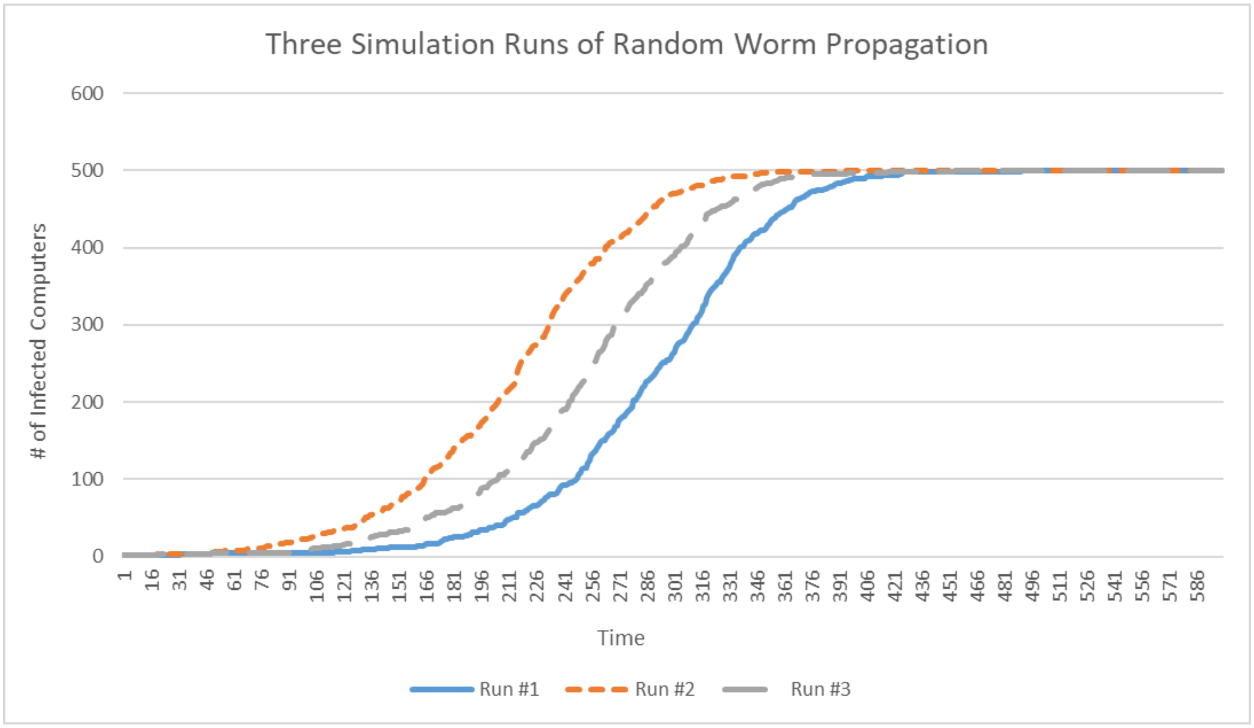
Worm Propagation Report

Simulated random worm infection (600 total time ticks):



Test Run #1 finished at time tick 491

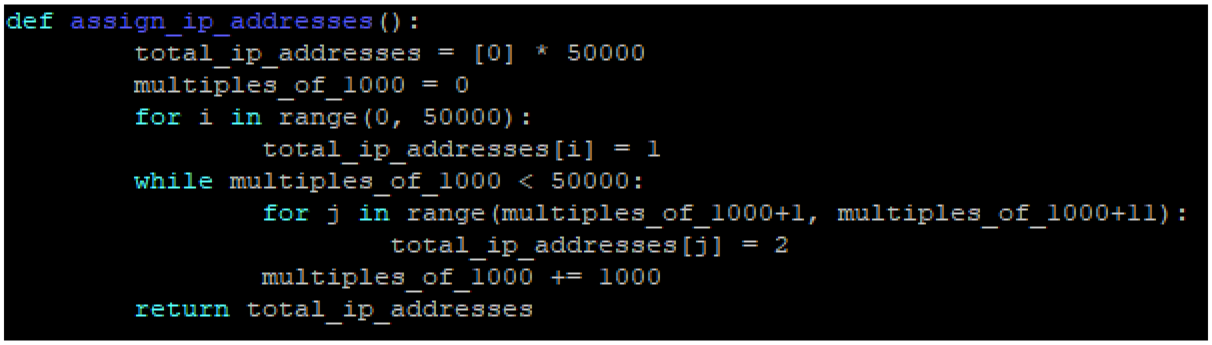
Test Run #2 finished at time tick 391

Test Run #3 finished at time tick 454

The program random_scanning_worm.py was executed in order to obtain the results displayed above.

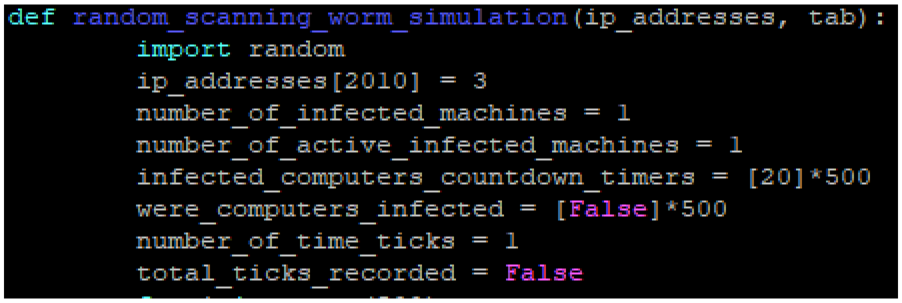
```
#!/usr/bin/env python3
```

I've included a shebang for random_scanning_worm.py so it can be executed within the Eustis machine as the command "./random_scanning_worm.py".



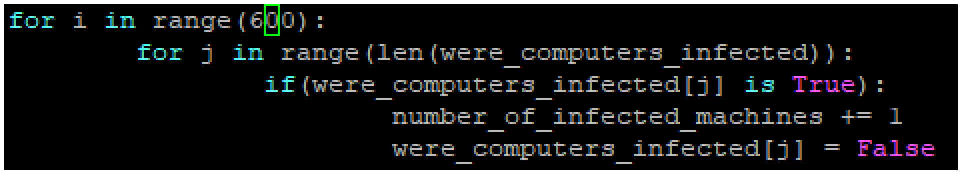
```python
def assign_ip_addresses():
        total_ip_addresses = [0] * 50000
        multiples_of_1000 = 0
        for i in range(0, 50000):
                total_ip_addresses[i] = 1
        while multiples_of_1000 < 50000:
                for j in range(multiples_of_1000+1, multiples_of_1000+11):
                        total_ip_addresses[j] = 2
                multiples_of_1000 += 1000
        return total_ip_addresses
```

In the function “assign_ip_addresses()”, we initialize our array “total_ip_addresses” with a size of 50,000 to simulate our mini-internet. We create another variable called “multiples_of_1000” and set it to 0. Before utilizing “multiples_of_1000”, we initialize all IP addresses with the value of 1, which means the IP address is immune to infection. However, we want specific IP addresses to be susceptible to infection, so we utilize our “multiples_of_1000” variable within the following double loop. While “multiples_of_1000” has not reached 50,000 signaling the end of the total IP addresses, then for each 1000+1 entry up to 1000+11 entry, assign this IP address a value of 2. This means that IP addresses 1-10 will be assigned a value of 2, and for every 1000 IP addresses, the process will repeat for 1001-1010, and so forth. We now see “multiples_of_1000” being incremented by 1000 to carry out the previously explained execution. After “total_ip_addresses” has its values assigned and finalized, we return the array outside the function to an awaiting variable.



