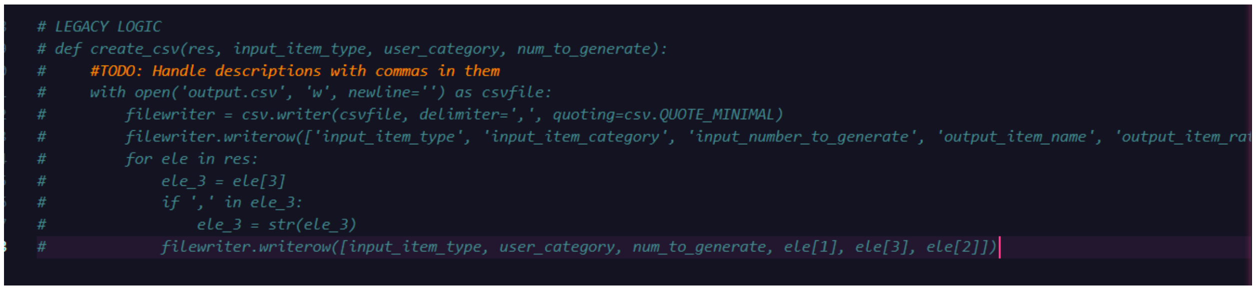
Steinar Jennings

CS361

Code Smells:

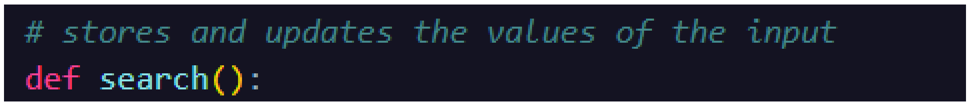
My code smelled a lot. After making rough changes to fit the mold for sprint 4, I left my code completely un-refactored, and have saved separate versions.

My code had commented out code (in the csv logic section, this was the only change in that file and it was removed.



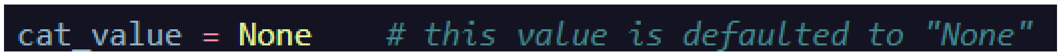
```
# LEGACY LOGIC
# def create_csv(res, input_item_type, user_category, num_to_generate):
#     #TODO: Handle descriptions with commas in them
#     with open('output.csv', 'w', newline='') as csvfile:
#         filewriter = csv.writer(csvfile, delimiter=',', quoting=csv.QUOTE_MINIMAL)
#         filewriter.writerow(['input_item_type', 'input_item_category', 'input_number_to_generate', 'output_item_name', 'output_item_ra
#         for ele in res:
#             ele_3 = ele[3]
#             if ',' in ele_3:
#                 ele_3 = str(ele_3)
#             filewriter.writerow([input_item_type, user_category, num_to_generate, ele[1], ele[3], ele[2]])
```

The first way my code smelled was due to obsolete comments. My code has several areas that were commented that had to deal with previous logic I began implementing but changed at the end.



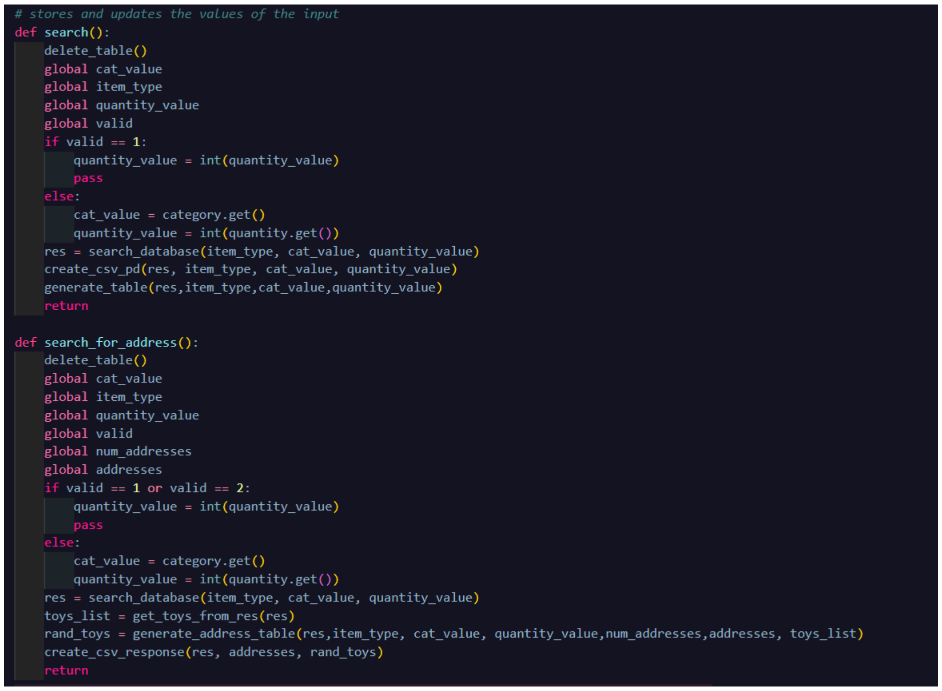
```
# stores and updates the values of the input
def search():
```

