csv(file\_path)

print("Client side message cache cleared")

msg=input('>>')

s.send(msg.encode())

print("Message has been sent")

print("")

elif msg == '/v':

conf = input('You want to view the Server message cache?(y/n): ')

if conf == 'y':

S\_cache = x.SMsgs.copy()

S\_cache.dropna(inplace=True)

iterate(S\_cache)

else: continue

msg = input('>>')

s.send(msg.encode())

print("Message has been sent")

print("")

except:

cont=input('Save messages offline?(y/n): ')

temp\_ser = pd.Series(dtype='object')

temp\_df = pd.DataFrame(columns=['SMsgs', 'CMsgs'])

while cont == 'y':

msg=pd.Series(input(str('>>')))

temp\_ser = temp\_ser.append(msg)

cont = input(str('Continue? (y/n): '))

temp\_df['CMsgs'] = temp\_df['CMsgs'].append(temp\_ser, ignore\_index=True)

x=x.append(temp\_df, ignore\_index=True)

try:

y=pd.read\_csv(r'C:\Users\Aniru\OneDrive\ Documents\Temp.csv')

except:

y=pd.read\_csv(file\_path)

a=check(1, x\_copy, ser1=y.SMsgs)

c=y.SMsgs.copy()

for i in range(len(c)):

if a[i]==True:

c.drop(i, inplace=True)

else: continue

y.SMsgs=c.dropna().reset\_index().drop(columns='index')

x.SMsgs=y.SMsgs.dropna().reset\_index().drop(columns='index')

try:

x.CMsgs=y.CMsgs.append(temp\_df['CMsgs'], ignore\_index=True).dropna().reset\_index().drop(columns='index')

except:

x.CMsgs=y.CMsgs.dropna().reset\_index(). drop(columns='index')

rem\_na(x.SMsgs, x.CMsgs)

x.dropna(thresh=1, inplace=True)

x.set\_index('SMsgs', inplace=True)

try:

x.to\_csv(r'C:\Users\Aniru\OneDrive\ Documents\Temp.csv')

except:

x.to\_csv(file\_path)

try:

del temp\_df, temp\_ser

except: pass

s.close()

exit()

1. The Graphing Program:

from misc import \*

import matplotlib.animation as anm

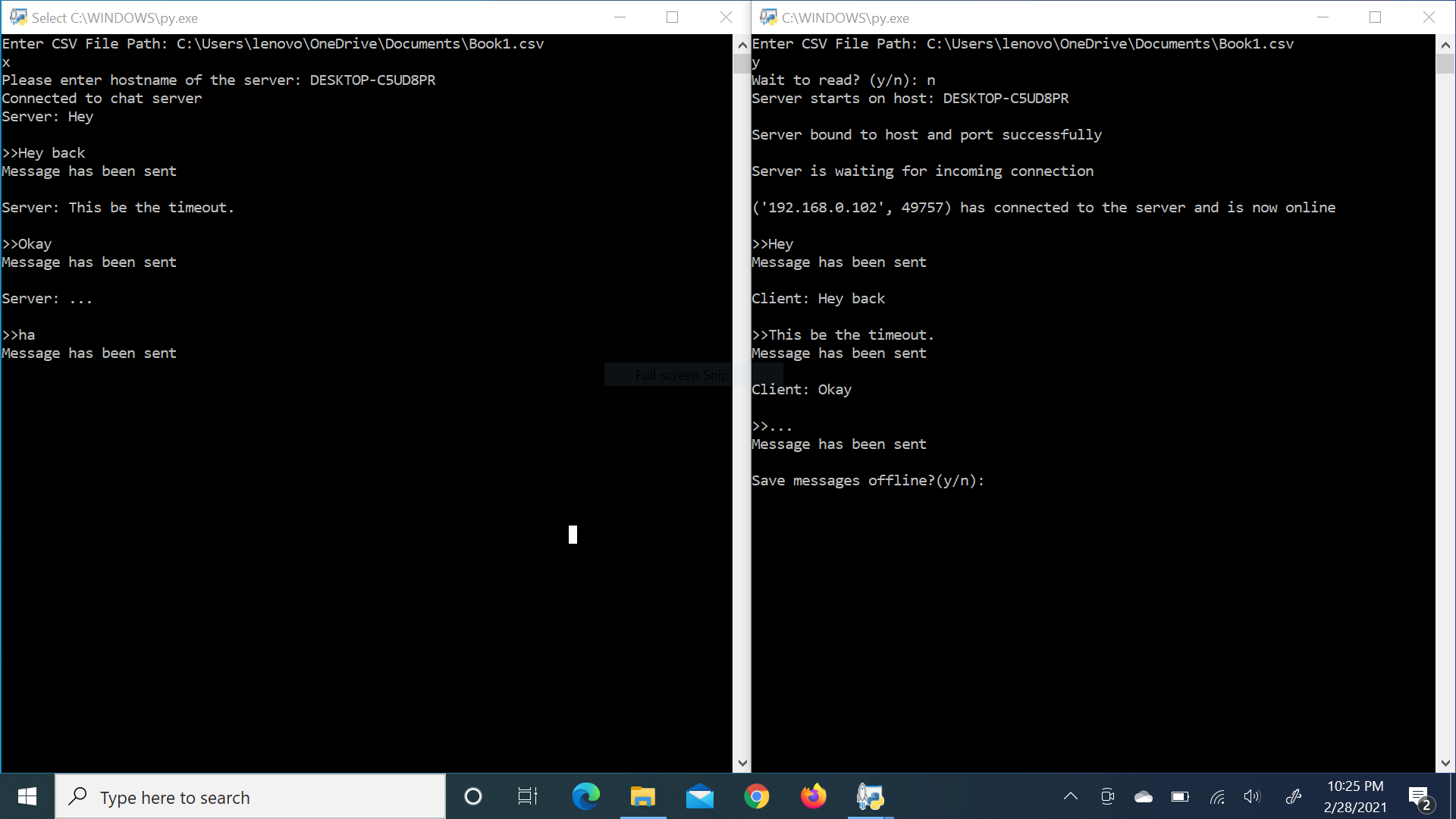
fig, axs = graph\_create\_plot(3)

