

Algorithm for file updates in Python

Project description

In my organization, access to restricted content is managed through an IP allow list stored in the "allow_list.txt" file. A separate list is maintained to track IP addresses that should be revoked. I developed an algorithm to automate the process of updating the allow list by removing any IP addresses that no longer require access.

Open the file that contains the allow list

In the initial step of the algorithm, I assigned the filename "allow_list.txt" as a string to the variable `import_file`, which I then used to open the file.

```
# Assign `import_file` to the name of the file  
  
import_file = "allow_list.txt"
```

Then, I used a `with` statement to open the file:

```
# Build `with` statement to read in the initial contents of the file  
  
with open(import_file, "r") as file:
```

To access the IP addresses listed in the allow list, my algorithm reads the file using a context manager. By using the `with` statement alongside the `open()` function in read mode, I ensure the file is properly handled and automatically closed after use. In the line `with open(import_file, "r") as file:`, the first argument specifies which file to work with, while the second ("r") tells the function to open it in read-only mode. The `as` keyword assigns the opened file to the variable `file`, which I use to interact with its contents during the execution of the block.

Read the file contents

In order to read the file contents, I used the `.read()` method to convert it into the string.

```
with open(import_file, "r") as file:

    # Use `.read()` to read the imported file and store it in a variable named `ip_addresses`

    ip_addresses = file.read()
```

By opening the file in read mode ("r") using the `.open()` function within a `with` block, I was able to access its contents through the `.read()` method. This method retrieves the full contents of the file as a single string. I used it on the file object defined in the `with` statement and stored the resulting string in a variable named `ip_addresses`.

Essentially, this step loads the entire content of "**allow_list.txt**" into memory as a string, which I can later process and manipulate within my Python script to extract and organize relevant data.

Convert the string into a list

In order to remove individual IP addresses from the allow list, I needed it to be in list format. Therefore, I next used the `.split()` method to convert the `ip_addresses` string into a list:


```
# Use `.split()` to convert `ip_addresses` from a string to a list

ip_addresses = ip_addresses.split()
```

To transform the IP address data into a more manageable format, I used the `.split()` method directly on the `ip_addresses` string. This method breaks up the string wherever it finds whitespace and returns the individual segments as elements in a new list. Since the IP addresses in this case are separated by spaces, `.split()` effectively isolates each one. I then stored the resulting list by overwriting the original `ip_addresses` variable, making it easier to identify and remove specific entries from the allow list.

Iterate through the remove list

A key part of my algorithm involves iterating through the IP addresses that are elements in the `remove_list`. To do this, I incorporated a `for` loop:

```
 # Build iterative statement
# Name loop variable `element`
# Loop through `remove_list`

for element in remove_list:
```

In Python, a `for` loop is used to cycle through each item in a collection, such as a list. Its main role in this type of algorithm is to execute a set of instructions for every item in the group. The loop begins with the keyword `for`, followed by a temporary variable — in this case, `element`. The word `in` signals that the loop should pull items one by one from the `ip_addresses` sequence, assigning each to `element` as the loop runs.

Remove IP addresses that are on the remove list

To ensure accurate access control, my script filters out any IP address listed in both the `ip_addresses` list and the `remove_list`. Since `ip_addresses` contained only unique entries, I was able to accomplish this using the following approach:

```
for element in remove_list:

    # Create conditional statement to evaluate if `element` is in `ip_addresses`

    if element in ip_addresses:

        # use the `.remove()` method to remove
        # elements from `ip_addresses`

        ip_addresses.remove(element)
```

To avoid runtime errors, I began by adding a safeguard inside the loop: a condition that checked if the current item existed in the `ip_addresses` list. This was necessary because calling `.remove()` on a non-existent item would trigger an exception.

If the condition was met, I proceeded to delete the item from `ip_addresses` by passing the loop variable as an argument to the `.remove()` method, effectively filtering out any IP address listed in `remove_list`.

Update the file with the revised list of IP addresses

To wrap up the algorithm, I had to overwrite the allow list file with the updated IP addresses. This required transforming the list into a single string format. I accomplished this using the `.join()` function.

```
# Convert `ip_addresses` back to a string so that it can be written into the text file

ip_addresses = "\n".join(ip_addresses)
```

The `.join()` function merges all elements from an iterable into a single string. It operates on a string that specifies the delimiter inserted between each element during the concatenation. In this algorithm, I utilized `.join()` to transform the `ip_addresses` list into one continuous string, which I then supplied as input to the `.write()` function to save the content into the file `"allow_list.txt"`. By using the newline character (`"\n"`) as the delimiter, I ensured that each IP address would appear on its own separate line.

After that, I employed another `with` block along with the `.write()` method to overwrite the file with the updated data:

```
# Build `with` statement to rewrite the original file
with open(import_file, "w") as file:
    # Rewrite the file, replacing its contents with `ip_addresses`
    file.write(ip_addresses)
```

This time, I included a second parameter `"w"` in the `open()` function within my `with` block. This parameter tells Python to open the file in write mode, which clears any existing content so new data can be saved. By opening the file this way, I'm able to use the `.write()` method inside the `with` block. The `.write()` method takes a string and outputs it directly into the opened file, overwriting what was there before.

My goal was to save the revised allow list, formatted as a string, back into `"allow_list.txt"`. Doing so ensures that any IP addresses removed from the list will no longer have permission to access the restricted resources. To update the file, I called `.write()` on the file handle created in the `with` statement, passing the variable `ip_addresses` as the content to write, effectively replacing the old file data with the new list.

Summary

I developed a procedure to eliminate IP addresses listed in the `remove_list` from the authorized addresses recorded in the `"allow_list.txt"` file. The process started by opening the file and reading its contents as a string, which I then transformed into a list assigned to the variable `ip_addresses`. Next, I looped through each IP in the `remove_list` and checked if it existed within `ip_addresses`. Whenever a match was found, I removed that IP from the list using the `.remove()` function. Finally, I reconstructed the updated list into a single string using `.join()` and overwrote the original file with this cleaned set of allowed IPs.