The search function was going to do this, but I decided to leave the functionality out.

My code also smelled due to some redundant comments. There are others detailed in the refactored file. 

```
cat_value = None    # this value is defaulted to "None"
```

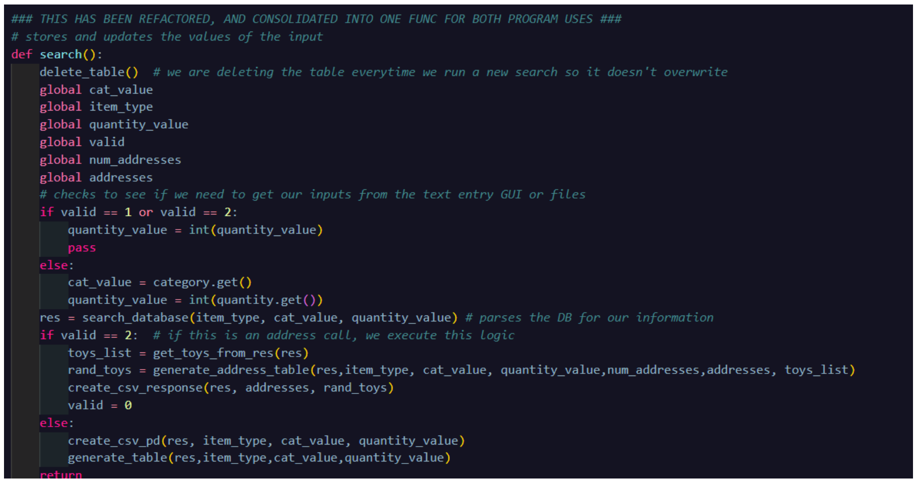
My code smelled badly of redundant functions, however, I had a really hard time consolidating functions due to minor tweaks that involved a lot of code adjustments. I did however consolidate a few functions, but left comments in my file where I made the executive decision not to refactor. These decisions are detailed in the refactored code and refer to instances where making the function handle different types of input resulted in long refactors making it less readable.



```python
# stores and updates the values of the input
def search():
    delete_table()
    global cat_value
    global item_type
    global quantity_value
    global valid
    if valid == 1:
        quantity_value = int(quantity_value)
        pass
    else:
        cat_value = category.get()
        quantity_value = int(quantity.get())
    res = search_database(item_type, cat_value, quantity_value)
    create_csv_pd(res, item_type, cat_value, quantity_value)
    generate_table(res,item_type,cat_value,quantity_value)
    return

def search_for_address():
    delete_table()
    global cat_value
    global item_type
    global quantity_value
    global valid
    global num_addresses
    global addresses
    if valid == 1 or valid == 2:
        quantity_value = int(quantity_value)
        pass
    else:
        cat_value = category.get()
        quantity_value = int(quantity.get())
    res = search_database(item_type, cat_value, quantity_value)
    toys_list = get_toys_from_res(res)
    rand_toys = generate_address_table(res,item_type, cat_value, quantity_value,num_addresses,addresses, toys_list)
    create_csv_response(res, addresses, rand_toys)
    return
```