anim = anm.FuncAnimation(fig, graphing.graph\_plot)

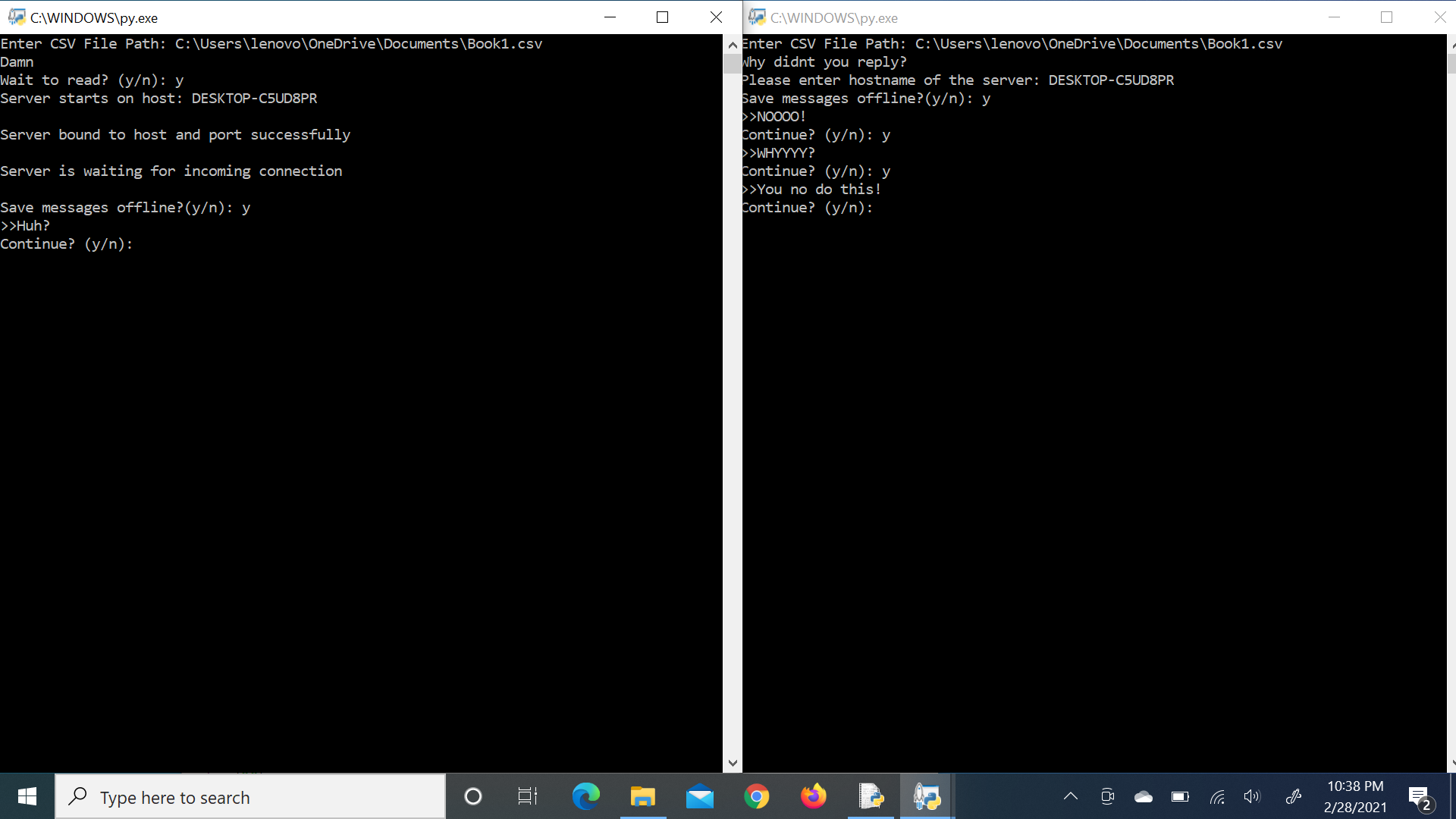
plt.show()

