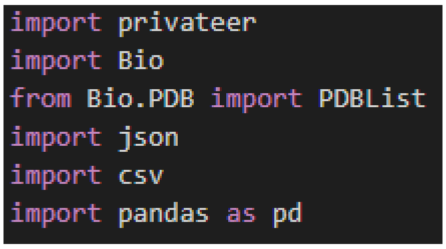
Jordan Dialpuri

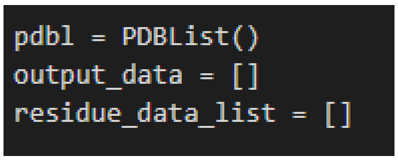
Code Explanation

The first part of the code are imports, importing these modules means that you can perform certain actions such as downloading PDB files through biopython etc. You can look pandas, biopython up if you want to know more about what else they can do.



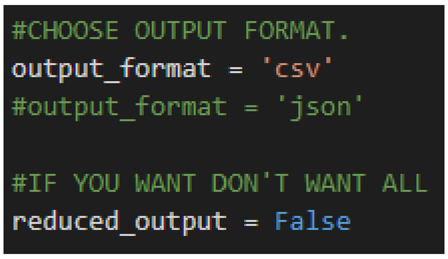
```
import privateer
import Bio
from Bio.PDB import PDBList
import json
import csv
import pandas as pd
```

The next part, we initiate a class called pdbl which just means that we are now able to access code that is in the biopython module using the pdbl tag. Then we create two lists which are used to store the output (output_data does sugar_sugar and residue_data_list does residue_sugar outputs).

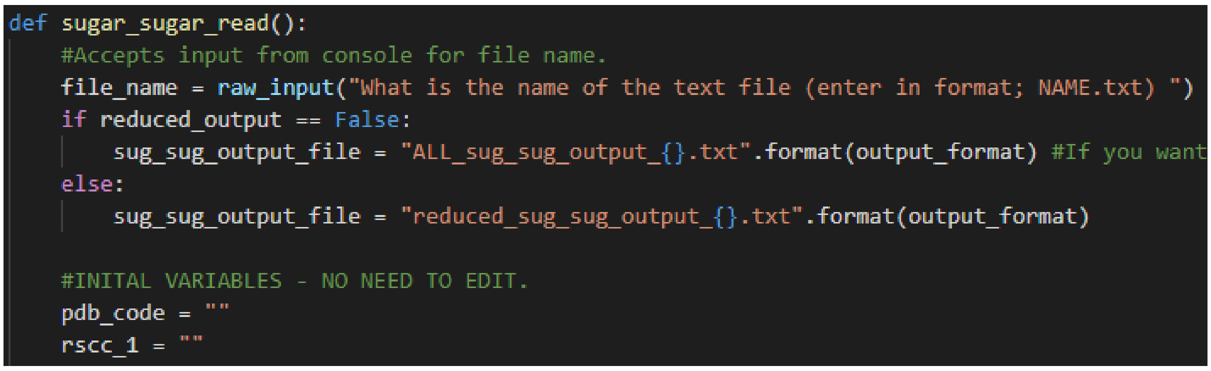


```
pdbl = PDBList()
output_data = []
residue_data_list = []
```

The next part is where you select what output, this is referenced later when we get to the point where we need to put something in the list above (i.e. do we put it in csv format or json format into that list). Reduced_output just means that none of the extra data like Detected type, mFo, Ctx etc is outputted to the document.

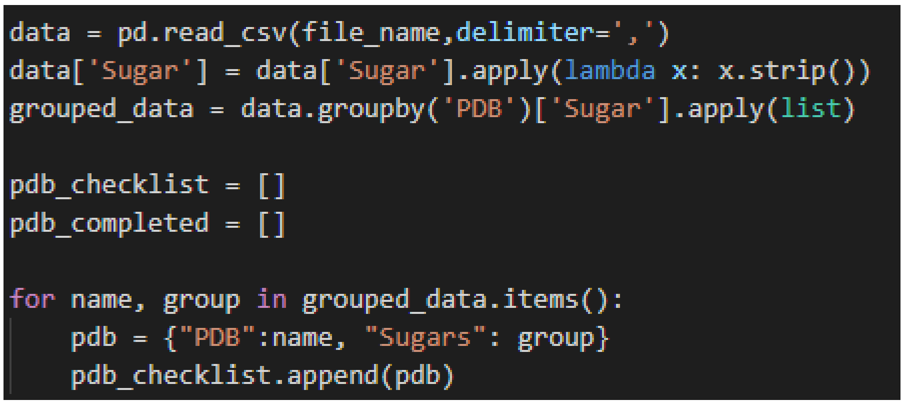


```
#CHOOSE OUTPUT FORMAT.
output_format = 'csv'
#output_format = 'json'

#IF YOU WANT DON'T WANT ALL
reduced_output = False
```



