Design Document

REQUIRED

- Latest version of Python

Dependencies

- https://pypi.org/project/pdfplumber/
- https://pypi.org/project/PyQt5/

PDFReader

Argument:

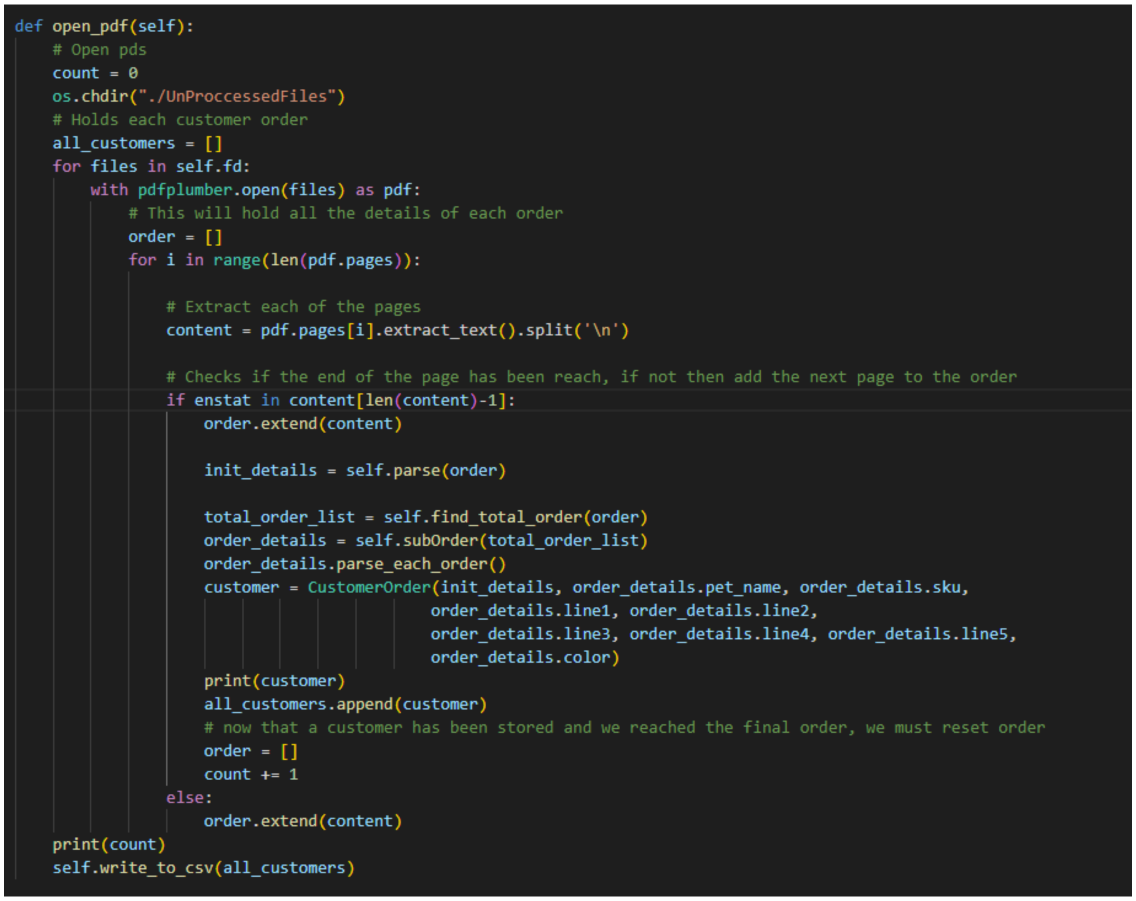
- A list of filenames in pdf format
- Name of the file to write to

Following methods:

- open_pfd()
- Write_to_csv(arg)
- subOrder(arg)
- find_order_num(arg)

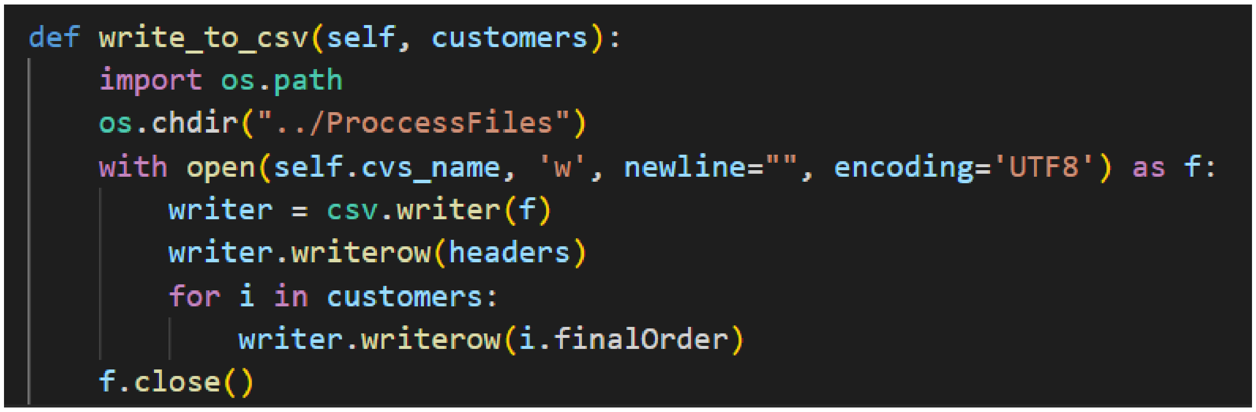
This is the main function that is handling each order. In this function we first look into the UnProcessedFiles directory to locate each pdf file. Once in there, iit will loop through each pdf file one at a time. Then it will loop through each page and gather one page contents at a time. At each page we have to check if the order is completed. To do this we checked if the end statement is contents. If so, then we have one fully completed order. If not then then we continue onto the next page.

Once we have the full order we call find_order_number which takes the contents of that order. Then call subOrder to find the sub order of each order. When that is completed it will return a Track object and the method parse_each_order() will be called. This will organize the data into appropriate SKU, Color, Line1,Line2,Line3,Line4,Line5. From there we will create a customer object with all of the information needed and then add the customer to a customer list. When we have each customer we then call write_to_csv, that will write all the customers into the csv file.