***Output Screens***

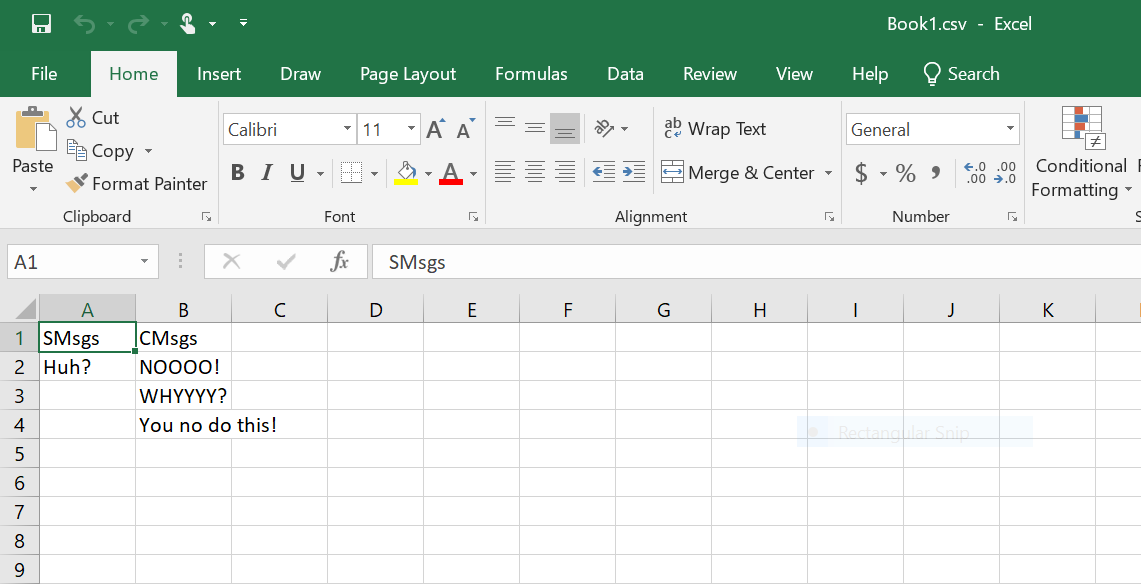
1. Server-Client chat:



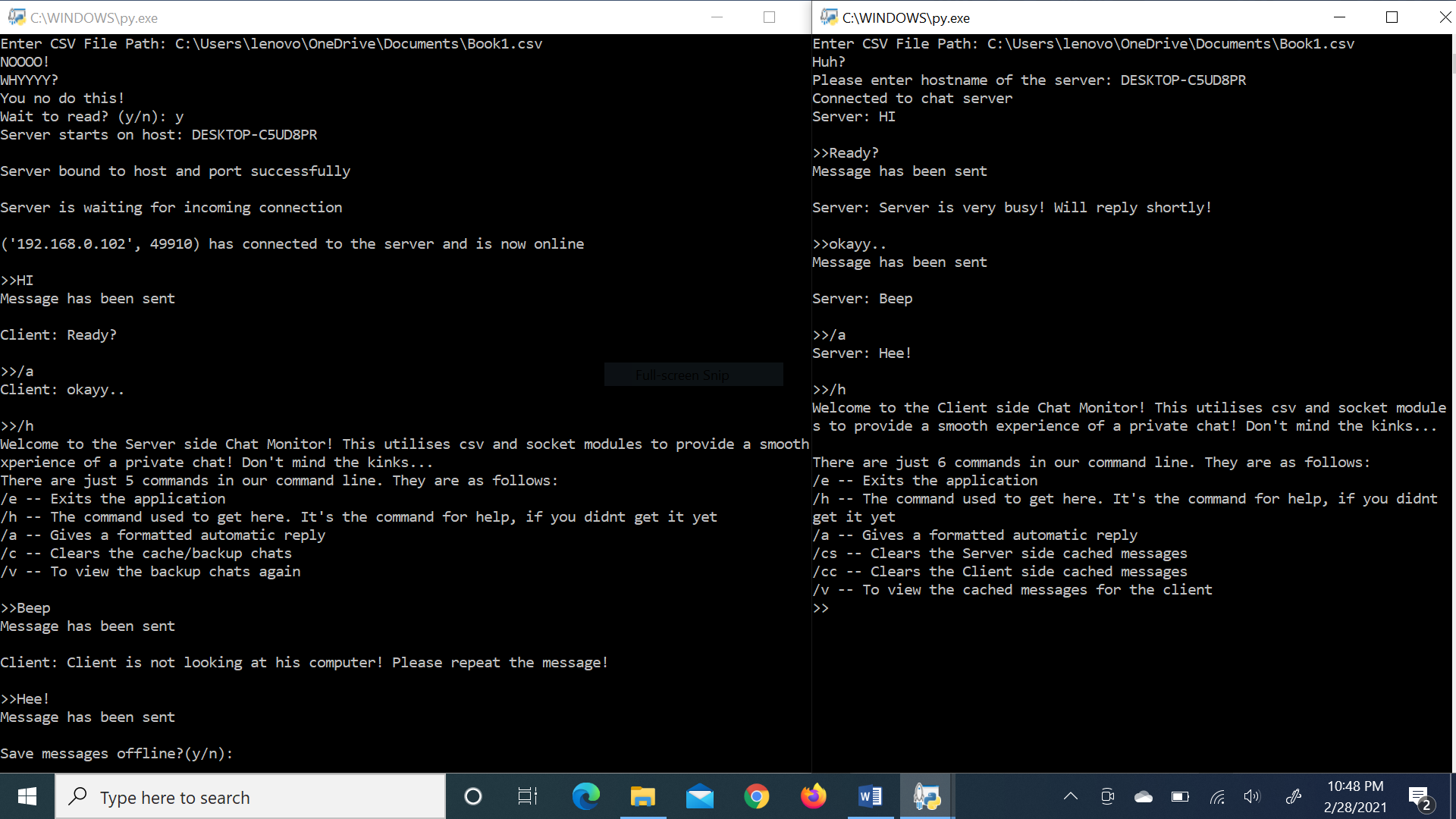
1. Saving Messages from client and server:



1. Csv Screen for Saving the Output:



1. The Command Line Interface:



1. Visualization of Resource Consumption:



***CONCLUSION***

Building a good chat application is way too complex. There exists many facets of the program to define and rewrite to fit in with the other parts. This project on building a simple two-person socket chat provides a good insight into what the basics of a chat application should be, and how to create the necessary backups to support it wherever necessary. The performance graphs included also help in monitoring consumption of resources, further helping in streamlining the program to its absolute best. As a whole, this project helps one understand the structure and components of chat systems in an effective way.

***Bibliography***

1. Informatics Practices with Python for Class XII by Preeti Arora.
2. [www.wikipedia.com](http://www.wikipedia.com) for information about all modules and libraries.
3. docs.python.org for information about embedded libraries.
4. pandas.pydata.org for information about pandas library.
5. numpy.org for information about the NumPy package.
6. matplotlib.org for information on the matplotlib library.
7. [towardsdatascience.com](http://www.towardsdatascience.com) for miscellaneous information.