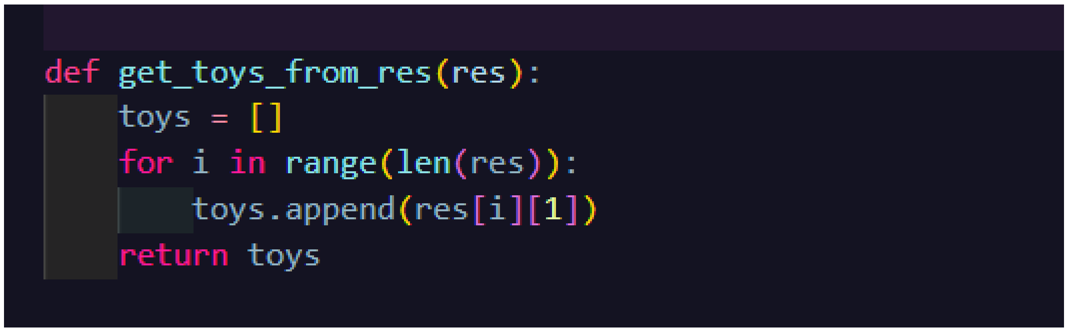
This is an instance where I was able to refactor.



```python
### THIS HAS BEEN REFACTORED, AND CONSOLIDATED INTO ONE FUNC FOR BOTH PROGRAM USES ###
# stores and updates the values of the input
def search():
    delete_table()  # we are deleting the table everytime we run a new search so it doesn't overwrite
    global cat_value
    global item_type
    global quantity_value
    global valid
    global num_addresses
    global addresses
    # checks to see if we need to get our inputs from the text entry GUI or files
    if valid == 1 or valid == 2:
        quantity_value = int(quantity_value)
        pass
    else:
        cat_value = category.get()
        quantity_value = int(quantity.get())
    res = search_database(item_type, cat_value, quantity_value) # parses the DB for our information
    if valid == 2:  # if this is an address call, we execute this logic
        toys_list = get_toys_from_res(res)
        rand_toys = generate_address_table(res,item_type, cat_value, quantity_value,num_addresses,addresses, toys_list)
        create_csv_response(res, addresses, rand_toys)
        valid = 0
    else:
        create_csv_pd(res, item_type, cat_value, quantity_value)
        generate_table(res,item_type,cat_value,quantity_value)
    return
```

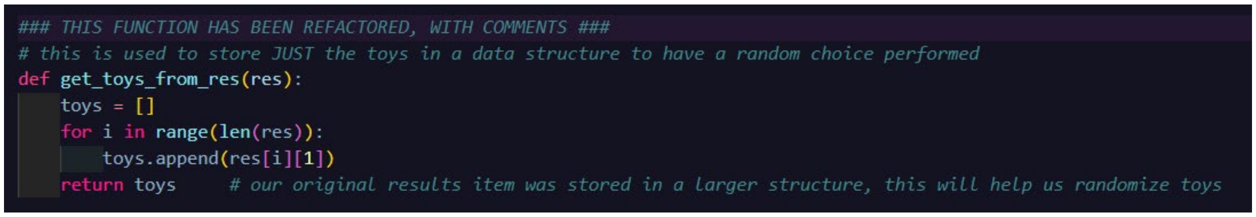
Refactored.

In general, my code also smelled due to a lack of comments, and function descriptions.



```python
def get_toys_from_res(res):
    toys = []
    for i in range(len(res)):
        toys.append(res[i][1])
    return toys
```

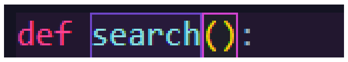
This functions purpose is not clear, it would be better to define why it exists, and when it would be used.



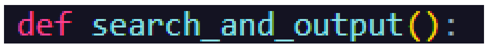
```python
### THIS FUNCTION HAS BEEN REFACTORED, WITH COMMENTS ###
# this is used to store JUST the toys in a data structure to have a random choice performed
def get_toys_from_res(res):
    toys = []
    for i in range(len(res)):
        toys.append(res[i][1])
    return toys     # our original results item was stored in a larger structure, this will help us randomize toys
```

I did have some functions longer than 10 lines, however many of the lines were GUI adjustments that could not so easily be consolidated. In the other cases where functions were long, I felt like it would make the code less readable to separate into subfunctions since the logic is all related closely.

I believe the only function with "many jobs" in my code is my search function, which also calls for CSV files to be produced. Instead of splitting it I have renamed it, because it only uses one line to call for CSV production. All of the other functions relate to roughly one activity.