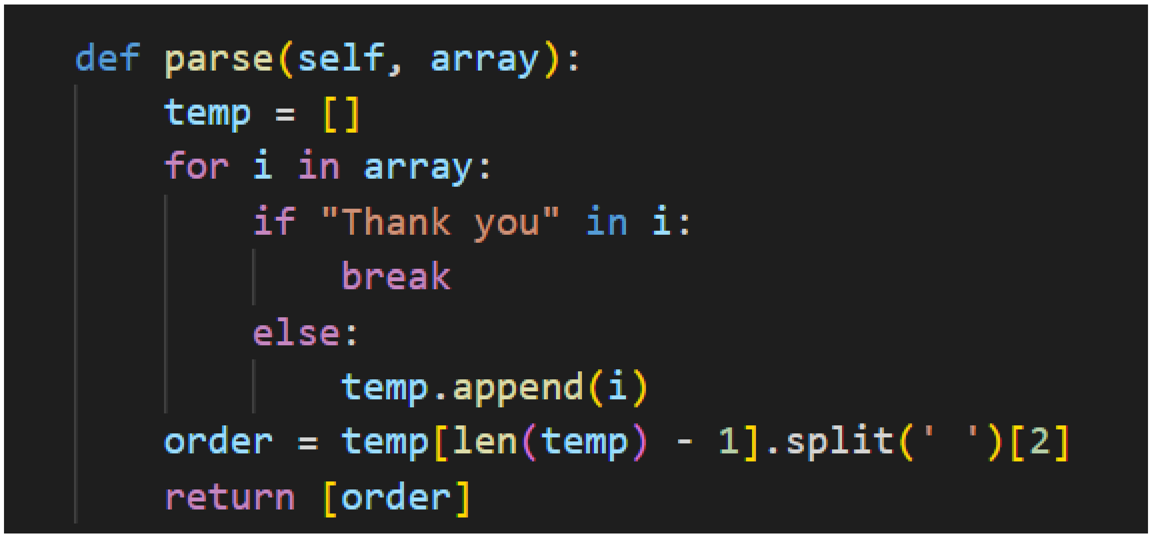
```
def open_pdf(self):
    # Open pds
    count = 0
    os.chdir("./UnProccessedFiles")
    # Holds each customer order
    all_customers = []
    for files in self.fd:
        with pdfplumber.open(files) as pdf:
            # This will hold all the details of each order
            order = []
            for i in range(len(pdf.pages)):

                # Extract each of the pages
                content = pdf.pages[i].extract_text().split('\n')

                # Checks if the end of the page has been reach, if not then add the next page to the order
                if enstat in content[len(content)-1]:
                    order.extend(content)

                    init_details = self.parse(order)

                    total_order_list = self.find_total_order(order)
                    order_details = self.subOrder(total_order_list)
                    order_details.parse_each_order()
                    customer = CustomerOrder(init_details, order_details.pet_name, order_details.sku,
                                             order_details.line1, order_details.line2,
                                             order_details.line3, order_details.line4, order_details.line5,
                                             order_details.color)
                    print(customer)
                    all_customers.append(customer)
                    # now that a customer has been stored and we reached the final order, we must reset order
                    order = []
                    count += 1
                else:
                    order.extend(content)
    print(count)
    self.write_to_csv(all_customers)
```

This function will take a list of customers and store the files in the ProcessFiles folder. When the file is being written, it will initially write the headers, which are in the following order: OrderNumber,Pet Names, SKU, Color, Line1,Line2,Line3,Line4,Line5. After that it will write each customer into that file.



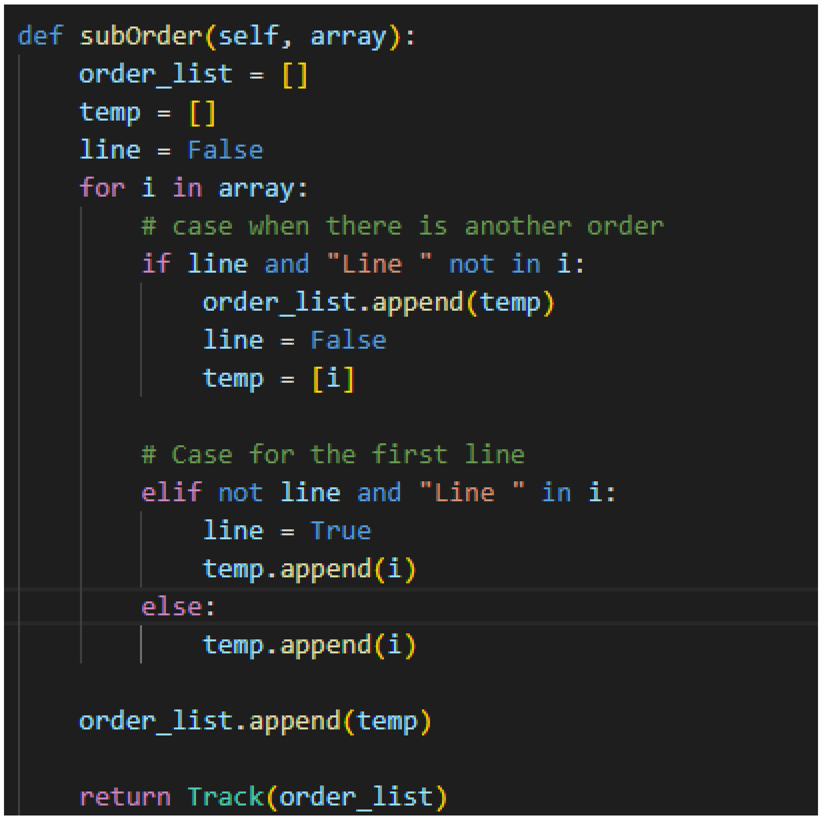
```
def write_to_csv(self, customers):
    import os.path
    os.chdir("../ProccessFiles")
    with open(self.cvs_name, 'w', newline="", encoding='UTF8') as f:
        writer = csv.writer(f)
        writer.writerow(headers)
        for i in customers:
            writer.writerow(i.finalOrder)
    f.close()
```

This function will find the order number for each customer. It does this by looking through each order and once it finds the word “Thank you” then the order number will be in the last index of that array.



```python
def parse(self, array):
    temp = []
    for i in array:
        if "Thank you" in i:
            break
        else:
            temp.append(i)
    order = temp[len(temp) - 1].split(' ')[2]
    return [order]
```

This function finds all the sub orders and appends each order into the order list. It checks whether to see if a line is in that order and when there is we know to continue adding to the order until we see no more line.