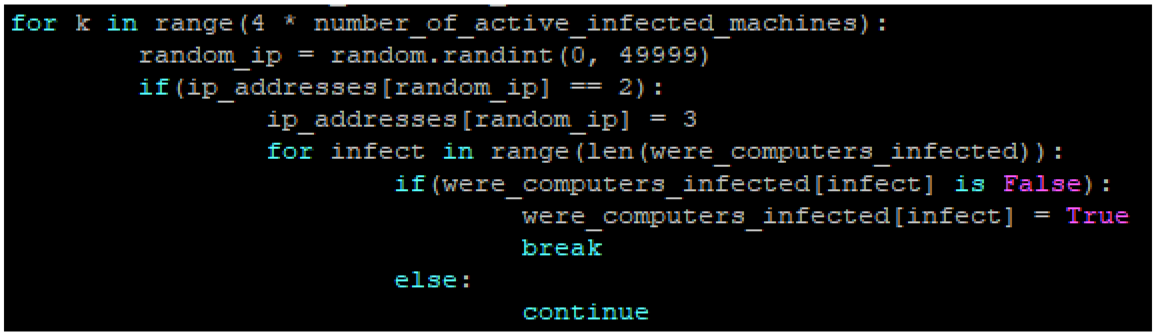
```
def random_scanning_worm_simulation(ip_addresses, tab):
        import random
        ip_addresses[2010] = 3
        number_of_infected_machines = 1
        number_of_active_infected_machines = 1
        infected_computers_countdown_timers = [20]*500
        were_computers_infected = [False]*500
        number_of_time_ticks = 1
        total_ticks_recorded = False
```

The function “random_scanning_worm_simulation(ip_addresses, tab)” is a large function so the following explanation will be broken down into sections to fully clarify its functionality. We pass in our total IP addresses array into this function along with a “tab” character, which will create an indented space when we run our 2nd and 3rd tests. We import the random module and set the IP address at index 2010 to be our first infected machine. We tally our total number of “number_of_infected_machines” to 1, this will help inform us in the future when we’ve infected all 500 susceptible IP addresses. We also set “number_of_active_infected_machines” to 1; this variable will keep track of all active, scanning machines working to infect others. “infected_computers_countdown_timers” is initialized with a size of 500, this will store and keep track of when the compromised IP addresses become active in order to infect other machines. The variable “were_computers_infected”, is initialized with values of false and given a size of 500, ensuring that when an IP address is infected we keep a record of the instance. We create a variable called “number_of_time_ticks” and set it to 1, this will keep track of the total time it took to infect 500 total, susceptible IP addresses. Our variable “total_ticks_recorded” is set to false, which will in turn, be set to true after the “number_of_time_ticks” has been recorded.



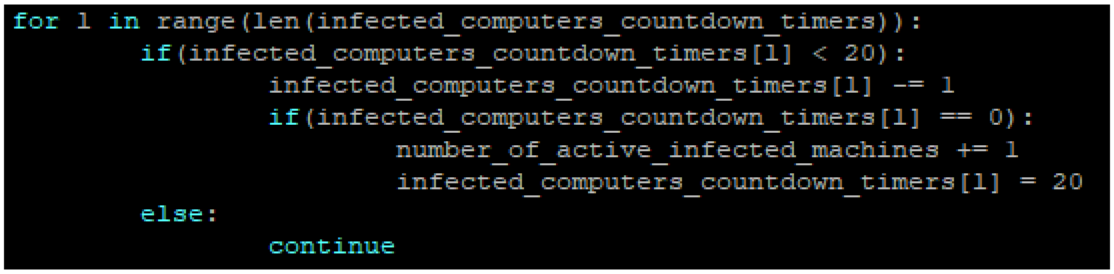
```
for i in range(600):
        for j in range(len(were_computers_infected)):
                if(were_computers_infected[j] is True):
                        number_of_infected_machines += 1
                        were_computers_infected[j] = False
```

In the next code block in the function, our for “i” loop will signify our 500 trials for each simulated run. The for “j” loop will traverse the entire length of the “were_computers_infected” list and the following conditional will check throughout the array for a “true” value. If a value of “true” is found, we add an additional infected machine to our “number_of_infected_machines” list. We then set the value where we found our “true” value to “false” within the “were_computers_infected” list in order to reset it back to its initial state to avoid recounting the same infected machine.



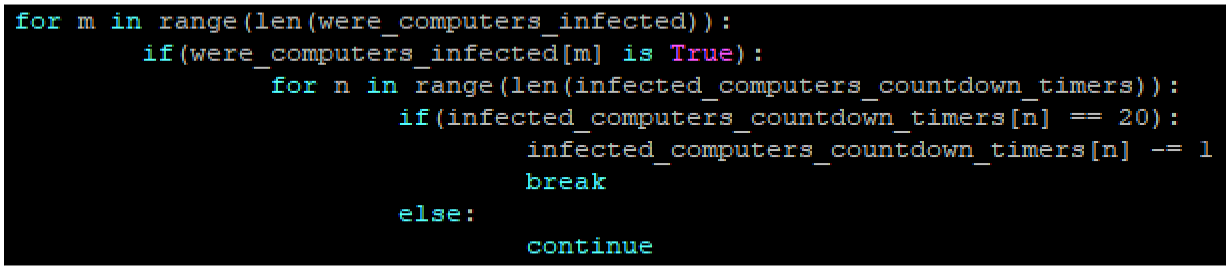
```
for k in range(4 * number_of_active_infected_machines):
        random_ip = random.randint(0, 49999)
        if(ip_addresses[random_ip] == 2):
                ip_addresses[random_ip] = 3
                for infect in range(len(were_computers_infected)):
                        if(were_computers_infected[infect] is False):
                                were_computers_infected[infect] = True
                                break
                        else:
                                continue
```

The next code block in the function will simulate the infected machines actively pinging 4 IP addresses in order to infect and spread the worm out to additional machines. The for “k” loop will simulate 4 IP pings per infected machine in an “activated” state. For each of these ping attempts, we assign “random_ip” to a random integer between 0 and 49,999 to avoid going outside the bounds of our “ip_addresses” list. We’re able to generate a random integer thanks to the “random” module we imported earlier at the beginning of the function. The following conditional check will verify if the “ip_addresses” list at the index specified by our randomly generated integer value is found to be in a “susceptible” state. If the check returns a “true” value, then set the IP address at the index that it was found to 3, indicating the machine is now in an “infected” state. The next for “infect” loop, will traverse the entire “were_computers_infected” list. Our following conditional checks to see if we’ve reached a point within the list that contains a value of “false”. When this specific position is found, we set it to “true”, in order to keep a record of the machines we’ve infected throughout this entire IP scanning process. After we set this record to a value of “true”, then break out of the “infect” loop, only one machine was infected so there’s no need to keep traversing throughout the list. Otherwise, we “continue” to loop throughout the list until we finally find a “false” value to set “true” for our record-keeping purposes.



```
for l in range(len(infected_computers_countdown_timers)):
        if(infected_computers_countdown_timers[l] < 20):
                infected_computers_countdown_timers[l] -= 1
                if(infected_computers_countdown_timers[l] == 0):
                        number_of_active_infected_machines += 1
                        infected_computers_countdown_timers[l] = 20
        else:
                continue
```

