CS131 Python Project

API key:
AIzaSyABVqAoTINdlIqqsK3AFZrcz-aZxfXrL4k

implementing server and client
event driven programming
the part we have to implement is servers need to be able to communicate with each other so the latest client information is on every server

Use only asyncio and standard python libraries, and
- aiohttp
- json

commands to test server:
- telnet
- nc

need flooding algorithm for client info to be propagated to other servers besides the one client communicated with

Questions for Office Hours

1. When should the server stop reading from a client? Should it read all the way until EOF and then process the commands the client sends, or should it maintain a buffer and process each 4 whitespace separated fields greedily as commands?
    1. Or does this not matter?
    2. It seems like it matters since the EOF way relies on the client to close the write end of transport without closing the connection, and then receiving all the responses by the server
    3. **WE SHOULD NOT READ UNTIL EOF: BRETT WILL CLARIFY WITH SAKETH IF WE SHOULD PROCESS GARBAGE AND THROW IT AWAY OR IF WE SHOULD PROCESS 4 AT A TIME**
    4. **UPDATE: read until newline**
2. If we're allowed to stop reading from a client at EOF, we're also allowed to have servers connect to each other, send messages, and close the connections each time a message is sent and reopen it right?
    1. i.e. s1 open, s2 connects to s1, s2 updates s1 by sending msg and closing write end of transport, s1 sends acknowledgement, s2 and s1 disconnect, s1 and s2 reconnect (so that the write ends of transport can be reopened)

3. Can the servers commnicate to each other using any sort of message? For example, can we make up a new command that will allow servers to be handled like clients but have their commands processed differently?
    1. **CAN MAKE UP COMMANDS FOR SERVERS TO COMMUNICATE WITH EACH OTHER**

BE MORE GENERAL ABOUT STANDARD/VALIDATION STUFF
WON'T MONITOR ANYTHING SERVERS SEND TO EACH OTHER

Server as client to another server:
sends:

- AT Goloman +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963997

TODO:
Still need to log the server new connections and dropped connections

Info about asyncio
Generators in Python are functions whose execution can be stopped and resumed at any time.

e.g.
# is going to allow yielding: can yield results at some point midway between the computation
def squaresUntil(x):
  I = 0
  While I <= x:
    Yield i**2
     I++
  Return x

Basically it runs, yields at yield and returns a result, then waits to be run again. It waits to be run again by the event loop!

Gen = squaresUntil(5) # doesn't run the function, returns a generator that represents the function that can be run many times

```
Try:
  While True:
    Print(next(gen))
    doSomething()
  except StopIteration as e:
    print('returned ', e.value)
    # when a generator returns, it throws an exception with the result
```

if yield is inside the function, Python recognizes the function as a generator. Don't have to explicitly label it as a generator

What if we have a generator that uses like another generator?
Use the yield from keyword
e.g.

```
def squares2(x, y):
  a = yield from squaresUntil(x) # delegates work to other generator
  print(a)
  b = yield from squaresUntil(y)
  print(b)
```

basically yield from gets the returned value of the generator
what prints from this run?
0 1 4 2 <- for a
0 1 4 9 16 4

yield from is syntactic sugar for:

```
# try:
#   while True:
#     yield next(gen)
# except StopIteration as e:
#   return e.value
```

Coroutines
Cooperative non-preemptive multitasking:
The coroutine decides when it gets to stop
The caller is able to send information back into the coroutine in the middle of its execution

They use yield to communicate with their caller

```
def echo():
```

```
  a = None
  while True:
    # set a to whatever is passed into the coroutine
    # have to stop own computation: yield something
    a = (yield)
    print(a)

cor = echo() # create a coroutine object
next(cor)       # allow echo to step through computation until it yields control back to caller function
I = 0
While True:
  cor.send(i)   # can send any object, is basically IPC
  I = I + 1
```

Event loop schedules all the coroutines (called callbacks on the event loop)
When a coroutine stops, we delegate control back to the loop (caller), and it can run the next thing on the loop

Writing an event loop

```
scheduled = []
while True:
  if event.happened():
    event = getEvent()
    cor = handle.event(event)
    next(cor)
    scheduled.append(cor)
  run_next(scheduled)

def handle_event(event):
  m = event.message
  yield from tcp_send(m)
  return
```

await is the same thing as yield from, but it can only happen in async function
async functions are basically coroutines
theres syntactic sugar that you have to follow to ensure that you have async

change handle_event into async function:

```
async def handle_event(event):
```

```
m = event.message
await tcp_send(m)
return
```