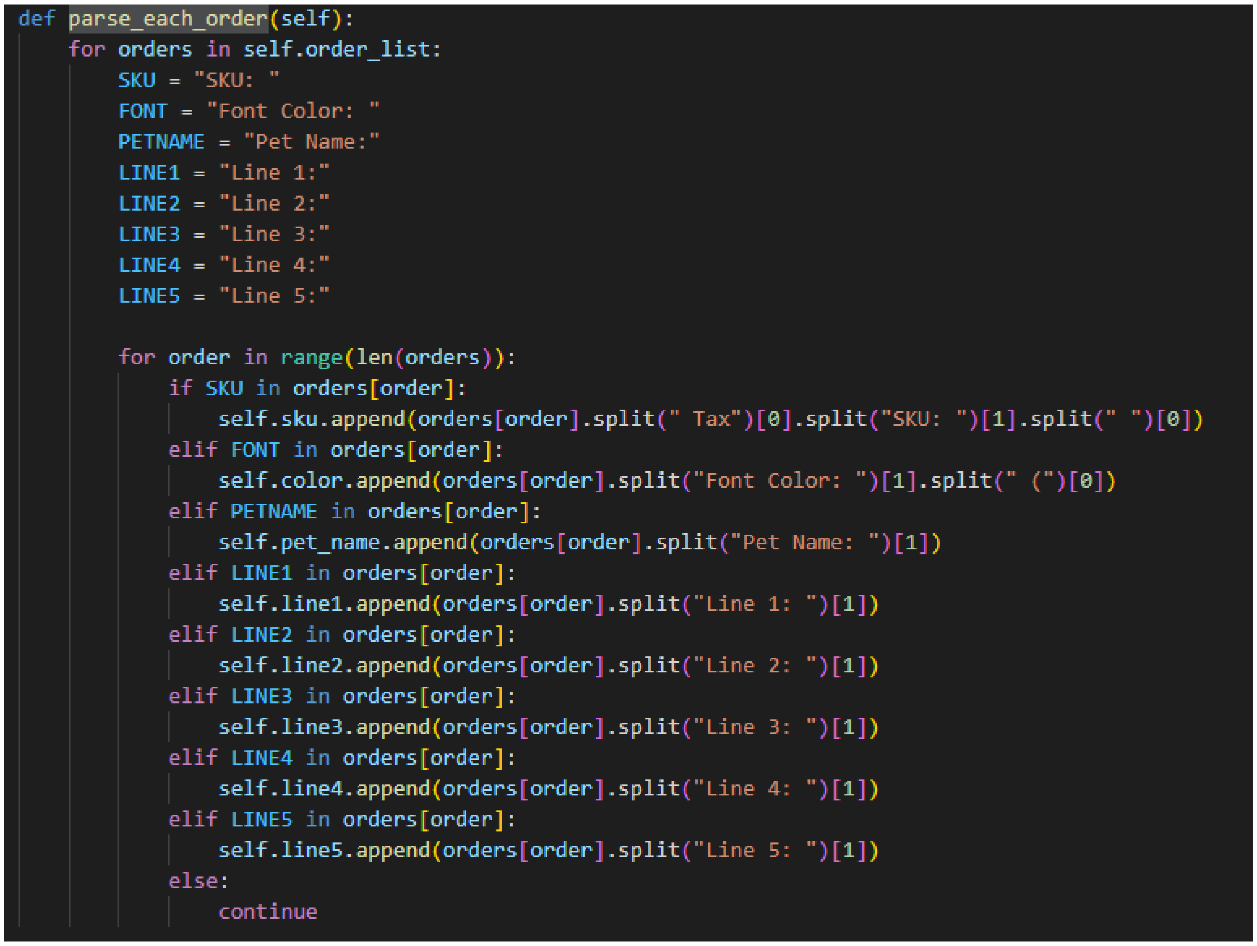
```python
def subOrder(self, array):
    order_list = []
    temp = []
    line = False
    for i in array:
        # case when there is another order
        if line and "Line " not in i:
            order_list.append(temp)
            line = False
            temp = [i]

        # Case for the first line
        elif not line and "Line " in i:
            line = True
            temp.append(i)
        else:
            temp.append(i)

    order_list.append(temp)

    return Track(order_list)
```

Track
Arguments: Order list

Following methods:

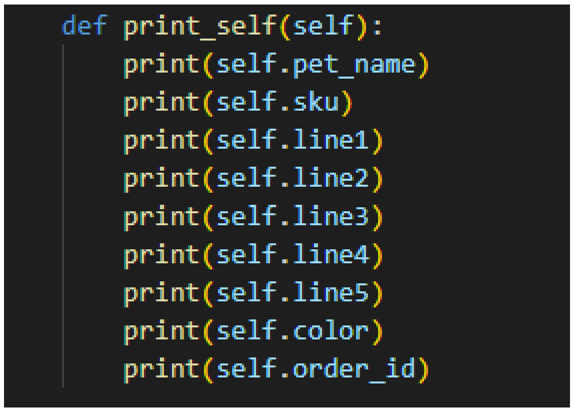
- parse_each_order()
- print-self

This function looks at each parsed sub-order and appends each order into its appropriate list.



```python
def parse_each_order(self):
    for orders in self.order_list:
        SKU = "SKU: "
        FONT = "Font Color: "
        PETNAME = "Pet Name:"
        LINE1 = "Line 1:"
        LINE2 = "Line 2:"
        LINE3 = "Line 3:"
        LINE4 = "Line 4:"
        LINE5 = "Line 5:"

        for order in range(len(orders)):
            if SKU in orders[order]:
                self.sku.append(orders[order].split(" Tax")[0].split("SKU: ")[1].split(" ")[0])
            elif FONT in orders[order]:
                self.color.append(orders[order].split("Font Color: ")[1].split(" (")[0])
            elif PETNAME in orders[order]:
                self.pet_name.append(orders[order].split("Pet Name: ")[1])
            elif LINE1 in orders[order]:
                self.line1.append(orders[order].split("Line 1: ")[1])
            elif LINE2 in orders[order]:
                self.line2.append(orders[order].split("Line 2: ")[1])
            elif LINE3 in orders[order]:
                self.line3.append(orders[order].split("Line 3: ")[1])
            elif LINE4 in orders[order]:
                self.line4.append(orders[order].split("Line 4: ")[1])
            elif LINE5 in orders[order]:
                self.line5.append(orders[order].split("Line 5: ")[1])
            else:
                continue
```

This function prints all of the self variables. Note not used in main programming only used for testing.



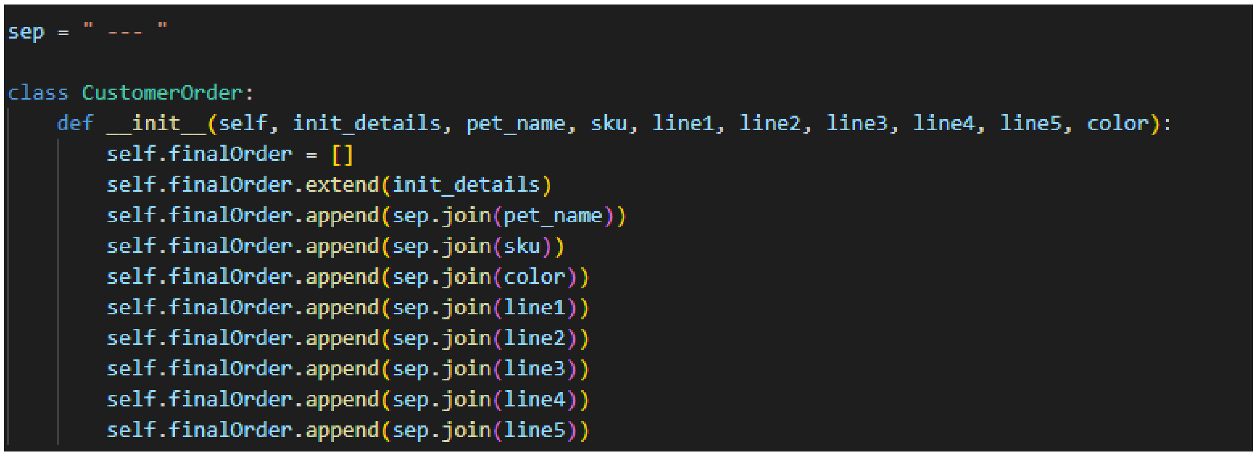
```python
def print_self(self):
    print(self.pet_name)
    print(self.sku)
    print(self.line1)
    print(self.line2)
    print(self.line3)
    print(self.line4)
    print(self.line5)
    print(self.color)
    print(self.order_id)
```