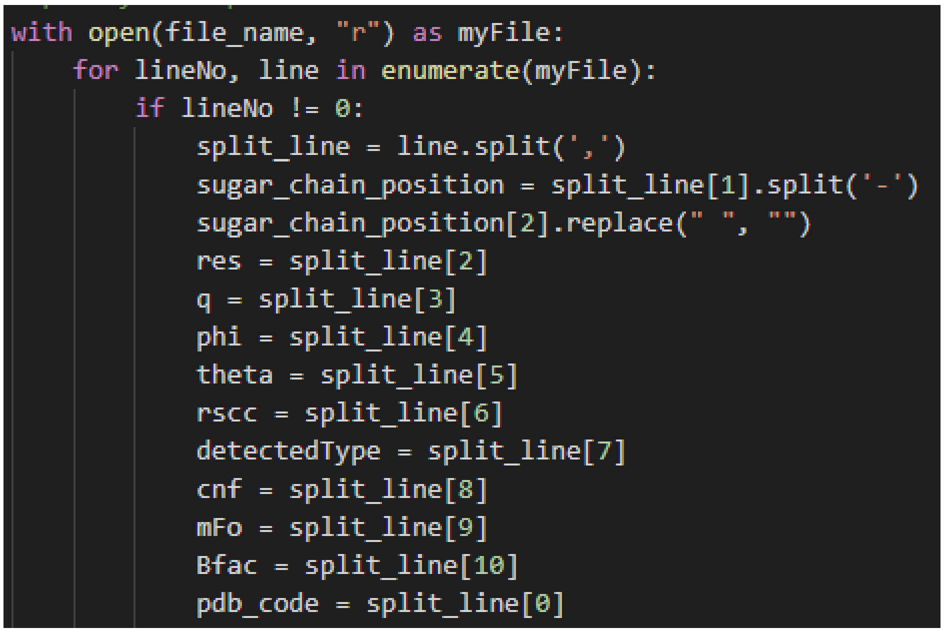
```
def sugar_sugar_read():
    #Accepts input from console for file name.
    file_name = raw_input("What is the name of the text file (enter in format; NAME.txt) ")
    if reduced_output == False:
        sug_sug_output_file = "ALL_sug_sug_output_{}.txt".format(output_format) #If you want
    else:
        sug_sug_output_file = "reduced_sug_sug_output_{}.txt".format(output_format)

    #INITAL VARIABLES - NO NEED TO EDIT.
    pdb_code = ""
    rscc_1 = ""
```

def sugar_sugar_read() declares a function named sugar_sugar_read(), basically a container for the code inside that is only run once we call it (at the end). file_name = raw_input() sets a variable called file_name as whatever the input from the user is, this is where you enter the file name of the split and is referenced later when it comes to opening that file. The if statement here is used to change the name of the output file depending on whether you put reduced_output = False or True just so you know in the directory which file it outputs. Then we initialise some variables pdb_code and rscc_1 (this is not needed and is something I forgot to remove).



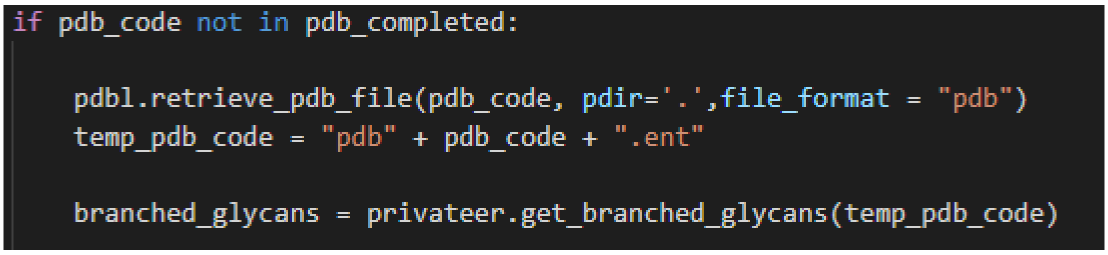
```python
data = pd.read_csv(file_name,delimiter=',')
data['Sugar'] = data['Sugar'].apply(lambda x: x.strip())
grouped_data = data.groupby('PDB')['Sugar'].apply(list)

pdb_checklist = []
pdb_completed = []

for name, group in grouped_data.items():
    pdb = {"PDB":name, "Sugars": group}
    pdb_checklist.append(pdb)
```