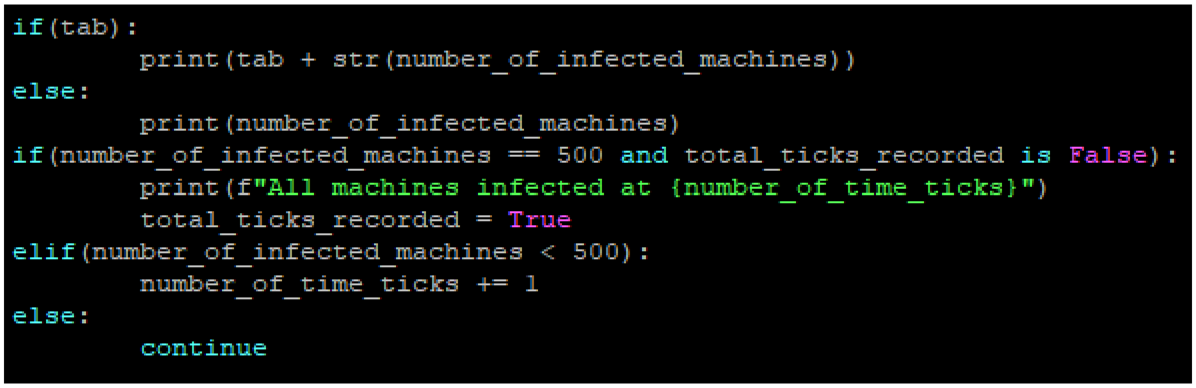
The following code block represents countdown timers for our “infected” machines before they mutate into an “active” state after the countdown reaches 0 from its starting point of 20. We start with the for “l” loop which will traverse the entirety of the “infected_computers_countdown_timers” list. If the “infected_computers_countdown_timers” contains a location where the value is less than 20, then decrement by 1, representing that this “infected” machine is one time tick closer to being switched into the “active” state. Furthermore, after immediately decrementing by 1 we check to see if this “infected” machine has reached a countdown value of 0. If the “infected” machine’s countdown is 0, then add to the “number_of_active_infected_machines”, symbolizing that we’ve gained another “active” infected machine that will proceed to infect others for each time tick from now on. We also set this point in the list back to a countdown value of 20 for the next “infected” machine and to avoid any situations where we could potentially recount the same “infected” machine. Finally, if there isn’t an instance where the countdown value is less than 20, then “continue” throughout the list until we find a value that fits this scenario.



```
for m in range(len(were_computers_infected)):
        if(were_computers_infected[m] is True):
                for n in range(len(infected_computers_countdown_timers)):
                        if(infected_computers_countdown_timers[n] == 20):
                                infected_computers_countdown_timers[n] -= 1
                                break
                        else:
                                continue
```

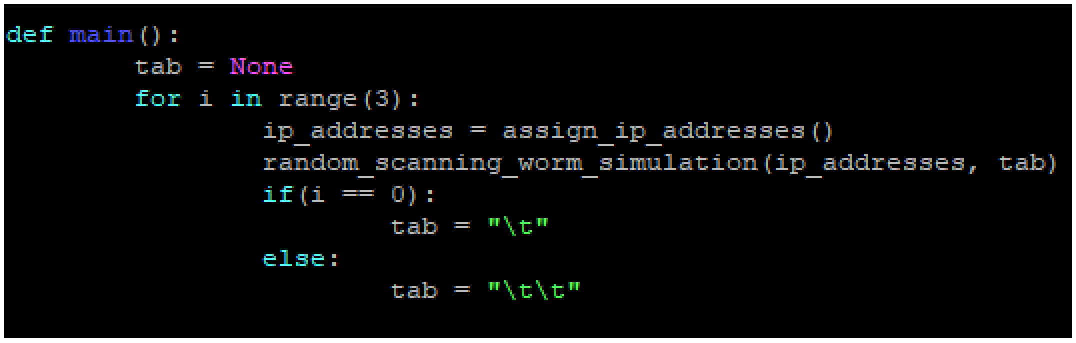
This next code block represents how our countdown timers begin to decrement for a newly “infected” machine. In the for “m” loop, we traverse the length of the “were_computers_infected” list. We use our conditional to find if there happens to be a value of “true” within this list, representing that a machine was “infected” in the last time tick. If this was indeed the case, then we being our for “n” loop to traverse the length of our “infected_computers_countdown_timers” list. The next conditional is for the purpose of finding a position within this list that contains a value of 20. Once we find a value of 20, we decrement by one, initiating our countdown timer which will continue its countdown in a previously aforementioned code block. After we initiate the countdown for our newly “infected” machine, we “break” out of this current loop to avoid any accidental recounts, and continue to traverse throughout the “were_computers_infected” list for additional instances of infected computers in order to initialize further countdown sequences. If we don’t immediately come across a value of

20, meaning that the countdowns for other machines are already in progress, then “continue” throughout the list until we come to a value of 20 in order to initiate a new countdown.



```
if(tab):
        print(tab + str(number_of_infected_machines))
else:
        print(number_of_infected_machines)
if(number_of_infected_machines == 500 and total_ticks_recorded is False):
        print(f"All machines infected at {number_of_time_ticks}")
        total_ticks_recorded = True
elif(number_of_infected_machines < 500):
        number_of_time_ticks += 1
else:
        continue
```

