Homework #6: **sockets** (200 pts)
Submit a compressed (.tgz) file with **source code**

- Your job is to implement *two* socket-based programs in ***Python***: **client.py** and **server.py**.
  - These programs will simulate bank transactions in a *client-server* architecture

- **client.py** has the following requirements:
  - takes two command line arguments: ***port*** and ***input.txt***
    - *port* : server's *port number* that client wants to connect with
      - **server.py** will be on *localhost:port*
    - *input.txt* : text file of bank transactions, with the following format:
      - *Name type amount*
        - *Name* : account holder's name
        - *type* : transaction type, either "**credit**" or "**debit**"
        - *amount* : dollar amount of transaction
  - uses a *stream socket* to connect with **server.py**
  - reads *input.txt* and sends each transaction to **server.py**

- **server.py** has the following requirements:
  - takes one command line argument: ***port***
    - *port* : specifies the *port* to serve on **localhost**
  - creates a *stream socket* on *port* specified
  - *listens* and *accepts* incoming clients (with backlog of 5)
  - uses **threads** to handle multiple clients *simultaneously*
    - each **thread** will receive *client* transactions and modify a *global data structure* accordingly
      - **credit** will increase the account holder's balance
      - **debit** will decrease the account holder's balance
      - invalid transactions are silently discarded
  - maintains a data structure of *accounts*:
    - **Python** dictionary of *key-value* pairs
      - *key* – account holder's name (string)
      - *value* – account balance (decimal number)
    - initialized as empty when **server.py** begins
    - upon each transaction received:
      - if the account is NOT in the data structure, create / update
      - if the account IS in the data structure, update
  - handles **SIGINT** signal
    - upon catching *SIGINT*, **server.py** will write contents of *accounts* data structure to file named **log.txt** and **exit**
    - **log.txt**:
      - sorted alphabetically by *Name*
      - each line will contain *one* **account** and associated **balance**

- **Examples**

```
UNIX > ./client.py
usage: ./client port input_file

UNIX > ./server.py
usage: ./server.py port

UNIX > cat small_input.txt
Marc credit $2
Marc debit $1

UNIX > ./server.py 50000 &
[1] 2400

UNIX > for i in `seq 100`; do
> ./client.py 50000 small_input.txt &
> done
…

UNIX > kill -s SIGINT 2400

UNIX >
[1]+  Done                    ./server.py 50000

UNIX > cat log.txt
Marc $100.00

UNIX >
```

```
UNIX > ls inputs/ | wc –l
50

UNIX> wc -l inputs/input_1.txt
10000    inputs/input_1.txt

UNIX > ./server.py 50000 &
[1] 11093

//spawn 50 concurrent clients
UNIX > for i in `seq 1 50`; do
> ./client.py 50000 inputs/input_$i.txt &
> done
…

//send SIGINT to server.py process
//note: server.py may not exit immediately..
UNIX > kill -s SIGINT 11093

//use “top” command to check if server.py is still running
UNIX> top –p 11093
…

UNIX > head log.txt
Abigail $-5216.49
Addison $-710.37
Aiden $-4633.03
Alexander $-8299.12
Amelia $5118.17
Andrew $8670.54
Anna $5648.74
Aria $-1373.40
Aubrey $5979.45
Audrey $-7390.03

UNIX > tail log.txt
Samuel $7338.88
Scarlett $-2654.16
Sebastian $-6337.16
Sofia $-5146.40
Sophia $9863.92
Victoria $2253.36
William $-9255.19
Wyatt $-4253.82
Zoe $2050.94
Zoey $899.27
```

- Hints:
  - work incrementally..
    - build a simple client / server, then expand
  - **mutex** on *accounts* data structure?
  - test, test, test – especially on **LARGE** files
  - be mindful of socket buffering
    - perhaps wait for *newline* before parsing message?
  - wait for **all threads** to complete before *writing* **log** file
  - Look into Python's **threading** and **socket** modules
  - Have fun! Python is awesome!!