This code is where the pandas module comes in, it takes in the file_name that the user inputs and then stores it in what is called a DataFrame (kind of like an excel spreadsheet but accessible via code). The data['Sugar'] = data['Sugar'].apply() line is very important and is used because in the input file there are two spaces after the sugar entry and this would interfere with the code later if it was not 'stripped' here. I used a lambda function here which basically is a neat way to strip every bit of data without use of loop (you can look lambda functions up if interested but it isn't essential to understand them). Then the data is grouped by the PDB code and all the sugars are collected. This is where the increased speed of the 1.2/1.3 function comes from because instead of running through every line like 1.1 did this only runs through unique PDBs. The for loop is used to store these in a neat way rather than a groupedDataFrame (hard to work worth).



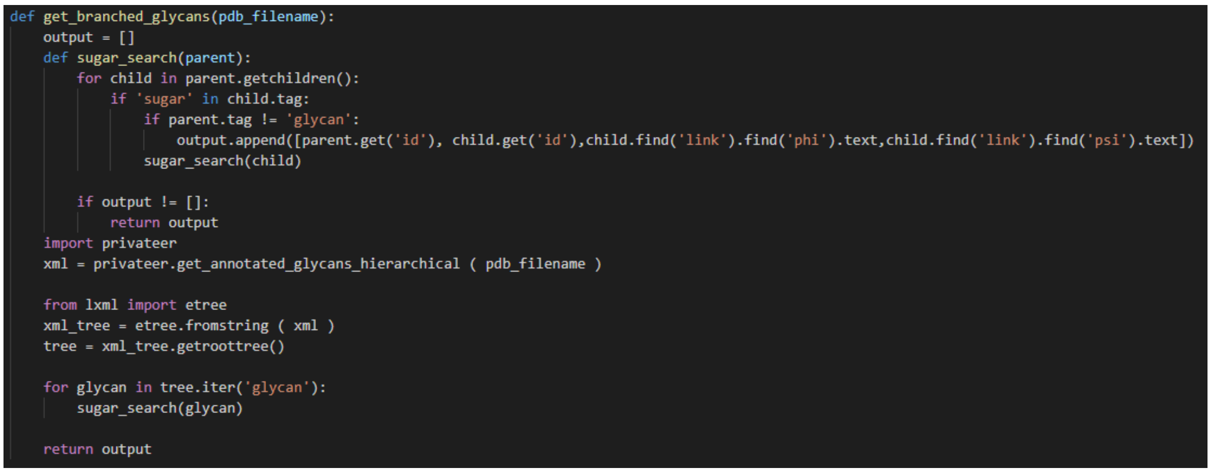
```python
with open(file_name, "r") as myFile:
    for lineNo, line in enumerate(myFile):
        if lineNo != 0:
            split_line = line.split(',')
            sugar_chain_position = split_line[1].split('-')
            sugar_chain_position[2].replace(" ", "")
            res = split_line[2]
            q = split_line[3]
            phi = split_line[4]
            theta = split_line[5]
            rscc = split_line[6]
            detectedType = split_line[7]
            cnf = split_line[8]
            mFo = split_line[9]
            Bfac = split_line[10]
            pdb_code = split_line[0]
```

At this point we are opening the file in read mode (denoted by the "r"), and for each line in the file except the first one we are going to split it into a list with elements every time there is a comma (look at the input file). The rest of the code (from res downwards) is unnecessary and something left over from the previous versions. Except pdb_code = split_line[0] which is where we get the PDB code to put into privateer from.