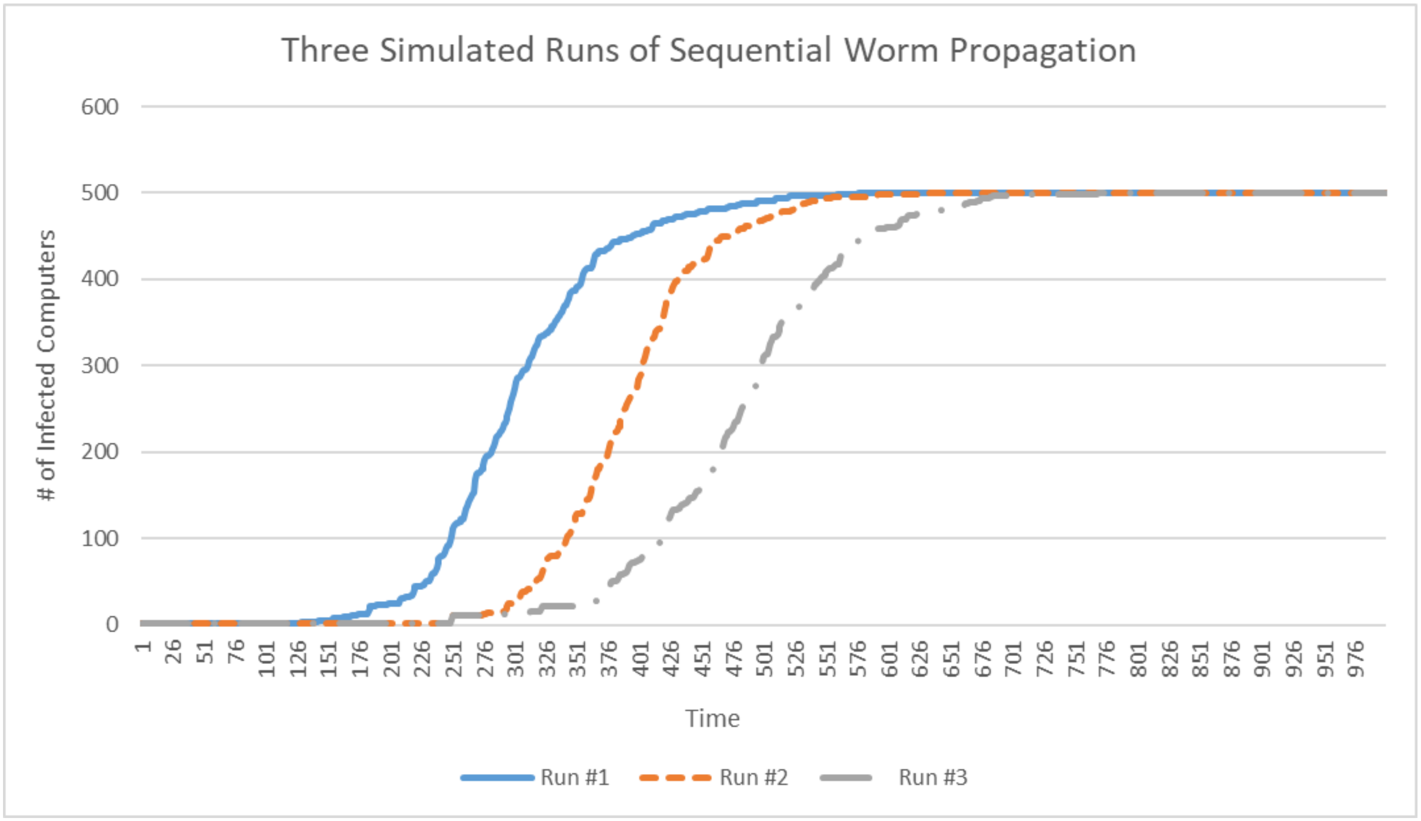
This is the final code block within the “random_scanning_worm_simulation” function. We check to see if there’s a “tab” present before printing the total number of infected machines to the console. This is to help separate our test data during our 2nd and 3rd runs. Otherwise, we print the “number_of_infected_machines” for our current time tick as normal. For the next conditional, we check if we’ve finally infected all 500 “susceptible” machines and if we printed out our final results for total time ticks. If this conditional is met, we print out the “number_of_time_ticks” for our data records and set the “total_ticks_recorded” to “true” so we can avoid printing out redundant, erroneous data. If we find that our total number of infected machines is less than 500, then continue to increment our total number of time ticks to represent that our test run is still ongoing. But when we do get to the point where we’ve recorded our total “number_of_time_ticks”, there’s no longer any need to increment “number_of_time_ticks” so we can “continue” on without dealing with this conditional again.



```
def main():
        tab = None
        for i in range(3):
                ip_addresses = assign_ip_addresses()
                random_scanning_worm_simulation(ip_addresses, tab)
                if(i == 0):
                        tab = "\t"
                else:
                        tab = "\t\t"
```

Our “main” function will be dealing with the execution and processing of our program. We initialize our “tab” variable to “None”, this is the case for now since our 1st test run will be initiated. We set our for “i” loop to a range of 3, representing all 3 of our test runs. We initialize our total number of IP addresses for each of our simulated runs. Now we execute “random_scanning_worm_simulation” with our IP addresses and “tab” if needed. We create a conditional after our first run concludes, when the first run is over we initialize the tab symbol “\t” within our tab variable. When our second run reaches a finished state, we initialize this same “tab” variable with an additional tab sequence, creating “\t\t” within the variable. This concludes the functionality for the random_scanning_worm.py program.

Simulated sequential worm infection (1000 total time ticks):



Test Run #1 finished at time tick 577

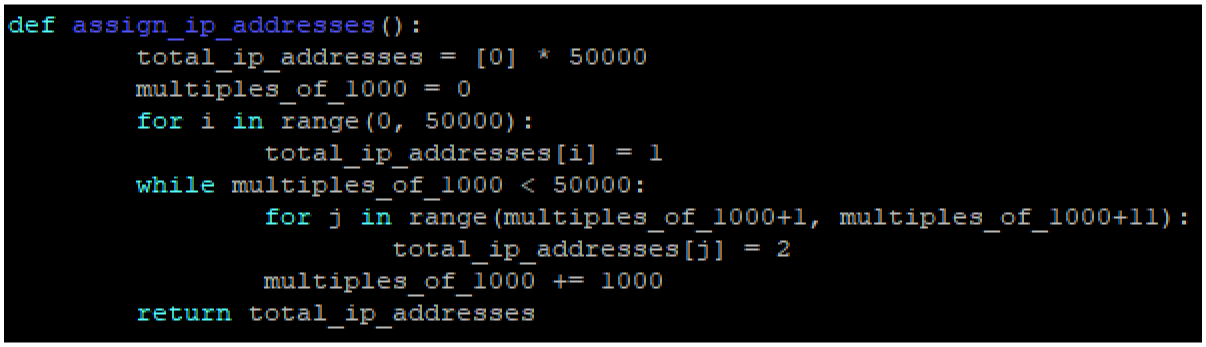
Test Run #2 finished at time tick 628

Test Run #3 finished at time tick 769

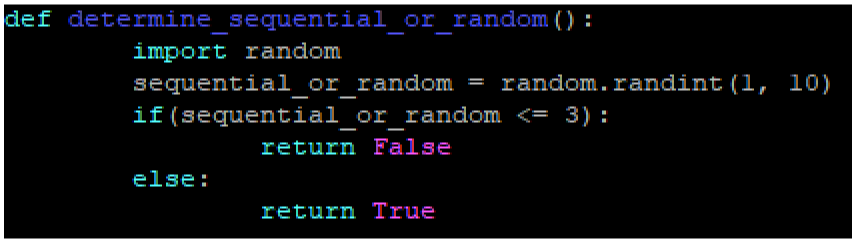
The program sequential_scanning_worm.py was executed in order to obtain the results displayed above.

```
#!/usr/bin/env python3
```

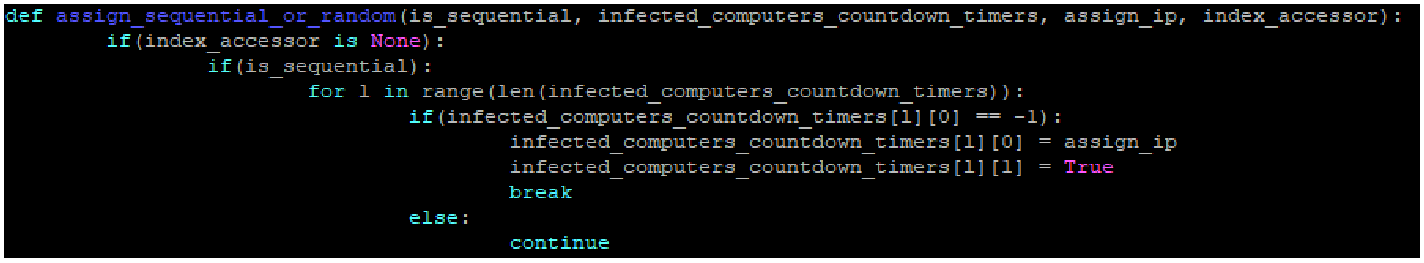
I've included a shebang for sequential_scanning_worm.py so it can be executed within the Eustis machine as the command "./sequential_scanning_worm.py".


```
def assign_ip_addresses():
        total_ip_addresses = [0] * 50000
        multiples_of_1000 = 0
        for i in range(0, 50000):
                total_ip_addresses[i] = 1
        while multiples_of_1000 < 50000:
                for j in range(multiples_of_1000+1, multiples_of_1000+11):
                        total_ip_addresses[j] = 2
                multiples_of_1000 += 1000
        return total_ip_addresses
```

This function was repurposed from the random_scanning_worm.py program and contains the same functionality as previously described.