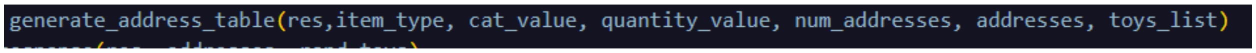


```python
def search():
```



```python
def search_and_output():
```

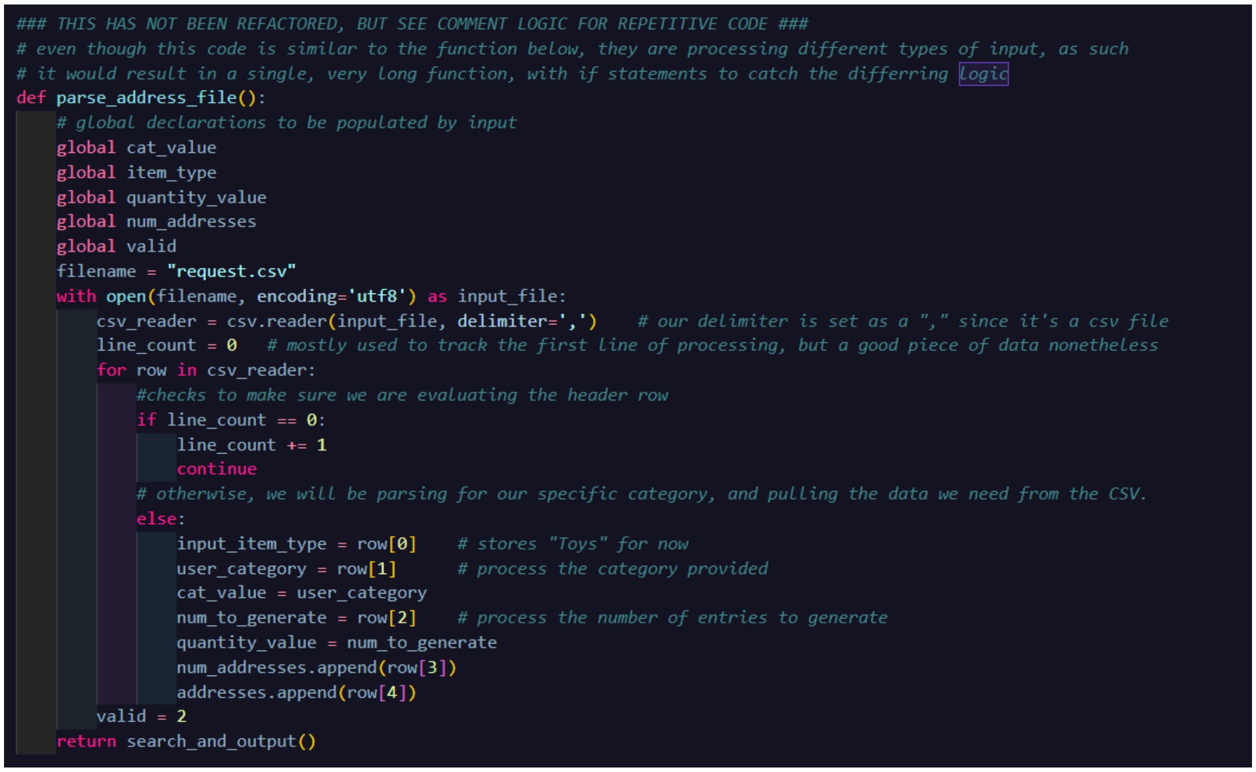
Some of my functions have many parameters, but I believe this is also valid, because it helps us see what is being passed. The parameters are named clearly, and it would not make sense to group them in a parameters data structure in my opinion.