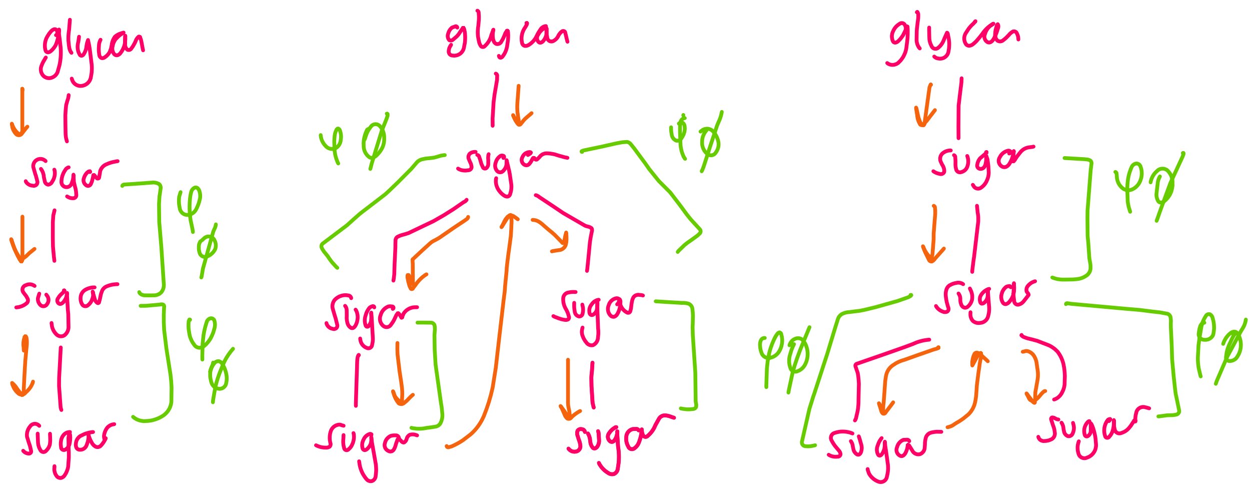
```python
if pdb_code not in pdb_completed:

    pdbl.retrieve_pdb_file(pdb_code, pdir='.',file_format = "pdb")
    temp_pdb_code = "pdb" + pdb_code + ".ent"

    branched_glycans = privateer.get_branched_glycans(temp_pdb_code)
```

Here we check that the code is not in the ones we've already done (as to not repeat it) and then we use that class initialised earlier to download the pdb file using biopython. You probably noticed that this created lots of files with the name format pdb4cdh.ent which is what we need to put into privateer for it to recognise the file, hence the addition of those elements onto the pdb_code which is then put into the call to the privateer function.

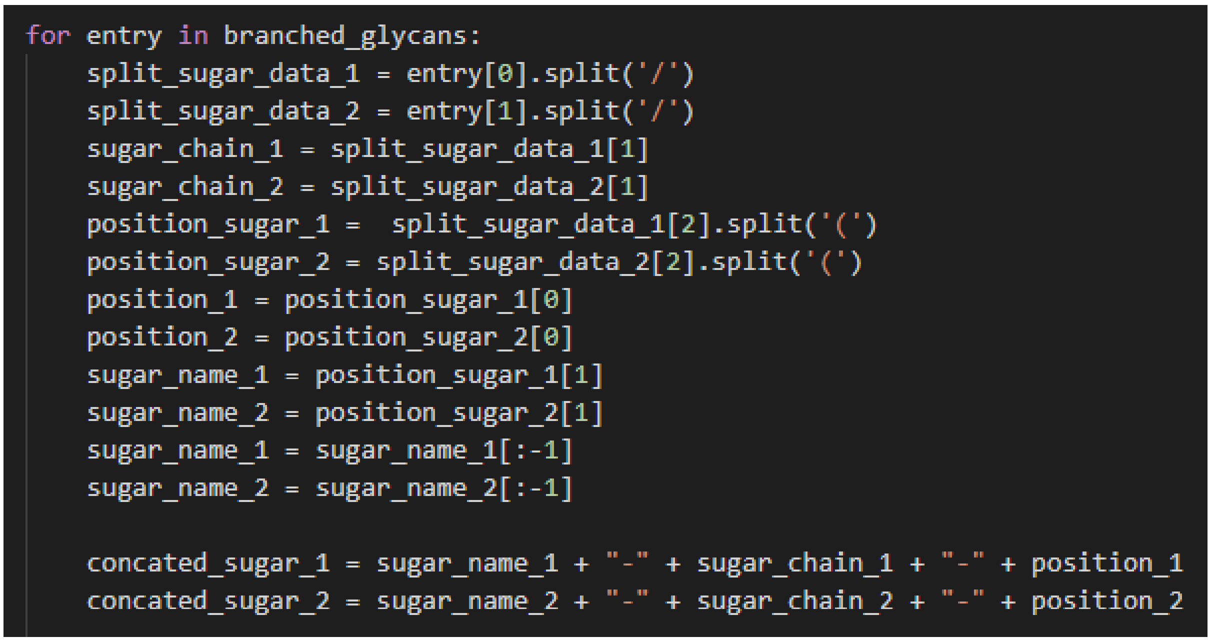


```python
def get_branched_glycans(pdb_filename):
    output = []
    def sugar_search(parent):
        for child in parent.getchildren():
            if 'sugar' in child.tag:
                if parent.tag != 'glycan':
                    output.append([parent.get('id'), child.get('id'),child.find('link').find('phi').text,child.find('link').find('psi').text])
                sugar_search(child)

        if output != []:
            return output
    import privateer
    xml = privateer.get_annotated_glycans_hierarchical ( pdb_filename )

    from lxml import etree
    xml_tree = etree.fromstring ( xml )
    tree = xml_tree.getroottree()

    for glycan in tree.iter('glycan'):
        sugar_search(glycan)

    return output
```