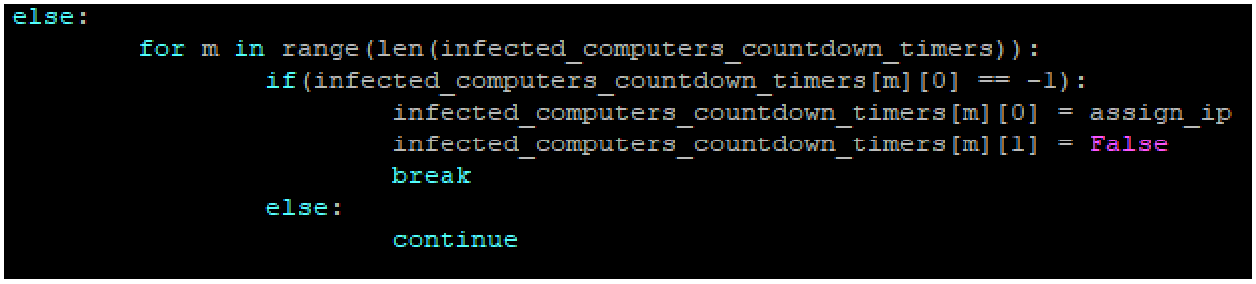

```
def determine_sequential_or_random():
        import random
        sequential_or_random = random.randint(1, 10)
        if(sequential_or_random <= 3):
                return False
        else:
                return True
```

This function's main purpose is to determine whether an "infected" IP address will have a "random scanning" or "sequential scanning" behavior when attempting to infect other machines. We import the "random" module to determine the aforementioned behavior. We create a variable called "sequential_or_random" and set it to a value from 1-10. We check to see if the value is less than or equal to 3, since the randomly generated integer can only be 1-3 to satisfy this condition, it has a 30% chance of returning "false". If it happens to be 4-10, given the fact it has a 70% to do so, then return the value "true".



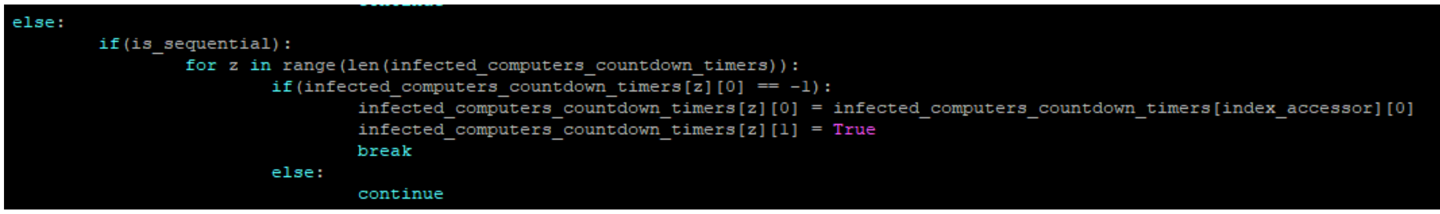
```
def assign_sequential_or_random(is_sequential, infected_computers_countdown_timers, assign_ip, index_accessor):
        if(index_accessor is None):
                if(is_sequential):
                        for l in range(len(infected_computers_countdown_timers)):
                                if(infected_computers_countdown_timers[l][0] == -1):
                                        infected_computers_countdown_timers[l][0] = assign_ip
                                        infected_computers_countdown_timers[l][1] = True
                                        break
                                else:
                                        continue
```

The following function “assign_sequential_or_random” will need to be broken into parts due to its complexity and for the sake of clarity. This function will assign the infected IP address to our “infected_computers_countdown_timers” double-list to keep track and organize all of the infected “sequential scanning” and “random scanning” behavior infected machines. At a specific point within a future code block, “index_accessor” will be explained in further detail but its main purpose is to access the contents of our double-list “infected_computers_countdown_timers”. If the “index_accessor” is null and wasn’t passed into the array, then we continue on to our next conditional. This conditional checks if the IP address to store into our list will take on a “sequential behavior” before being assigned to a spot in “infected_computers_countdown_timers”. If the IP address will be “sequential scanning”, then traverse throughout the length of our double-list until we find a vacant spot with a value of -1. If the value we come across within the list is -1, meaning we found a vacancy, then overwrite this location’s -1 value to the corresponding IP address number from the 50,000 total IP addresses. We then set the boolean value contained at this location to “true”, representing a “sequential scanning” IP address. After we assign this “sequential scanning” IP address within our “infected_computers_countdown_timers” double-list, then “break” out of the loop to avoid any accidental recounts. Otherwise, “continue” until we find a vacant spot within the double-list.



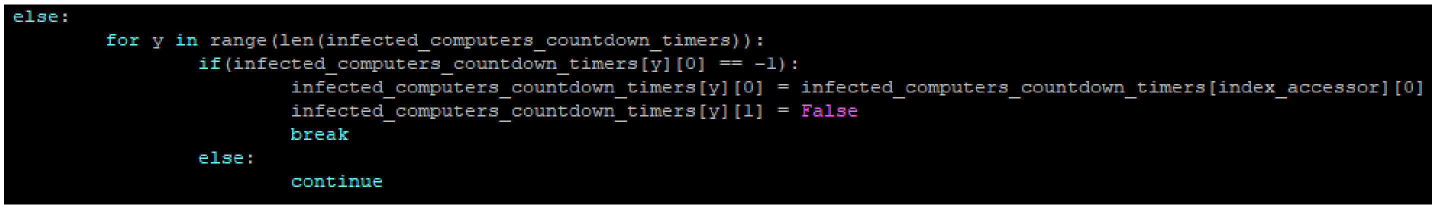
```
else:
        for m in range(len(infected_computers_countdown_timers)):
                if(infected_computers_countdown_timers[m][0] == -1):
                        infected_computers_countdown_timers[m][0] = assign_ip
                        infected_computers_countdown_timers[m][1] = False
                        break
                else:
                        continue
```

This code block functions in the same way as the formerly mentioned code block. The only difference this time around is assigning a value of "false" to designate a "random scanning" behavior when we assign the IP address to this location within the double-list.