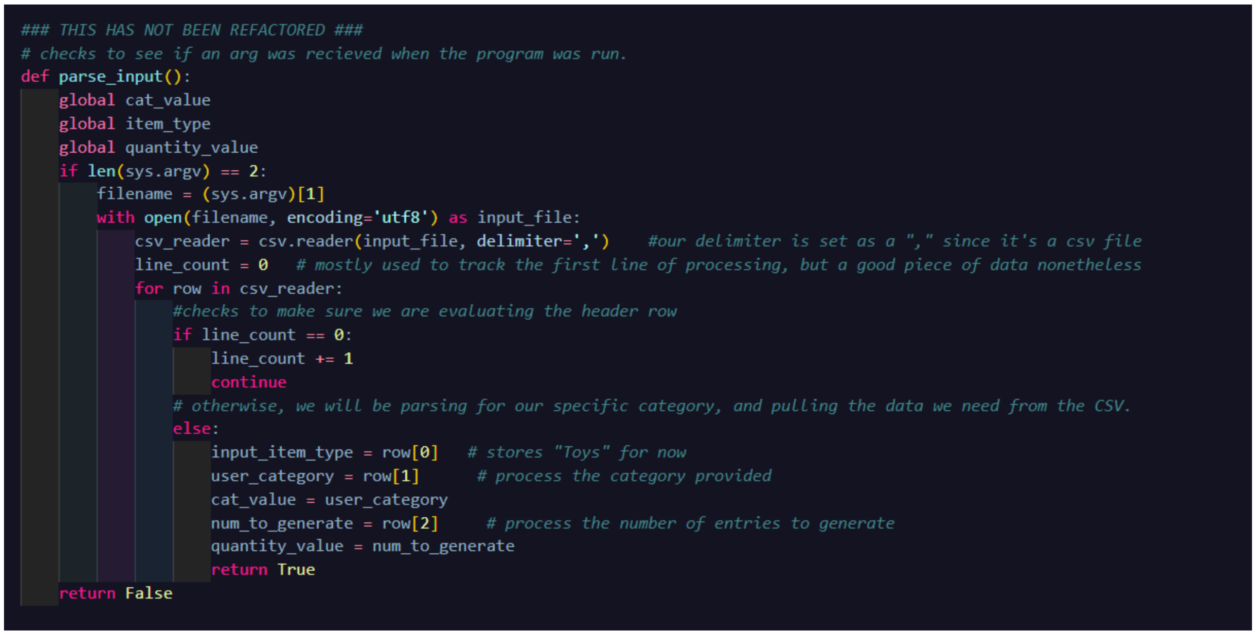
```python
generate_address_table(res,item_type, cat_value, quantity_value, num_addresses, addresses, toys_list)
```

See the example above, it makes sense to include many items, because we have many column in our output. It would result in more complex code to submit the parameters in a single data struct then process.

As I have mentioned above, there is duplicate code in my submission. I have tried my best to clear as much of it out as possible, but I believe that many of my new functions would not make sense if I tried to add if statements to handle each slightly different logic, it would be too burdensome and still need to repeat withing the code. I have detailed these instances in the comments in the refactored code.



```
### THIS HAS NOT BEEN REFACTORED, BUT SEE COMMENT LOGIC FOR REPETITIVE CODE ###
# even though this code is similar to the function below, they are processing different types of input, as such
# it would result in a single, very long function, with if statements to catch the differring logic
def parse_address_file():
    # global declarations to be populated by input
    global cat_value
    global item_type
    global quantity_value
    global num_addresses
    global valid
    filename = "request.csv"
    with open(filename, encoding='utf8') as input_file:
        csv_reader = csv.reader(input_file, delimiter=',')    # our delimiter is set as a "," since it's a csv file
        line_count = 0   # mostly used to track the first line of processing, but a good piece of data nonetheless
        for row in csv_reader:
            #checks to make sure we are evaluating the header row
            if line_count == 0:
                line_count += 1
                continue
            # otherwise, we will be parsing for our specific category, and pulling the data we need from the CSV.
            else:
                input_item_type = row[0]    # stores "Toys" for now
                user_category = row[1]      # process the category provided
                cat_value = user_category
                num_to_generate = row[2]    # process the number of entries to generate
                quantity_value = num_to_generate
                num_addresses.append(row[3])
                addresses.append(row[4])
        valid = 2
    return search_and_output()
```