 The privateer function get_branched_glycans contains another function called sugar_search. After importing modules and getting xml code for the protein that it was called for it searches through the xml to get the sugar sugar linkages for every chain there, this is the new method I used to get the branched glycans. Imagine a structure like this:

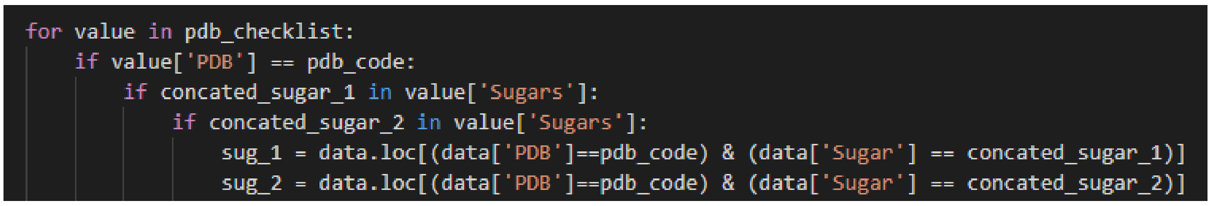


it goes down one chain as far as it can (recording sugars as it goes) and then once it hits the end (it can't find any more sugars) it comes back up to the branch point and then goes down again until there are no more branch points, this is called recursion and is something you can look up if you want to learn more but essentially it goes through all the sugar-sugar linkages recording the psi and phi values along the way. this function returns those angles for every one as well as the name of the two sugars in a 2D array [[name, name, angle, angle],[name,name,angle,angle]]. This is then put into the variable branched_glycans in the auto program.



```python
for entry in branched_glycans:
    split_sugar_data_1 = entry[0].split('/')
    split_sugar_data_2 = entry[1].split('/')
    sugar_chain_1 = split_sugar_data_1[1]
    sugar_chain_2 = split_sugar_data_2[1]
    position_sugar_1 =  split_sugar_data_1[2].split('(')
    position_sugar_2 = split_sugar_data_2[2].split('(')
    position_1 = position_sugar_1[0]
    position_2 = position_sugar_2[0]
    sugar_name_1 = position_sugar_1[1]
    sugar_name_2 = position_sugar_2[1]
    sugar_name_1 = sugar_name_1[:-1]
    sugar_name_2 = sugar_name_2[:-1]

    concated_sugar_1 = sugar_name_1 + "-" + sugar_chain_1 + "-" + position_1
    concated_sugar_2 = sugar_name_2 + "-" + sugar_chain_2 + "-" + position_2
```

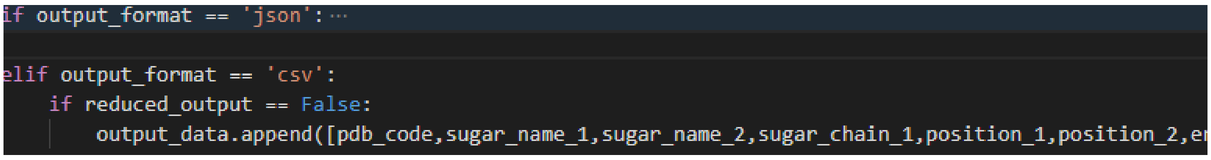
for every [name,name,angle,angle] that is returned this splits the format from privateer NAG/C/6 into NAG C and 6 which can be used for the output. It is also converted to the format used in the input for later use.



```python
for value in pdb_checklist:
    if value['PDB'] == pdb_code:
        if concated_sugar_1 in value['Sugars']:
            if concated_sugar_2 in value['Sugars']:
                sug_1 = data.loc[(data['PDB']==pdb_code) & (data['Sugar'] == concated_sugar_1)]
                sug_2 = data.loc[(data['PDB']==pdb_code) & (data['Sugar'] == concated_sugar_2)]
```