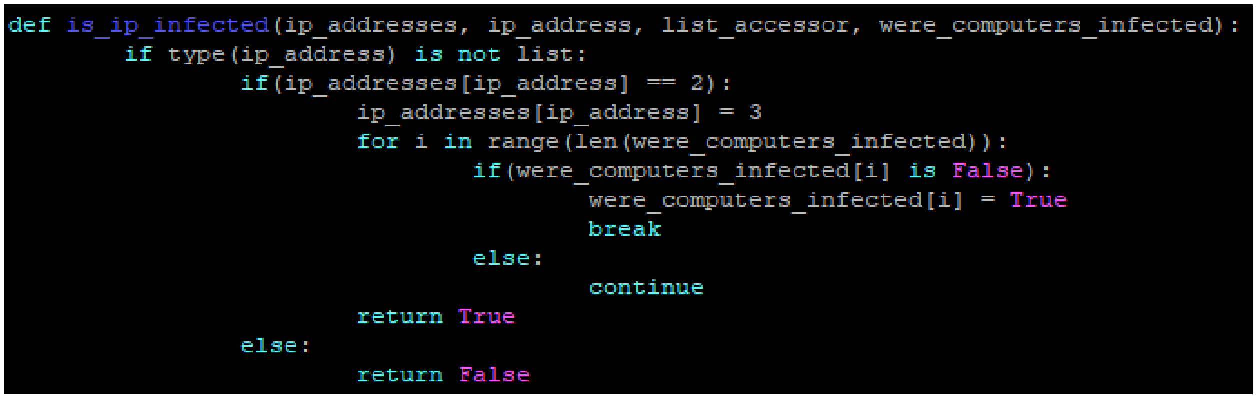
```
else:
        if(is_sequential):
                for z in range(len(infected_computers_countdown_timers)):
                        if(infected_computers_countdown_timers[z][0] == -1):
                                infected_computers_countdown_timers[z][0] = infected_computers_countdown_timers[index_accessor][0]
                                infected_computers_countdown_timers[z][1] = True
                                break
                        else:
                                continue
```

The following code block is the "else" statement that corresponds with the conditional checking for an "index_accessor", meaning this "else" clause will only run when it detects a value in the "index_accessor" variable. This only occurs when a "sequential scanning" behavior IP address happens to infect an IP address in a "susceptible" state. The following conditional and for "z" loop work as similarly described in the aforementioned code blocks. When assigning our "sequential scanning" behavior IP address a spot in our double-list, we take into account that it was infected by an IP that was also stored in our double-list. We access the point where this infected IP address infected another IP address in a "susceptible" state and store that IP address in a new location within the double-list. This process will be explained in further detail later in the report.



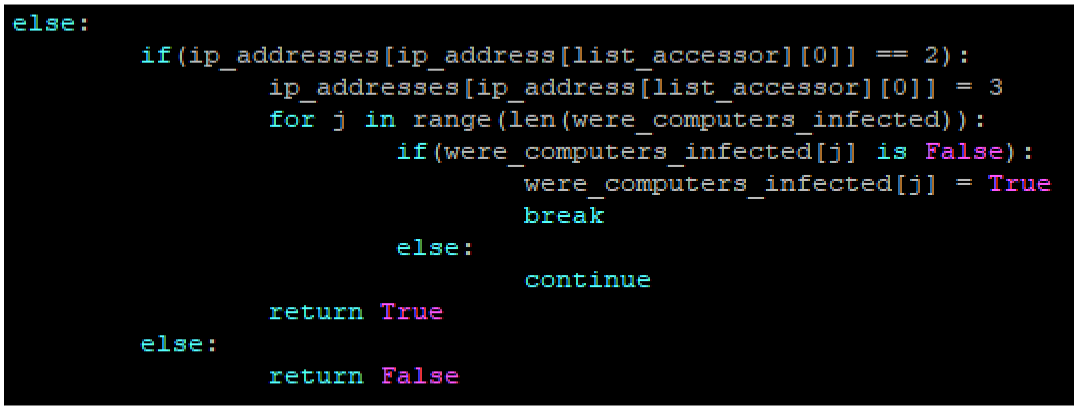
```
else:
        for y in range(len(infected_computers_countdown_timers)):
                if(infected_computers_countdown_timers[y][0] == -1):
                        infected_computers_countdown_timers[y][0] = infected_computers_countdown_timers[index_accessor][0]
                        infected_computers_countdown_timers[y][1] = False
                        break
                else:
                        continue
```

The following code block works in conjunction with the previously mentioned code block and will assign a newly infected IP address with the "random scanning" behavior this time around.



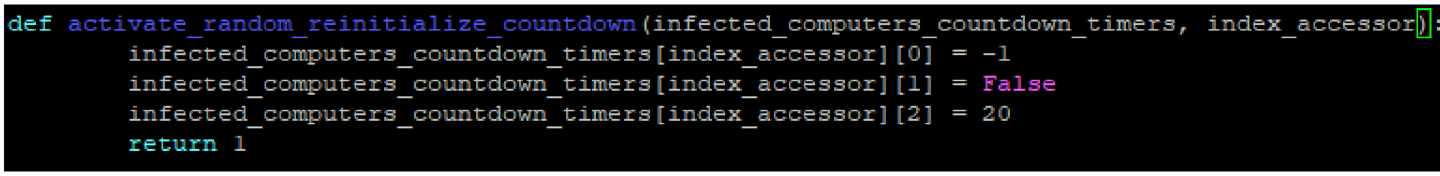
```
def is_ip_infected(ip_addresses, ip_address, list_accessor, were_computers_infected):
        if type(ip_address) is not list:
                if(ip_addresses[ip_address] == 2):
                        ip_addresses[ip_address] = 3
                        for i in range(len(were_computers_infected)):
                                if(were_computers_infected[i] is False):
                                        were_computers_infected[i] = True
                                        break
                                else:
                                        continue
                        return True
                else:
                        return False
```

The following function checks to if an IP address was infected or not and returns the corresponding boolean value for either case. We pass in our total IP addresses array, our index point that can take the form of either an integer or a list for our variable "ip_address". Another index accessor gets passed in as "list_accessor" and the "were_computers_infected" list of booleans gets passed in as well. We check to see if the "ip_address" variable is in the form of a list, and if not, check this specific IP address within our total IP addresses list to see if it is in a "susceptible" state. If so, change the value to reflect that it is now "infected" and loop through our "were_computers_infected" list to check for a vacant spot to assign "true" for our records. Once we find this spot, then "break" out of the loop to avoid any accidental recounts. Otherwise, continue throughout the list until we find an empty spot. After assigning the value of "true" to our "were_computers_infected" list, return the value of "true" out of this function to a variable for further use. Else, return “false” which shows the IP address was not in a “susceptible” state for infection.



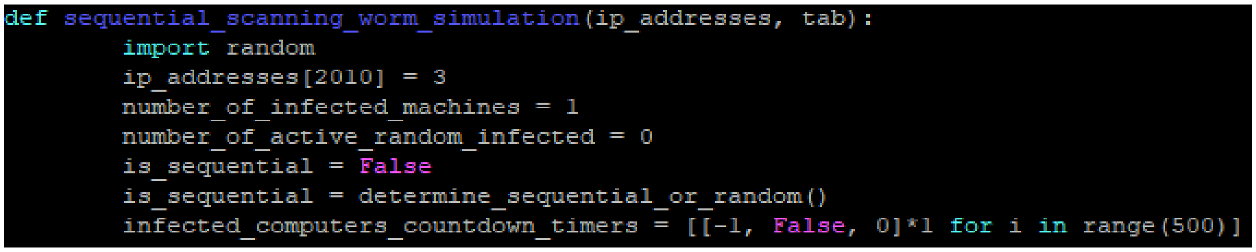
```
else:
        if(ip_addresses[ip_address[list_accessor][0]] == 2):
                ip_addresses[ip_address[list_accessor][0]] = 3
                for j in range(len(were_computers_infected)):
                        if(were_computers_infected[j] is False):
                                were_computers_infected[j] = True
                                break
                        else:
                                continue
                return True
        else:
                return False
```

This code block functions the same way as the before mentioned code block, only it will deal with the scenario of accessing our total IP addresses using the value contained within a double-list. We also see that “list_accessor” will be used as an additional index point to help access the IP address value within our double-list and the function will proceed as previously described.