Customer

Arguments: order numbers, pet names, skus, line1,line2,line3,line4,line5, color

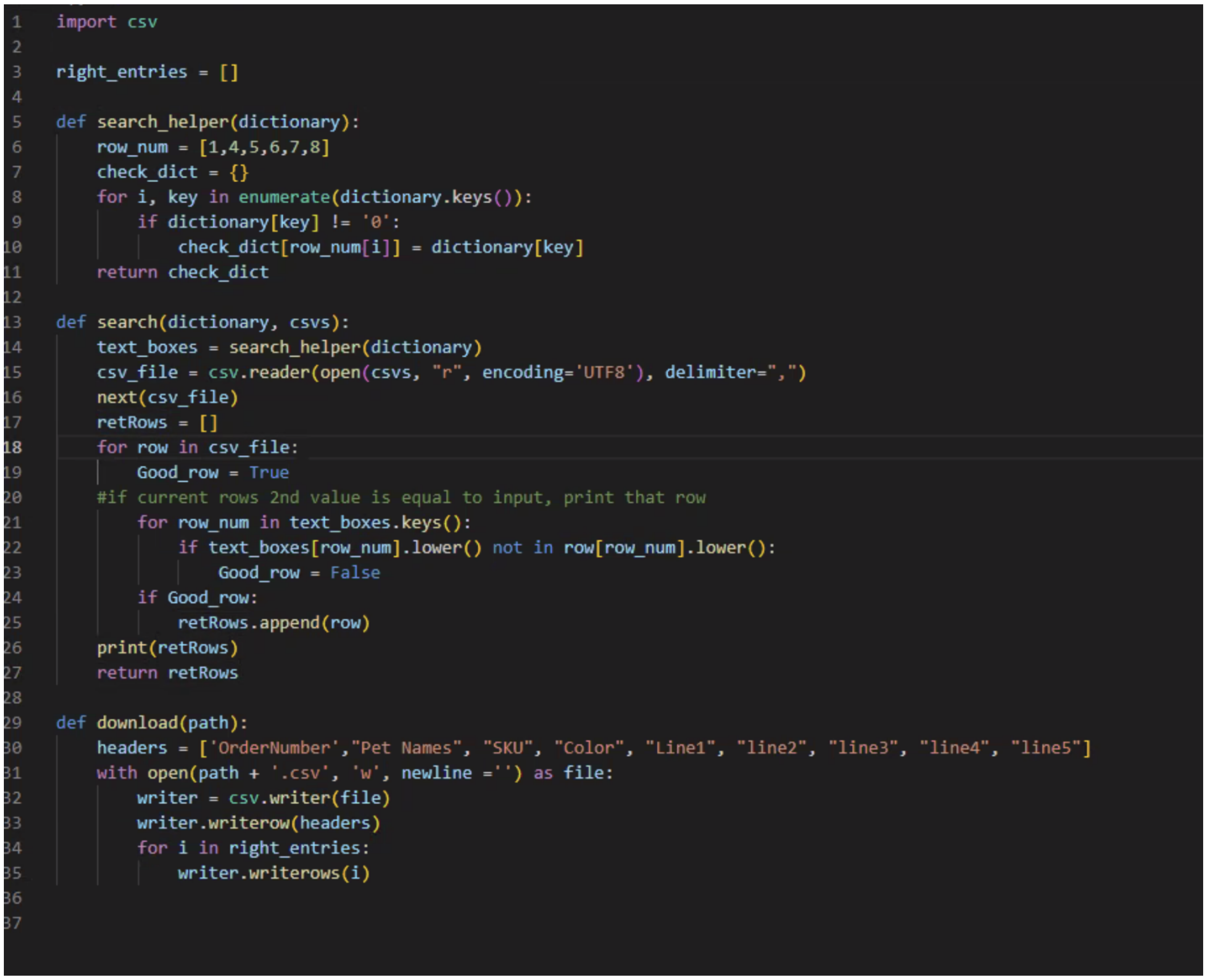
- None just appends everything into a list using a separator

The separator is used to concatenate multiple sub orders into one. This will be placed into the final list which will be used to write in the file.



```python
sep = " --- "

class CustomerOrder:
    def __init__(self, init_details, pet_name, sku, line1, line2, line3, line4, line5, color):
        self.finalOrder = []
        self.finalOrder.extend(init_details)
        self.finalOrder.append(sep.join(pet_name))
        self.finalOrder.append(sep.join(sku))
        self.finalOrder.append(sep.join(color))
        self.finalOrder.append(sep.join(line1))
        self.finalOrder.append(sep.join(line2))
        self.finalOrder.append(sep.join(line3))
        self.finalOrder.append(sep.join(line4))
        self.finalOrder.append(sep.join(line5))
```

—



```python
import csv

right_entries = []

def search_helper(dictionary):
    row_num = [1,4,5,6,7,8]
    check_dict = {}
    for i, key in enumerate(dictionary.keys()):
        if dictionary[key] != '0':
            check_dict[row_num[i]] = dictionary[key]
    return check_dict

def search(dictionary, csvs):
    text_boxes = search_helper(dictionary)
    csv_file = csv.reader(open(csvs, "r", encoding='UTF8'), delimiter=",")
    next(csv_file)
    retRows = []
    for row in csv_file:
        Good_row = True
    #if current rows 2nd value is equal to input, print that row
        for row_num in text_boxes.keys():
            if text_boxes[row_num].lower() not in row[row_num].lower():
                Good_row = False
        if Good_row:
            retRows.append(row)
    print(retRows)
    return retRows

def download(path):
    headers = ['OrderNumber',"Pet Names", "SKU", "Color", "Line1", "line2", "line3", "line4", "line5"]
    with open(path + '.csv', 'w', newline ='') as file:
        writer = csv.writer(file)
        writer.writerow(headers)
        for i in right_entries:
            writer.writerows(i)
```

The search.py file is meant to implement the search functionality by which users can filter through the processed version of the CSV files. Upon filtering through the processed version of the amazon orders, a user will be able to select the orders that they want to store and download said orders as a csv using the “download” function.

Search_Helper

Arguments: Dictionary
Returns: Dictionary

The search helper function will iterate through the dictionary passed in and will map each CSV index to the corresponding text value that was passed in. This is used to map the text that was written in textboxes to the indices of the csv file. The indices are manually typed in under row_num where 1 represents the pet name index of the file and 8 represents the 5th line.

Search

Arguments: Dictionary and CSV
Returns: List of proper rows

The search function is the filtration system and actually implements the search functionality. The search function takes in a dictionary which maps the text typed into the text box (value) to its label (key) & the CSV file that was processed.

The CSV file is read and then we check each row of the CSV file to see if it matches with any of the values type into the textboxes. In order to do this we call our Search_Helper function on the dictionary argument passed in which then returns a dictionary that points to the index of the csv file that corresponds to the text box label (1 - petname, 4 - line 1, 5 - line 2 etc.). The text that is written in the textboxes is matched to the row of each csv file and if a match is found then it is added to the “good row” list. At the end all of the good rows are returned.

