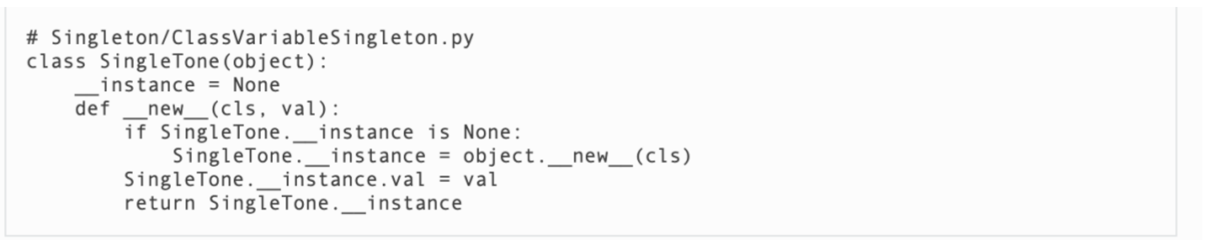
Peer to Peer Network: How my program works

1.Initialisation

To initialise the existence of peers, I have set up a main function in which initialises the Ping set up depending on whether it is initialising, joining, storing or requesting. I store the peer's instance specific information in a instance class called 'Store'. This is a single instance class (Singleton) meaning that I can only create one object/copy from it unlike normal classes in which can create multiple copies. The variables in which I store in the Store singleton are things like the peer's id, its join status, ping interval, the peerInstancePing object (functions establishing the UDP and TCP connection for setting up a peer, joining a peer and departing a peer, as well as the get & set functions of Peer's storing information (predecessors, files, successors)

To following code I have borrowed from the web to set up the singleton:



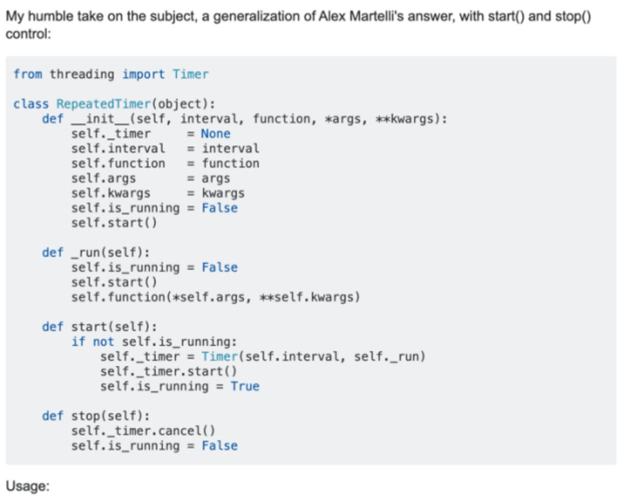
```
# Singleton/ClassVariableSingleton.py
class SingleTone(object):
    __instance = None
    def __new__(cls, val):
        if SingleTone.__instance is None:
            SingleTone.__instance = object.__new__(cls)
        SingleTone.__instance.val = val
        return SingleTone.__instance
```

https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Singleton.html

2.Ping successors

I ping successors using a UDP connection. I have a UDP server and client object stored in 'ping.py'. Since I need to run processes at the same time, I have used threading to ensure that I can run multiple function calls at the same time. Tasks such as sending a ping from client to server involves taking time to execute the task. If I did not use threading, the program would have to run through each line in the function before it begins to execute the code for the UDP server. This defeats the purpose of being able to establish a connection, as the server and client program should be run in parallel time.

The threading code in which I have borrowed from the web that allows UDP Client to continuously send requests at an interval is the following:



My humble take on the subject, a generalization of Alex Martelli's answer, with start() and stop() control:

```
from threading import Timer

class RepeatedTimer(object):
    def __init__(self, interval, function, *args, **kwargs):
        self._timer     = None
        self.interval   = interval
        self.function   = function
        self.args       = args
        self.kwargs     = kwargs
        self.is_running = False
        self.start()

    def _run(self):
        self.is_running = False
        self.start()
        self.function(*self.args, **self.kwargs)

    def start(self):
        if not self.is_running:
            self._timer = Timer(self.interval, self._run)
            self._timer.start()
            self.is_running = True

    def stop(self):
        self._timer.cancel()
        self.is_running = False
```

Usage:

https://stackoverflow.com/questions/3393612/run-certain-code-every-n-seconds

I have also implemented thread condition in the server code for both TCP and UDP. A condition represents the state of change in the application, and the thread will wait for a given condition or signal that the condition happened.