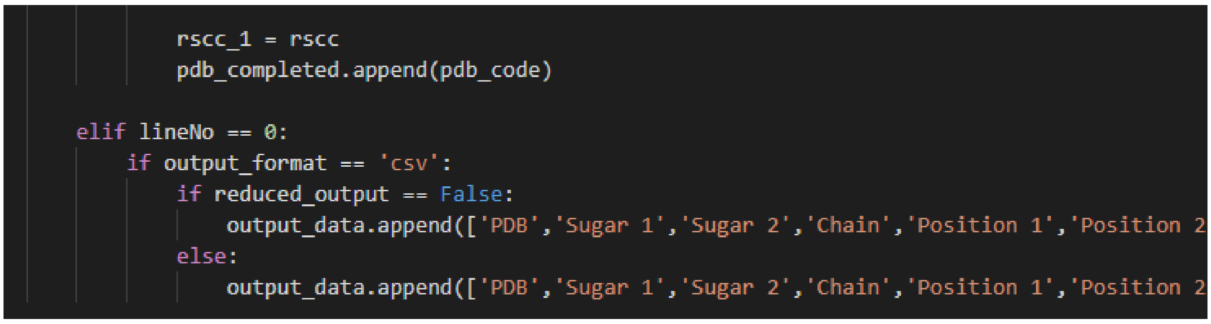
This goes through every entry into that checklist earlier (created from the input file) and when it finds one that matches it checks to see if the outputted sugars from privateer (Sugar 1 and Sugar 2 from the printed output) are in the input file. If they are then we know that the phi and psi are what we want and can proceed. This is how I overcame the branching issue along with the custom functions in the privateer.py file.

Then we locate where in the input file the PDB = the pdb we are on and the sugars are right so that we can store all the extra data temporarily before output. This was not in version 1.2 hence the repetition of the extra data.



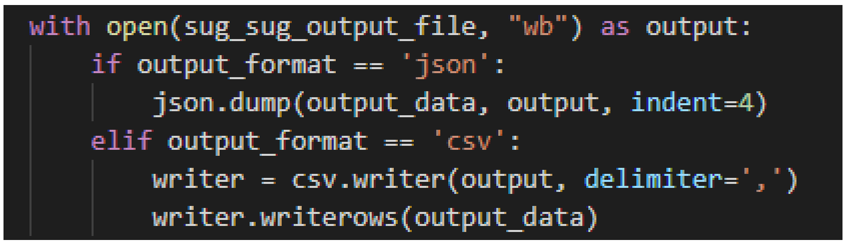
```python
if output_format == 'json':···

elif output_format == 'csv':
    if reduced_output == False:
        output_data.append([pdb_code,sugar_name_1,sugar_name_2,sugar_chain_1,position_1,position_2,e
```

Here we append all the data we have collected into the output_data list created at the start along with all the extra data which comes from sug_1 and sug_2.



```
            rscc_1 = rscc
            pdb_completed.append(pdb_code)

    elif lineNo == 0:
        if output_format == 'csv':
            if reduced_output == False:
                output_data.append(['PDB','Sugar 1','Sugar 2','Chain','Position 1','Position 2
            else:
                output_data.append(['PDB','Sugar 1','Sugar 2','Chain','Position 1','Position 2
```

After this, we set the pdb_code we just did to completed so we don't do it again (where the big time reduction also came from). And then we catch when we are on the first line because in the CSV format, we need to add a heading, and again there is a choice for what heading based if we are on reduced output.



```
with open(sug_sug_output_file, "wb") as output:
    if output_format == 'json':
        json.dump(output_data, output, indent=4)
    elif output_format == 'csv':
        writer = csv.writer(output, delimiter=',')
        writer.writerows(output_data)
```

Then once we have gone through every line of the input file we close that input file and then open the output file in "wb" (writing) mode. Then if we are using the json format we use json.dump to put it into the output file, and if we are in the csv format we write each row with the delimiter (separator) ' , '.

Then we exit the sugar_sugar function and the program finishes. Residue_sugar_read() is very similar although we don't need to go through the branching so that part of the code is slightly shorter only taking the top level (because that is the only place a residue sugar link can be).