Download

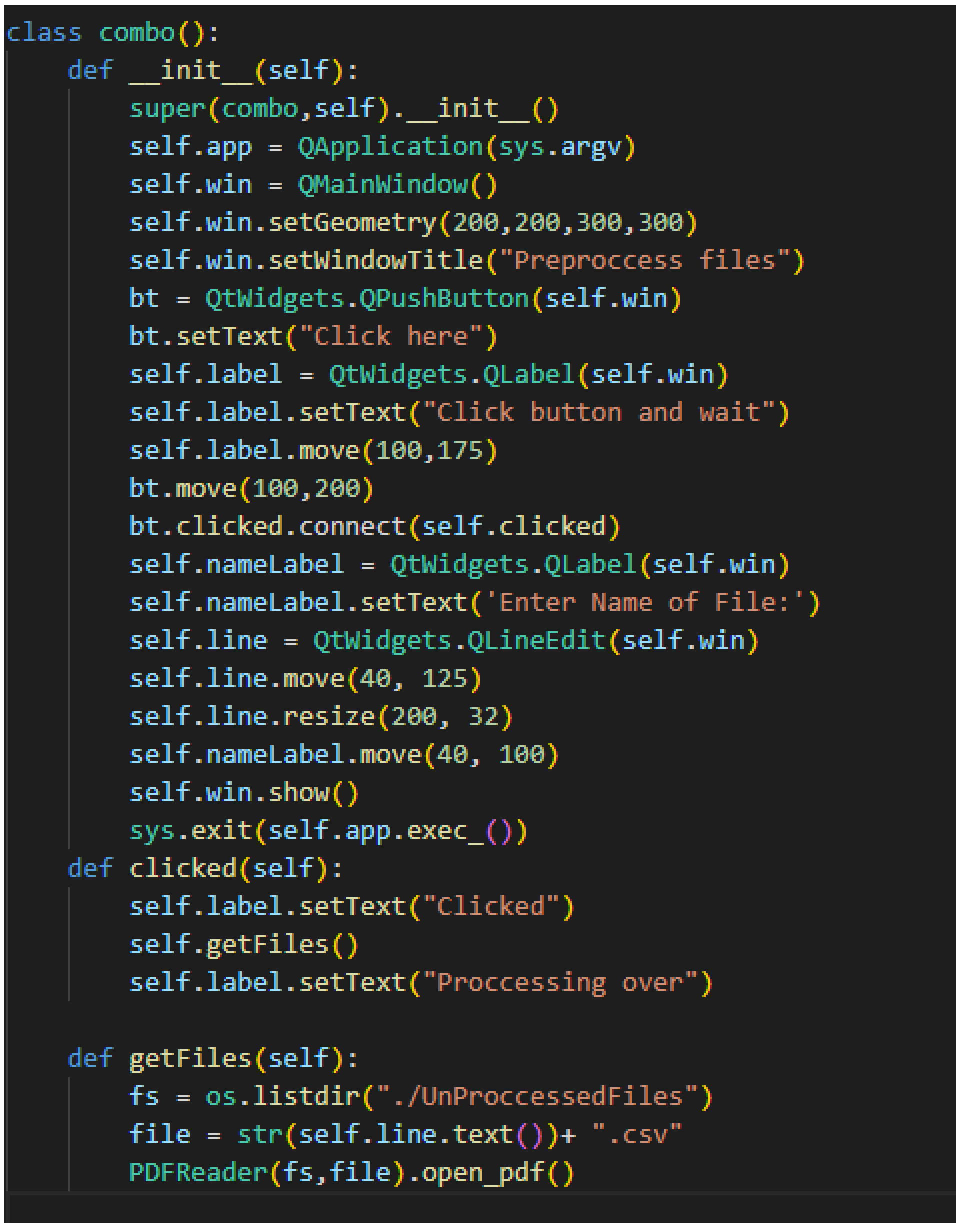
Arguments: path of the file to download
Returns: Written file

The download function is called once the get final csv button is pressed. This function will grab all of the CSV’s that the user requested and write them to a file name of their choice dependent upon the path that was passed in.

combo
Arguments:none
Following methods

- click()
- getFiles()

The click function is called when a button is clicked and then calls the getFiles method which, Gets a list of the name of files in the UnproccessedFiles directory. Then it calls the PDFReader which handles the customer orders.



```python
class combo():
    def __init__(self):
        super(combo,self).__init__()
        self.app = QApplication(sys.argv)
        self.win = QMainWindow()
        self.win.setGeometry(200,200,300,300)
        self.win.setWindowTitle("Preproccess files")
        bt = QtWidgets.QPushButton(self.win)
        bt.setText("Click here")
        self.label = QtWidgets.QLabel(self.win)
        self.label.setText("Click button and wait")
        self.label.move(100,175)
        bt.move(100,200)
        bt.clicked.connect(self.clicked)
        self.nameLabel = QtWidgets.QLabel(self.win)
        self.nameLabel.setText('Enter Name of File:')
        self.line = QtWidgets.QLineEdit(self.win)
        self.line.move(40, 125)
        self.line.resize(200, 32)
        self.nameLabel.move(40, 100)
        self.win.show()
        sys.exit(self.app.exec_())
    def clicked(self):
        self.label.setText("Clicked")
        self.getFiles()
        self.label.setText("Proccessing over")

    def getFiles(self):
        fs = os.listdir("./UnProccessedFiles")
        file = str(self.line.text())+ ".csv"
        PDFReader(fs,file).open_pdf()
```

Bugs
Possible bugs if the customer inputs any of these strings as a dogtag :

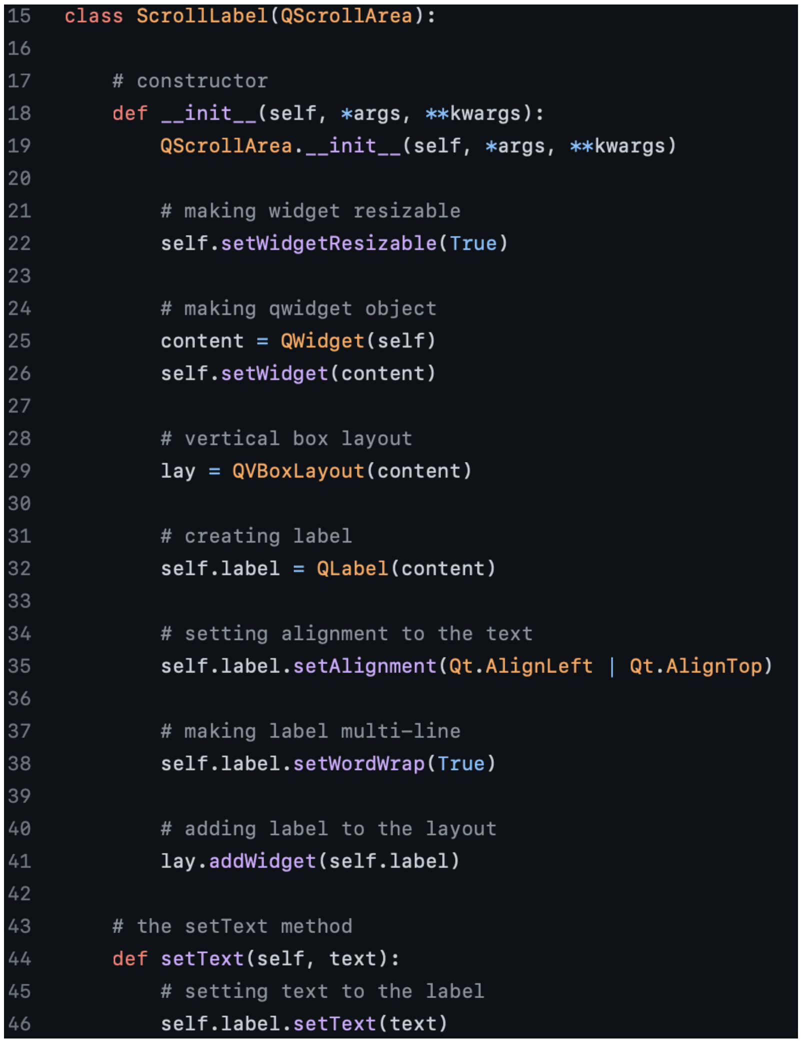
- SKU:
- Font Color:
- Pet Name:
- Line 1:
- Line 2
- Line 3:
- Line 4:
- Line 5:
- Tax
- click \"Contact the Seller.\

More Possible Bugs:

- The file in the UnprocessedFiles directory is not a correct pdf.
- If the file contains a date and time when the file was created.
    - This can be avoided by checking whether the first page in the pdf contains the date and time in the upper left corner.
- If you enter information and want to re enter another tag you might need to click cancel or tag won't appear.