I have set a timeout condition that is 1 second to check

3.Peer Joining

Starting from the known peer, I have a while loop that loops through the peers in the network and send messages via TCP connection. It checks the condition of whether the current known peer's first successor is greater than the peer that wants to join. I check this so I know which condition to slot the known peer and make updates in the corresponding peer's terminal. I am able to make updates in updating the newest successors because I have stored the peer's instance information regarding successors/predecessors in a class in the file called PeerMap. I created setters and getters of the arrays and called these functions I created in PeerMap class in the p2p.py file's class called peerInstancePing. Once it has successfully identified which peer is to be added and updated, I send over a UDP Client ping request to make sure that the peer it has newly added is successfully pinging it's connection to the server.

I have borrowed the following set up for UDP/TCP connection from the following websites:
TCP: https://pymotw.com/2/socket/tcp.html
UDP: https://wiki.python.org/moin/UdpCommunication

4.Peer departure(Graceful)

Once it receives a user input of "Quit", this will create a condition to start the successor updating process of the departed terminal. During UDP pinging, if there has been a message received by the client by the server, I store server's predecessor (which is the client) in an array called predecessors. This array is stored in PeerMap. Since I have stored the first and second successor of the current peer as well as the predecessor, the peer that is departing sends a message over TCP to the previous peer informing about its new successors that it will have. Once it finishes updating its new successors, it will now get rid of the old successors it previously had and update it with its new successors and start UDP pinging to ensure a connection. Once the client receives a message it has been updates, it will now update the next previous peer using the same method.

5.Peer departure(Abort)

This follows the same method as Peer departure, however instead of a typed user input, I have created an exception to handle keyboard interrupts. Therefore, once a keyboard interrupt happens, the peers pinging to the aborted peer will have socket timeouts which notifies to start the successor updates. Note: Although I have implemented this and it does print out correct statement, it doesn't seem to wait for the timeout.

6.Data insertion

Once it receives a user input of "Store *filename"*, I process the filename. I search through the current directory to make sure the filename exists in the directory before proceeding. If the file doesn't exist, I return an error saying "File does not exist" so that it will not execute the process of forwarding the file. If the file does exist, then I begin to use the hash function to produce the key of which peer the file should be stored at. . Since the file produces a hash of *n* stored at the peer closest successor of *n* in the circular DHT, I iterate through the peer and its successors, and check the condition when the port num of which the file should be inserted at is less than or equal to the current port number in which is in the loop. If it is less, it will keep forwarding until it meets the necessary conditions to break the loop. I store the file and append it to a list to keep track of the files stored in that Peer in the class PeerMap.

7.Data retrieval

Once it receives a user input of "Request *filename"*, I process the filename. I search through the current directory to make sure the filename exists in the directory before proceeding. If the file doesn't exist, I return an error saying "File does not exist" so that it will not execute the process of forwarding the file. Similar to data insertion, I loop through the next successors until it meets the condition where the hash of *n* is less than or equal to the current port number. It then appends and stores this file in the list. Next I transfer the file contents over TCP by bytes. A new file is created to store the transferred bytes over tcp.

Possible improvements

Improvements to the code would be:

- Better calculation of timeout time using the estimated RTT or calculating the probability of deciding if the peer is lost.
- Better implementation of if a peer is long, such that if the peer is uncertain of its live status, then immediately re-pinging the peer instead of waiting for the next interval
- Implementing more detailed complex case scenarios in transferring a file whilst another peer is joining
- Using ACKbit, FIN, seq,headings , checksum in sending the messages to organise and separate data better for better reliable data transfer and reducing file corruption
- Using pipeline protocols in file transfer and sending messages for overall better general error handling (caused by timeouts) such as Go back N for ensuring entire file transfer is complete even when there is a loss in the some of the sending packet by reducing all the data and ignoring duplicates.
- Better file re-transmission using fast-transmission technique when time out periods are long (such as the abrupt) to check whether the peer indeed dead or alive in a faster shorter amount f time.