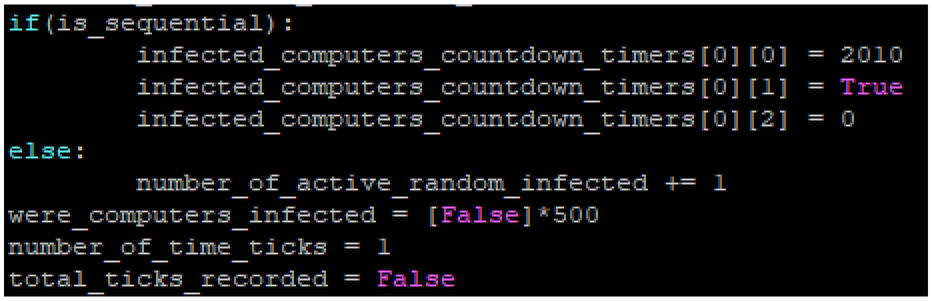
```
def activate_random_reinitialize_countdown(infected_computers_countdown_timers, index_accessor):
        infected_computers_countdown_timers[index_accessor][0] = -1
        infected_computers_countdown_timers[index_accessor][1] = False
        infected_computers_countdown_timers[index_accessor][2] = 20
        return 1
```

The purpose of this function is to convert a “random scanning” IP address into an “active” state and start scanning IP addresses randomly in the next time tick. This is done by returning a value of 1 to the variable that stores the total number of actively scanning “random” IP addresses. We initialize its location in the list back to its original state, there’s no longer any need to keep track of the “random scanning” IP address since it’s now active, so we make space in the double-list for the next infected IP address.



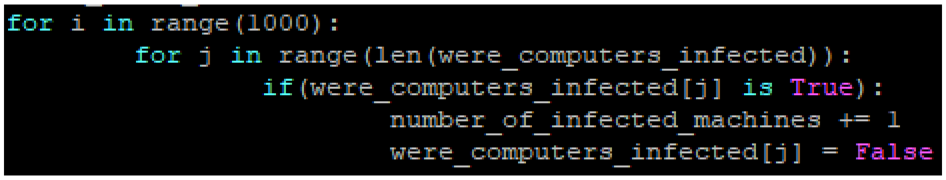
```
def sequential_scanning_worm_simulation(ip_addresses, tab):
        import random
        ip_addresses[2010] = 3
        number_of_infected_machines = 1
        number_of_active_random_infected = 0
        is_sequential = False
        is_sequential = determine_sequential_or_random()
        infected_computers_countdown_timers = [[-1, False, 0]*1 for i in range(500)]
```

This function will carry out the simulation for our “sequential scanning worm”. We pass in the IP addresses list and a “tab” variable, which will function as described within the “random scanning worm” simulation. We initialize many of the same values contained within the previous simulation, the only difference in this simulation is that we’ve included two additional variables: “is_sequential” and “infected_computers_countdown_timers” in the form of a double-list. Our “is_sequential” variable will check to see if the infected IP address, in this scenario the IP address located at index location 2010, will take on a “random scanning” or “sequential scanning” behavior. We also initialize our “infected_computers_countdown_timers” with a size of 500 and give it 3 entries to store: one for recording the IP address number, one for checking if it has a “random scanning” or “sequential scanning” behavior, and a countdown timer before the IP address changes into an “active” state.



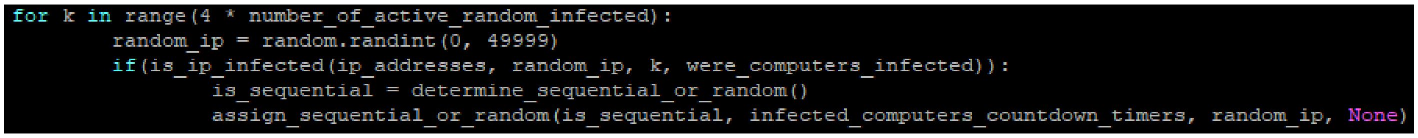
```
if(is_sequential):
        infected_computers_countdown_timers[0][0] = 2010
        infected_computers_countdown_timers[0][1] = True
        infected_computers_countdown_timers[0][2] = 0
else:
        number_of_active_random_infected += 1
were_computers_infected = [False]*500
number_of_time_ticks = 1
total_ticks_recorded = False
```

Based on the results returned from "is_sequential", if the infected IP address at index 2010 has a "sequential scanning" behavior, then store it within our double-list for future records. Otherwise, add onto the "random scanning" behavior and switch to an "active" state. The following variables function the same as in our previous simulation.



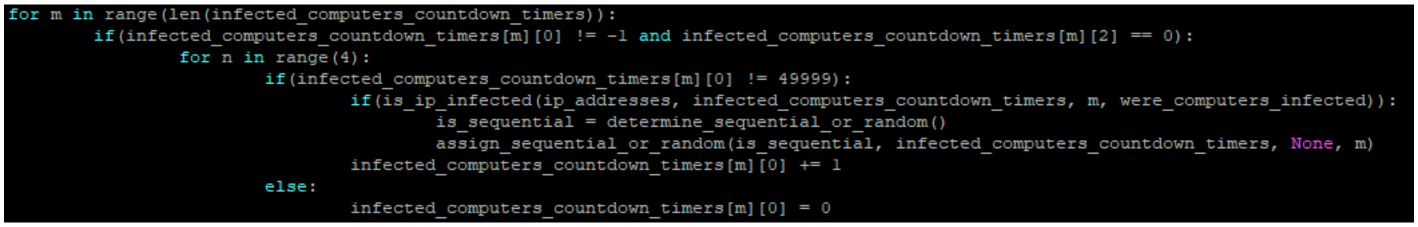
```
for i in range(1000):
        for j in range(len(were_computers_infected)):
                if(were_computers_infected[j] is True):
                        number_of_infected_machines += 1
                        were_computers_infected[j] = False
```

This code block represents the beginning of our 1000 trials for our simulation. The for "j" loop takes care of record keeping just like in our previous "random scanning" worm simulation.



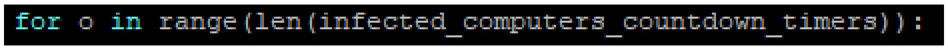
```
for k in range(4 * number_of_active_random_infected):
        random_ip = random.randint(0, 49999)
        if(is_ip_infected(ip_addresses, random_ip, k, were_computers_infected)):
                is_sequential = determine_sequential_or_random()
                assign_sequential_or_random(is_sequential, infected_computers_countdown_timers, random_ip, None)
```

This code block deals with the "random scanning" IP addresses in the "active" state, trying to infect additional IP addresses. It functions just like in our "random scanning" worm simulation and now contains additional functionality to determine if the IP address will have a "random scanning" or "sequential scanning" behavior before assigning it to our double-list. Please see previous descriptions of the functions being executed for more details.