MAIN

Class ScrollLabel(QScrollArea):



```
class ScrollLabel(QScrollArea):

    # constructor
    def __init__(self, *args, **kwargs):
        QScrollArea.__init__(self, *args, **kwargs)

        # making widget resizable
        self.setWidgetResizable(True)

        # making qwidget object
        content = QWidget(self)
        self.setWidget(content)

        # vertical box layout
        lay = QVBoxLayout(content)

        # creating label
        self.label = QLabel(content)

        # setting alignment to the text
        self.label.setAlignment(Qt.AlignLeft | Qt.AlignTop)

        # making label multi-line
        self.label.setWordWrap(True)

        # adding label to the layout
        lay.addWidget(self.label)

    # the setText method
    def setText(self, text):
        # setting text to the label
        self.label.setText(text)
```

This class ScrollLabel is a skeleton of the GUI it sets the dimensions in any way that was missed out in the dialogue.ui file.
Methods:

- __init__()
- setText(self,text)

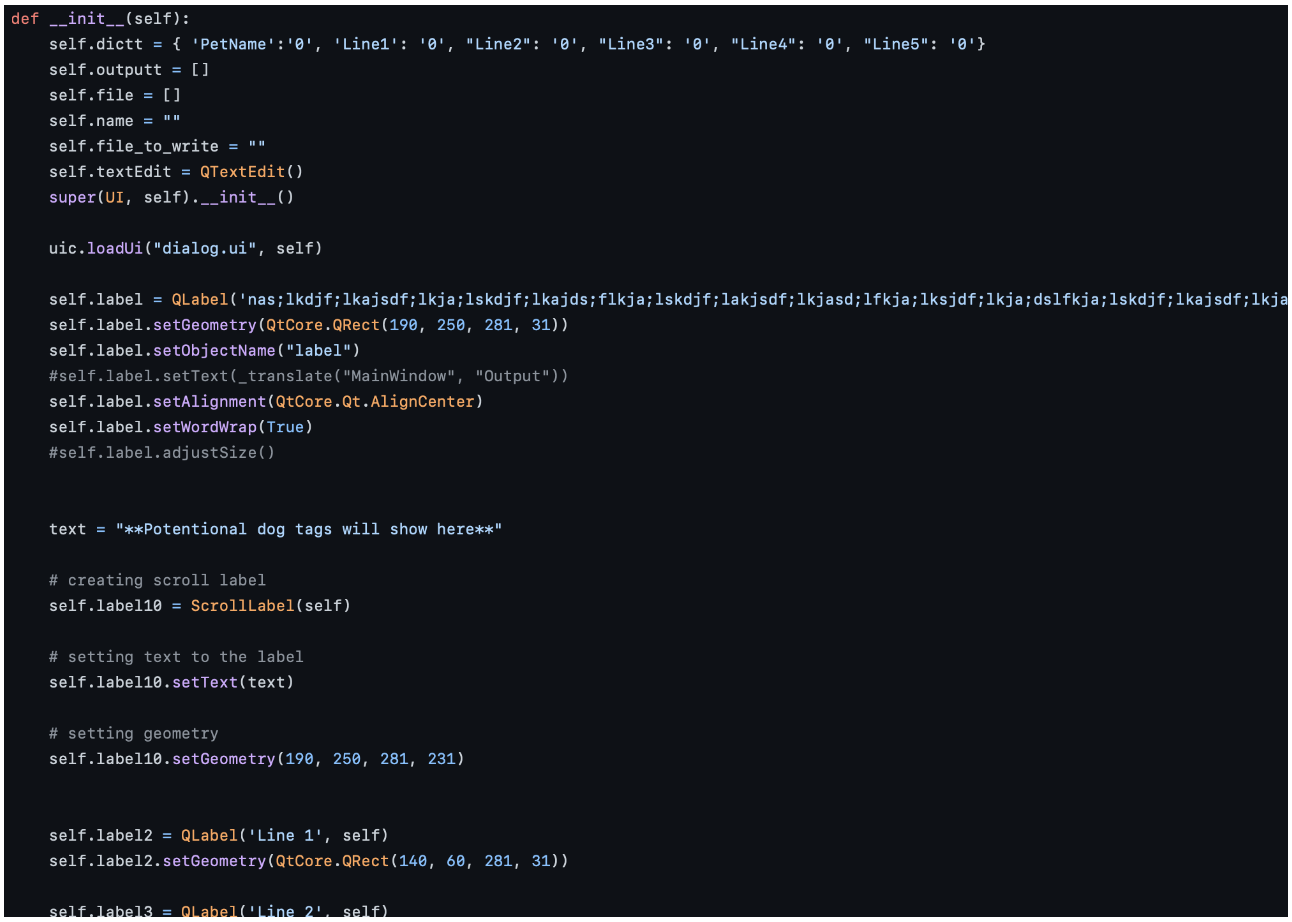
__init__() contains self and args that are needed to get the GUI to run on your machine.
setText(self, text) This method will set the text of your label with whatever is inputted.

Class UI(QMainWindow):

This class contains most of the functionality and initialization for the textboxes, buttons, etc on the GUI.

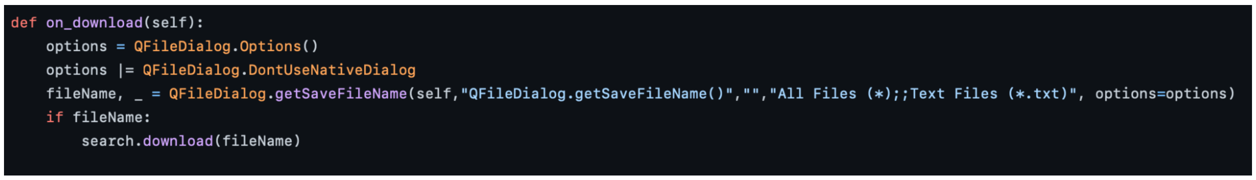
Following methods:

- __init__(self)
- on_download(self)
- on_cancel(self)
- on_enter(self)
- on_correct_info(self)
- on_openfile(self)