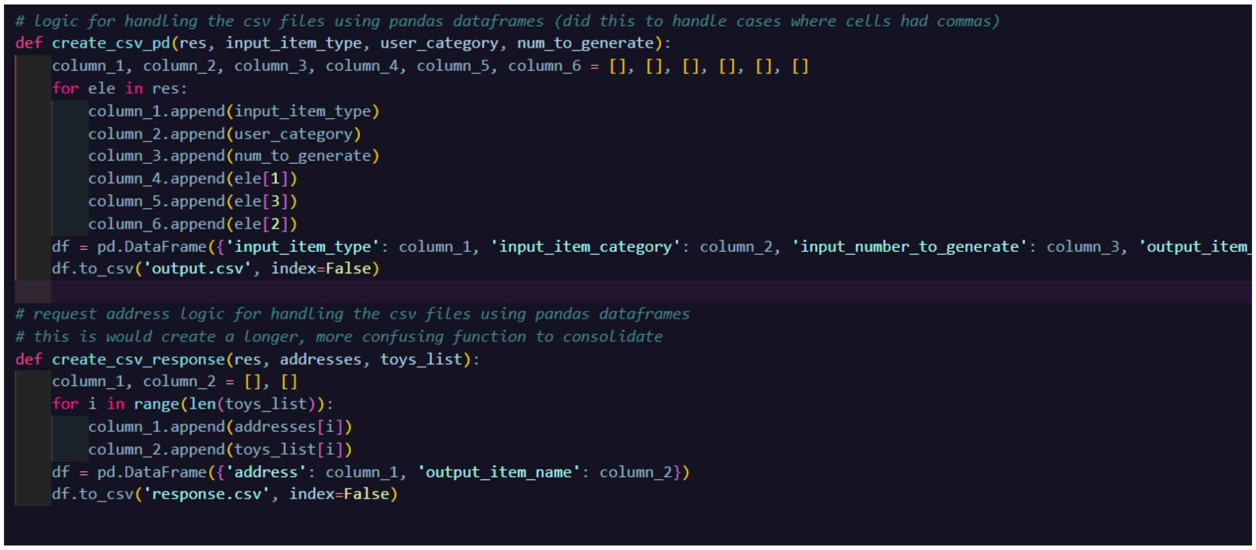
```
### THIS HAS NOT BEEN REFACTORED ###
# checks to see if an arg was recieved when the program was run.
def parse_input():
    global cat_value
    global item_type
    global quantity_value
    if len(sys.argv) == 2:
        filename = (sys.argv)[1]
        with open(filename, encoding='utf8') as input_file:
            csv_reader = csv.reader(input_file, delimiter=',')    #our delimiter is set as a "," since it's a csv file
            line_count = 0   # mostly used to track the first line of processing, but a good piece of data nonetheless
            for row in csv_reader:
                #checks to make sure we are evaluating the header row
                if line_count == 0:
                    line_count += 1
                    continue
                # otherwise, we will be parsing for our specific category, and pulling the data we need from the CSV.
                else:
                    input_item_type = row[0]  # stores "Toys" for now
                    user_category = row[1]     # process the category provided
                    cat_value = user_category
                    num_to_generate = row[2]     # process the number of entries to generate
                    quantity_value = num_to_generate
                    return True
    return False
```

These two blocks of code are very similar, but they serve very different cases. One is ONLY run on the start of the program, when an arg is passed and the other is run on a button press, and they have different values which would be processed and stored differently.

I believe my conventions are consistent across the program, naming conventions and parameters hold true, and items are named clearly. As such, the vague naming code smell would not apply either.

My only Long Comment smells are a result of the refactor, which I believe can't be dealt with since they provide information beyond the code.

CSV Logic Refactoring:

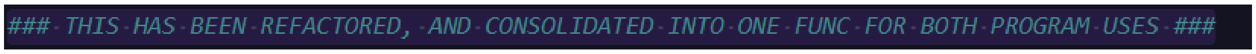


```
# logic for handling the csv files using pandas dataframes (did this to handle cases where cells had commas)
def create_csv_pd(res, input_item_type, user_category, num_to_generate):
    column_1, column_2, column_3, column_4, column_5, column_6 = [], [], [], [], [], []
    for ele in res:
        column_1.append(input_item_type)
        column_2.append(user_category)
        column_3.append(num_to_generate)
        column_4.append(ele[1])
        column_5.append(ele[3])
        column_6.append(ele[2])
    df = pd.DataFrame({'input_item_type': column_1, 'input_item_category': column_2, 'input_number_to_generate': column_3, 'output_item_
    df.to_csv('output.csv', index=False)

# request address logic for handling the csv files using pandas dataframes
# this is would create a longer, more confusing function to consolidate
def create_csv_response(res, addresses, toys_list):
    column_1, column_2 = [], []
    for i in range(len(toys_list)):
        column_1.append(addresses[i])
        column_2.append(toys_list[i])
    df = pd.DataFrame({'address': column_1, 'output_item_name': column_2})
    df.to_csv('response.csv', index=False)
```

I have elected to leave these as two different functions. Additionally, in my previous submission, there was a large block of commented out code, that has been deleted. The code referred to a non-pandas way of returning the csv.

The remaining refactoring can be clearly seen by my comments in the file



```
### THIS HAS BEEN REFACTORED, AND CONSOLIDATED INTO ONE FUNC FOR BOTH PROGRAM USES ###
```

As an example. In my submission, I have provided the code before and after refactoring and detailed where refactoring was performed so the review can view both files and know which areas to look for changes in.