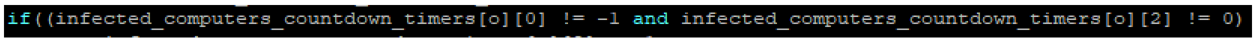


```
for m in range(len(infected_computers_countdown_timers)):
        if(infected_computers_countdown_timers[m][0] != -1 and infected_computers_countdown_timers[m][2] == 0):
                for n in range(4):
                        if(infected_computers_countdown_timers[m][0] != 49999):
                                if(is_ip_infected(ip_addresses, infected_computers_countdown_timers, m, were_computers_infected)):
                                        is_sequential = determine_sequential_or_random()
                                        assign_sequential_or_random(is_sequential, infected_computers_countdown_timers, None, m)
                                infected_computers_countdown_timers[m][0] += 1
                        else:
                                infected_computers_countdown_timers[m][0] = 0
```

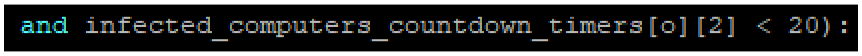
The following code block contains the functionality for our “sequential scanning” behavior of infected IP addresses. In the for “m” loop, we traverse throughout our double-list to check for a stored IP address and if it is in the “active” state. If this is the case we will commence with pinging the network 4 times with our for “n” loop. As long as we don’t hit the last index space within our IP addresses list, we check if the IP address we came across happens to be “susceptible”. If it is then determine was type of behavior it will be assigned and place it within our double-list. We also increment the IP address value within our double-list to continue scanning sequentially. If we reach the end of the IP addresses list, then we set the “sequential scanning” IP address to continue traversal at the beginning of the list using our “else” block.



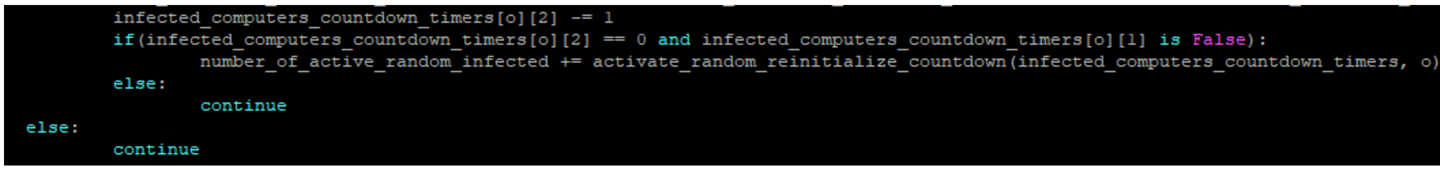
```
for o in range(len(infected_computers_countdown_timers)):
```



```
if((infected_computers_countdown_timers[o][0] != -1 and infected_computers_countdown_timers[o][2] != 0)
```

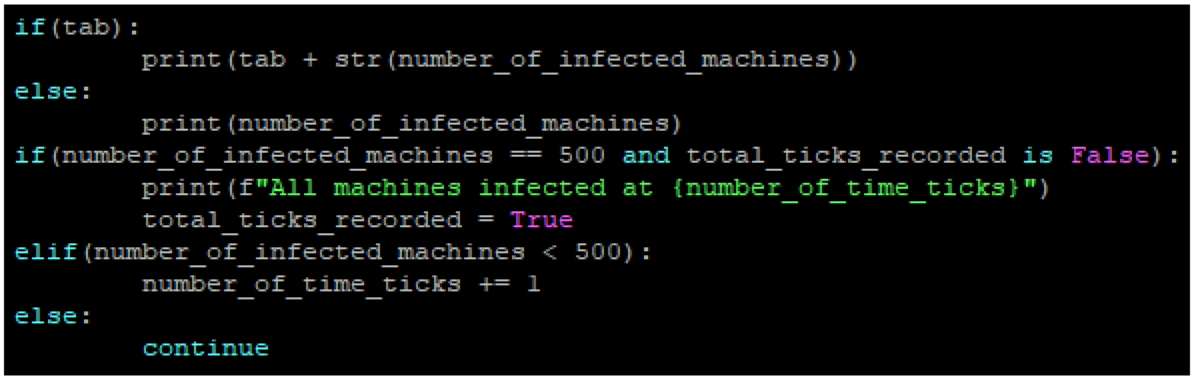


```
and infected_computers_countdown_timers[o][2] < 20):
```



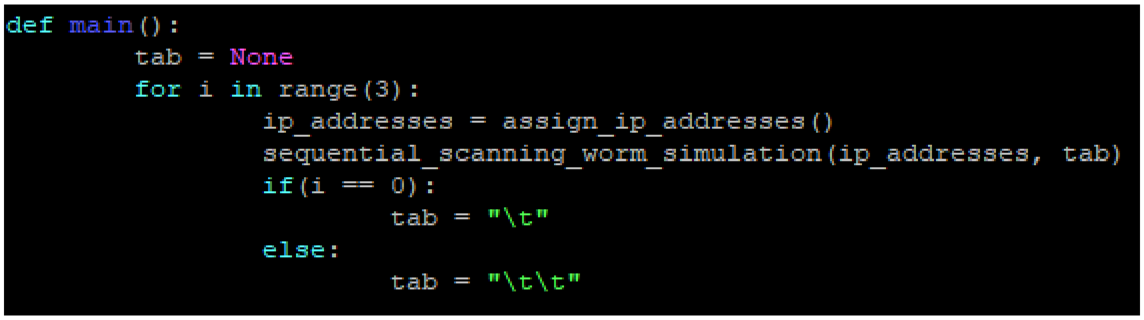
```
            infected_computers_countdown_timers[o][2] -= 1
            if(infected_computers_countdown_timers[o][2] == 0 and infected_computers_countdown_timers[o][1] is False):
                    number_of_active_random_infected += activate_random_reinitialize_countdown(infected_computers_countdown_timers, o)
            else:
                    continue
    else:
            continue
```

This code block deals with updating our countdown timers and updating our “active scanning” IP addresses if they also happen to reach 0. We traverse the double-list with our for “o” loop and check several factors within the following conditional. If our double-list has an IP address assigned at the current space during traversal, its countdown timer has not reached 0, and the timer is less than 20 due to a previous decrement, then decrement the countdown timer for this IP address by 1. The next conditional checks if our countdown timer for the assigned IP address immediately reached 0 and if it contains a "random scanning" behavior. If that is the case, then we'll increment the total number of "active random scanning" IP addresses by 1. Otherwise, "continue" throughout the double-list. In addition, if there is no entry to decrement a countdown timer then we will "continue" throughout the list.



```
if(tab):
        print(tab + str(number_of_infected_machines))
else:
        print(number_of_infected_machines)
if(number_of_infected_machines == 500 and total_ticks_recorded is False):
        print(f"All machines infected at {number_of_time_ticks}")
        total_ticks_recorded = True
elif(number_of_infected_machines < 500):
        number_of_time_ticks += 1
else:
        continue
```

This code block deals with printing out our total time ticks and including a tab whenever necessary. This functions in the same way as our "random scanning worm" simulation.



```
def main():
        tab = None
        for i in range(3):
                ip_addresses = assign_ip_addresses()
                sequential_scanning_worm_simulation(ip_addresses, tab)
                if(i == 0):
                        tab = "\t"
                else:
                        tab = "\t\t"
```

This function is responsible for the execution and running of our program. It is identical in essence to our "random scanning worm" simulation.