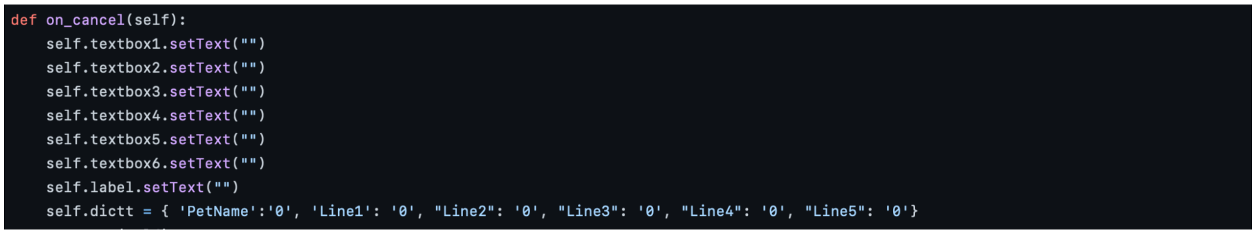
```python
def __init__(self):
    self.dictt = { 'PetName':'0', 'Line1': '0', "Line2": '0', "Line3": '0', "Line4": '0', "Line5": '0'}
    self.outputt = []
    self.file = []
    self.name = ""
    self.file_to_write = ""
    self.textEdit = QTextEdit()
    super(UI, self).__init__()

    uic.loadUi("dialog.ui", self)

    self.label = QLabel('nas;lkdjf;lkajsdf;lkja;lskdjf;lkajds;flkja;lskdjf;lakjsdf;lkjasd;lfkja;lksjdf;lkja;dslfkja;lskdjf;lkajsdf;lkja
    self.label.setGeometry(QtCore.QRect(190, 250, 281, 31))
    self.label.setObjectName("label")
    #self.label.setText(_translate("MainWindow", "Output"))
    self.label.setAlignment(QtCore.Qt.AlignCenter)
    self.label.setWordWrap(True)
    #self.label.adjustSize()


    text = "**Potentional dog tags will show here**"

    # creating scroll label
    self.label10 = ScrollLabel(self)

    # setting text to the label
    self.label10.setText(text)

    # setting geometry
    self.label10.setGeometry(190, 250, 281, 231)


    self.label2 = QLabel('Line 1', self)
    self.label2.setGeometry(QtCore.QRect(140, 60, 281, 31))

    self.label3 = QLabel('Line 2', self)
```

This method initializes the GUI in the final stages before it loads up on your machine. This contains some data to initialize for other methods in this class as well as the initialization and setup of the different labels, textboxes, and buttons for the GUI. This also connects certain buttons of the GUI to other methods in this class to take care of that specific functionality.



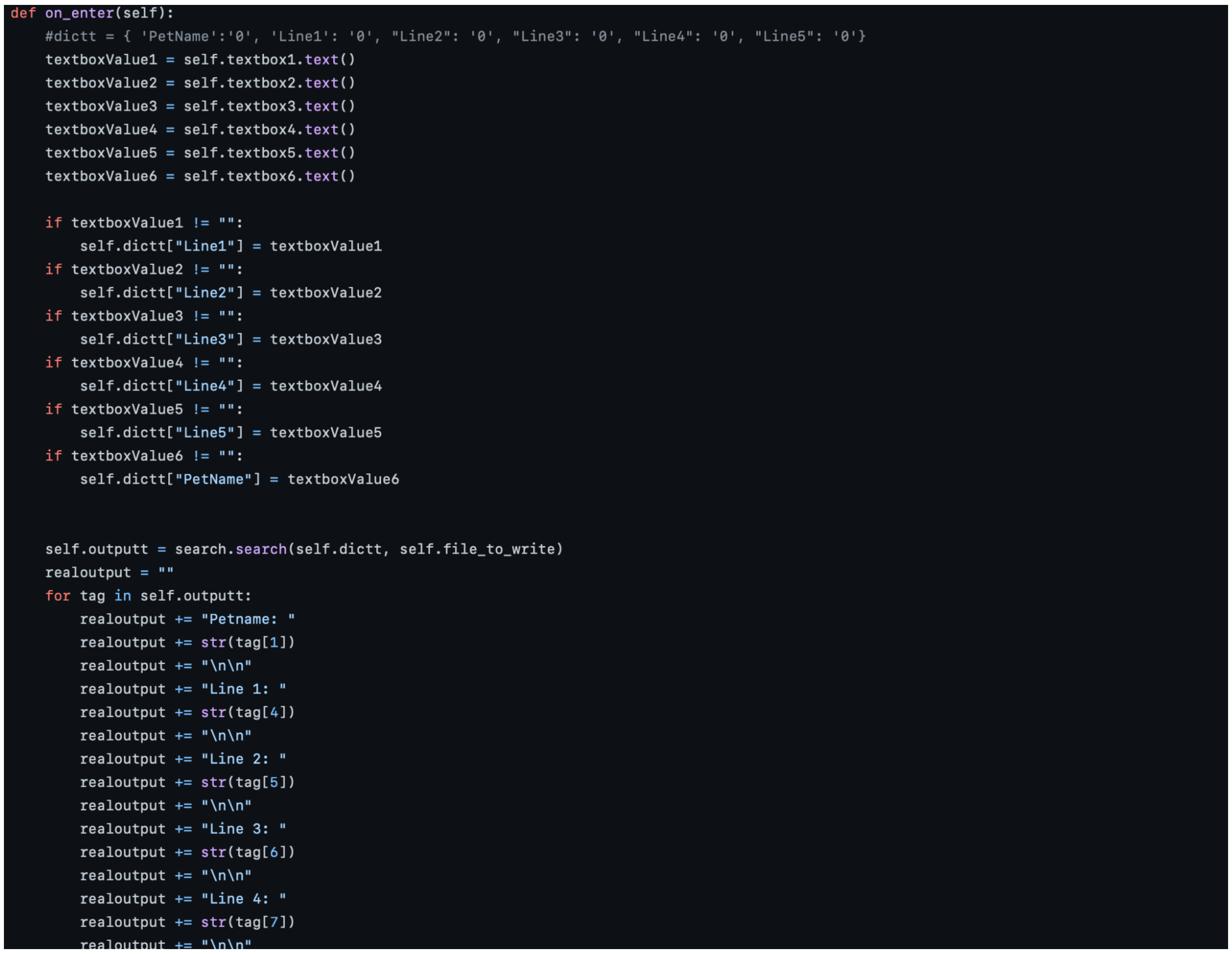
```python
def on_download(self):
    options = QFileDialog.Options()
    options |= QFileDialog.DontUseNativeDialog
    fileName, _ = QFileDialog.getSaveFileName(self,"QFileDialog.getSaveFileName()","","All Files (*);;Text Files (*.txt)", options=options)
    if fileName:
        search.download(fileName)
```

This method allows you to download the final csv with the information of all the dog tags you “scanned”. This will save it in your computer in your desired place and pass along the filename to search file for download to be used for other purposes as well. See search documentation for elaboration. This method is called when the “Get Final CSV” button is clicked.



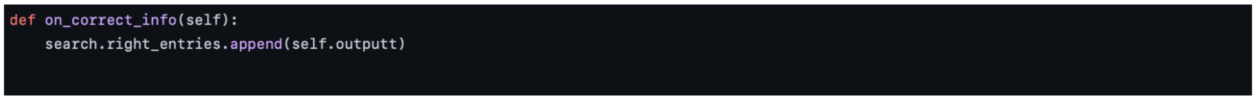
```
def on_cancel(self):
    self.textbox1.setText("")
    self.textbox2.setText("")
    self.textbox3.setText("")
    self.textbox4.setText("")
    self.textbox5.setText("")
    self.textbox6.setText("")
    self.label.setText("")
    self.dictt = { 'PetName':'0', 'Line1': '0', "Line2": '0', "Line3": '0', "Line4": '0', "Line5": '0'}
```

The on_cancel method is call upon when the “cancel” button is clicked. In the event of a mistake or for any reason you want to clear your entry of the dog tag this method will clear all textboxes for data input with empty strings. Also, this method will reset any other dictionaries or variables needed for the dog tag “scanning” process.



