# Algorithm for file updates in Python

## Project description

In this project, I developed a Python algorithm to manage access control by updating a file containing a list of IP addresses that are allowed to access restricted content. The scenario involves ensuring that IP addresses belonging to users who should no longer have access are removed from the allow list. The algorithm reads the current list of IP addresses from a file, compares it to a list of IPs that need to be removed, and updates the file by removing the specified addresses. This solution is designed to automate the process of maintaining up-to-date access control records, which is essential for ensuring secure access to sensitive information.

## Open the file that contains the allow list

To start the algorithm, I opened the file that contains the list of IP addresses allowed to access restricted information. I assigned the file name `"allow\_list.txt"` to a variable called `import\_file`.



I used a **with** statement to open the file because it automatically handles closing the file once the operation is done. Inside the **with** block, I used the **open()** function with two parameters:

* The first parameter is the file name (`import\_file`), and
* The second parameter is `"r"` to specify the file is being opened for reading.

The file content is assigned to the variable `file` so that it can be read and processed. Here is the Python code that accomplishes this:



## Read the file contents

After opening the file, I used the **.read()** method to read the entire content of the file as a string. This method reads all lines of the file at once and stores them in a variable called `ip\_addresses`.

The **.read()** function does not modify the original content but rather stores it in a Python variable for further manipulation. This is necessary for parsing and processing the data later in the algorithm.

Here is the Python code that accomplishes this:



This approach allows the entire contents of the file to be stored as a string in the `ip\_addresses` variable, which can then be split or manipulated as needed.

## Convert the string into a list

To process each IP address individually, I used the **.split()** method. The **.split()** method converts a string into a list by splitting it at each space or specified separator. In this case, the list of IP addresses was originally stored as a single string, and by applying the **.split()** method, it was broken down into a list where each element corresponds to an individual IP address.

Here’s the Python code that performs this operation:



By splitting the string, each IP address is now a separate element in the list. This allows for easier iteration, comparison, and removal of specific IP addresses.

## Iterate through the remove list

To check each IP address in the **remove\_list** and compare it with the **ip\_addresses** list, I used a **for** loop. The **for** loop iterates over each element in the **remove\_list**, assigning each IP address in turn to the variable **element**.

The syntax of a **for** loop in Python is simple:



This loop goes through every IP address in the remove\_list, which allows us to check if the current IP address (**element**) needs to be removed from the **ip\_addresses** list. The loop continues until it has processed every item in **remove\_list**.

This is an efficient way to compare and potentially remove multiple IP addresses at once.

## Remove IP addresses that are on the remove list

To remove the IP addresses that are present in both the **remove\_list** and the **ip\_addresses** list, I used an **if** conditional statement inside the **for** loop. The conditional checks if the current **element** from the **remove\_list** is present in **ip\_addresses**. If the condition evaluates as **True**, the **.remove()** method is applied to **ip\_addresses** to delete the matching IP address.

The structure of the conditional statement and the **.remove()** method looks like this:



This code checks whether the current IP address (**element**) exists in **ip\_addresses** and, if so, removes it from the list.

Since there are no duplicate IP addresses in the **ip\_addresses** list, using the **.remove()** method works correctly, ensuring that each IP address is removed only once and the process completes without issues.

## Update the file with the revised list of IP addresses

After removing the IP addresses from the **ip\_addresses** list, the list must be converted back into a string format so that it can be written back into the file. This is achieved by using the **.join()** method, which combines the elements of **ip\_addresses** into a single string. By applying **.join()** to the string **"\n"**, each IP address in the list is placed on a new line in the file. The code for this looks like this:



Next, the updated **ip\_addresses** string is written back into the file using a **with** statement, which opens the file with the **"w"** parameter. This parameter ensures that the contents of the file will be overwritten. The **.write()** method is used to write the revised list back to the file:



This process updates the original file, ensuring that the new allow list of IP addresses is saved correctly.

## Summary

The algorithm is designed to automate the process of updating an IP allow list by removing IP addresses that should no longer have access. It begins by opening and reading the contents of the allow list file, converting the string of IP addresses into a list for easier manipulation. The algorithm then iterates through a separate removal list and removes any matching IP addresses from the allow list. After the necessary IPs are removed, the updated list is converted back into a string, and the original file is overwritten with the revised list. This process ensures the file is kept current, minimizing manual effort and potential errors in access control management.