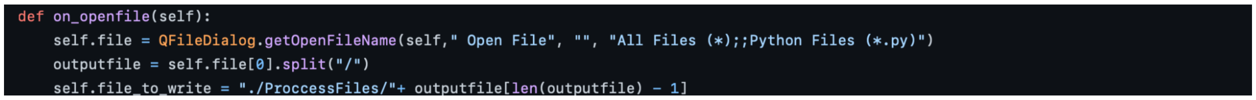
```
def on_enter(self):
    #dictt = { 'PetName':'0', 'Line1': '0', "Line2": '0', "Line3": '0', "Line4": '0', "Line5": '0'}
    textboxValue1 = self.textbox1.text()
    textboxValue2 = self.textbox2.text()
    textboxValue3 = self.textbox3.text()
    textboxValue4 = self.textbox4.text()
    textboxValue5 = self.textbox5.text()
    textboxValue6 = self.textbox6.text()

    if textboxValue1 != "":
        self.dictt["Line1"] = textboxValue1
    if textboxValue2 != "":
        self.dictt["Line2"] = textboxValue2
    if textboxValue3 != "":
        self.dictt["Line3"] = textboxValue3
    if textboxValue4 != "":
        self.dictt["Line4"] = textboxValue4
    if textboxValue5 != "":
        self.dictt["Line5"] = textboxValue5
    if textboxValue6 != "":
        self.dictt["PetName"] = textboxValue6


    self.outputt = search.search(self.dictt, self.file_to_write)
    realoutput = ""
    for tag in self.outputt:
        realoutput += "Petname: "
        realoutput += str(tag[1])
        realoutput += "\n\n"
        realoutput += "Line 1: "
        realoutput += str(tag[4])
        realoutput += "\n\n"
        realoutput += "Line 2: "
        realoutput += str(tag[5])
        realoutput += "\n\n"
        realoutput += "Line 3: "
        realoutput += str(tag[6])
        realoutput += "\n\n"
        realoutput += "Line 4: "
        realoutput += str(tag[7])
        realoutput += "\n\n"
```

This method is called upon when the “enter” button is clicked. Although it looks long and messy this method is very simple. In the first couple sections it combines all inputs from all the input boxes and pairs them up with “petname”, “line1”, “line2”, etc… for all inputs in a dictionary. This dictionary is then passed to the search functions to determine if these inputs match any inputs from the initial csv input from amazon. The search functions return a list of dog tags that match your user inputs. The method finishes off by running a for loop over all dog tags that matched and concatenates them into a string nicely to be outputted to the screen on the GUI.



```
def on_correct_info(self):
    search.right_entries.append(self.outputt)
```

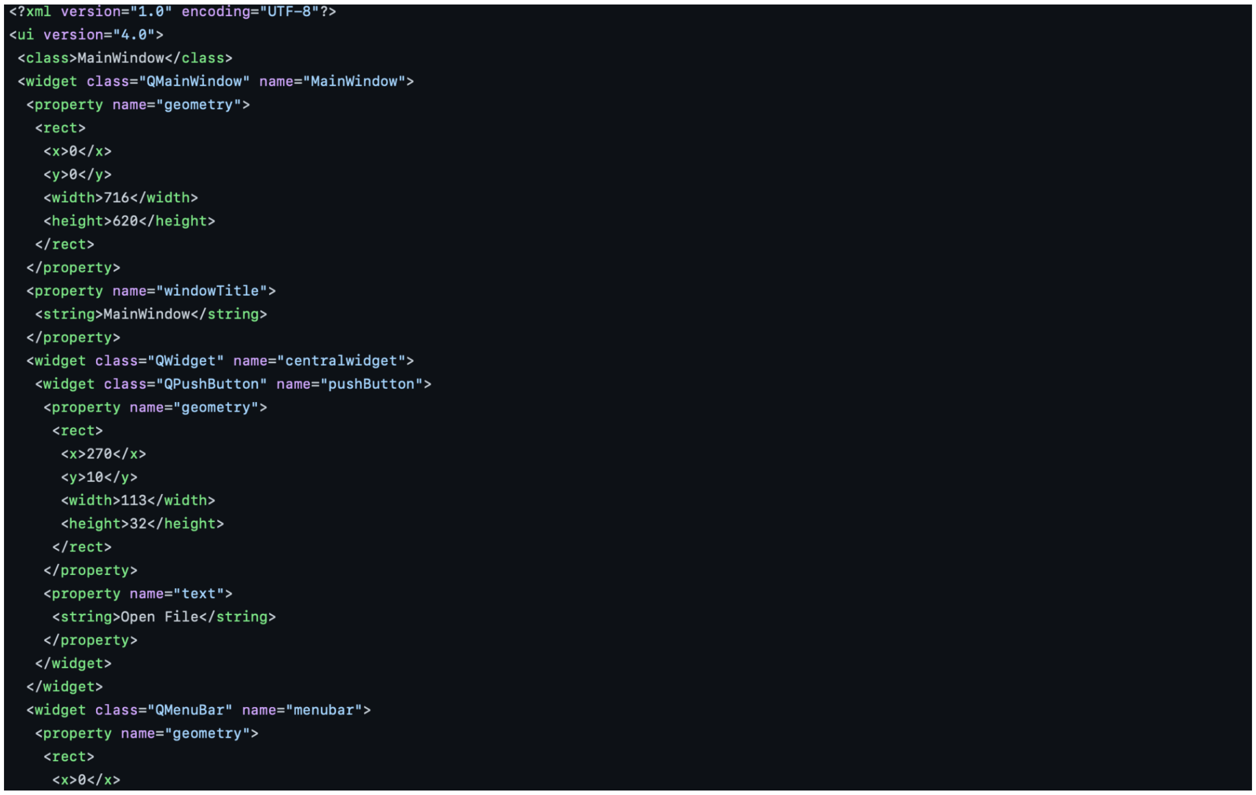
This method will take the dog tag that was inputted and matched in the system and append it to a list of all the other dog tags “scanned” and matched in the system. The dog tag will include all important information needed.



```
def on_openfile(self):
    self.file = QFileDialog.getOpenFileName(self," Open File", "", "All Files (*);;Python Files (*.py)")
    outputfile = self.file[0].split("/")
    self.file_to_write = "./ProccessFiles/"+ outputfile[len(outputfile) - 1]
```

This method will allow you to choose which processed file(made from multiple pre-processed files). To choose from and search/scan from.

Dialog



```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>716</width>
    <height>620</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>MainWindow</string>
  </property>
  <widget class="QWidget" name="centralwidget">
   <widget class="QPushButton" name="pushButton">
    <property name="geometry">
     <rect>
      <x>270</x>
      <y>10</y>
      <width>113</width>
      <height>32</height>
     </rect>
    </property>
    <property name="text">
     <string>Open File</string>
    </property>
   </widget>
  </widget>
  <widget class="QMenuBar" name="menubar">
   <property name="geometry">
    <rect>
     <x>0</x>
```

This file was used as a very early stage skeleton for the GUI. Since the beginning of creating this UI we have turned away from this method as it is easier to handle in the main file.Only very few parts of the GUI are created here except the skeleton GUI.