Group 5 - Programming Project Presentation

Created by Olivia B, Benjamin L, Noah H1, Elspeth CJ

Link to the project: https://github.com/LittleDinoCake/CF-Project

Overview:

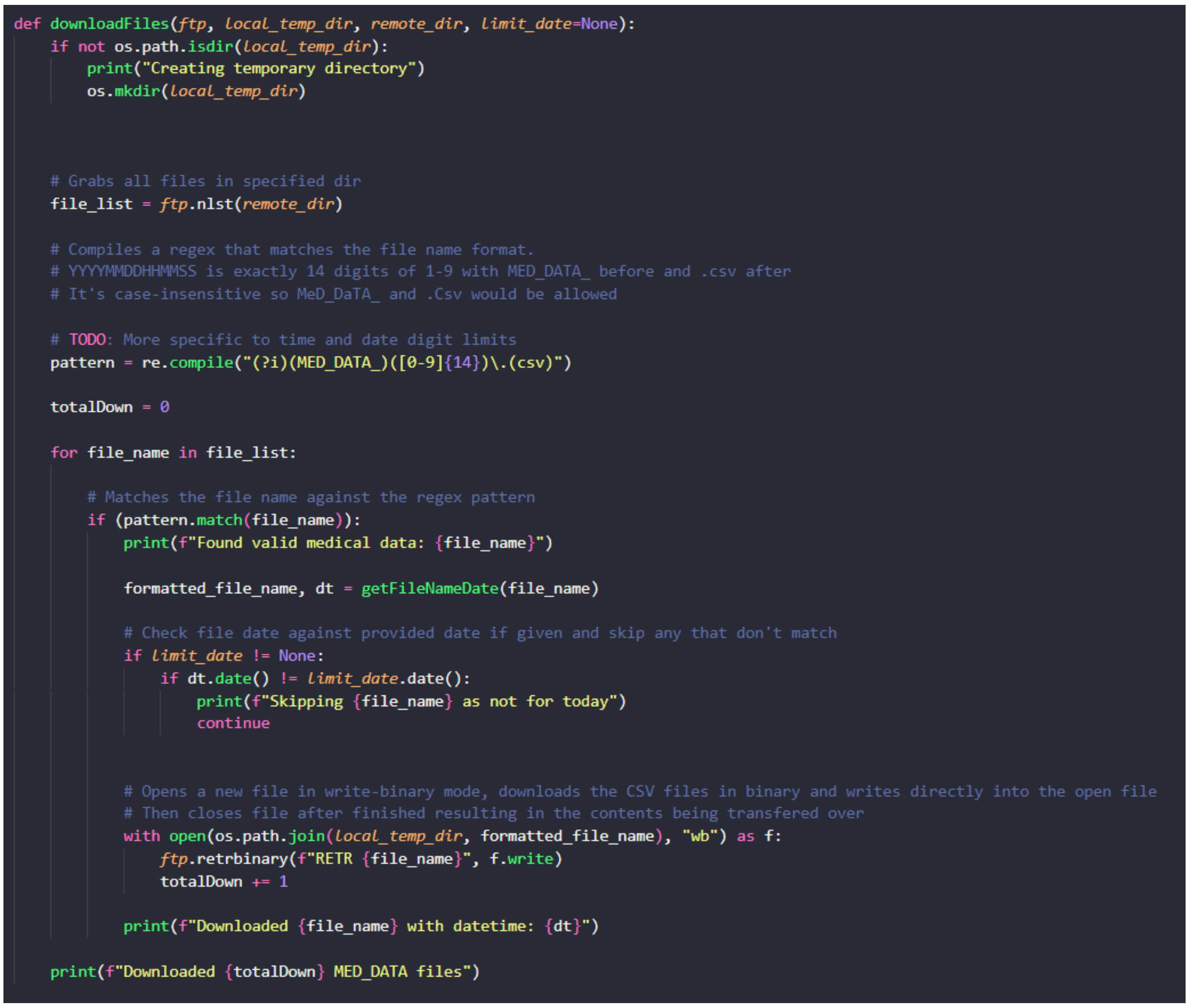
- We were asked to create a program allowing a University Medical School to be able to download pharmaceutical trial data from a trusted scientific partner for medical research
- The files are stored on an FTP server, and there should be both command-line options and an interactive front end
- There should also be error checking and logging for common issues, for example missing columns, or empty files
- As a team we chose to use Python in our solution, and work on the assumption that the users are able to use the command line to download any missing Python modules

Accessing the FTP server:

ftpmodule.py and run.py

The user creates or specify a local directory, into which files are downloaded.

All the files are collected from the server, but only the ones matching the user’s inputted date are downloaded to their machine (Figure 1).

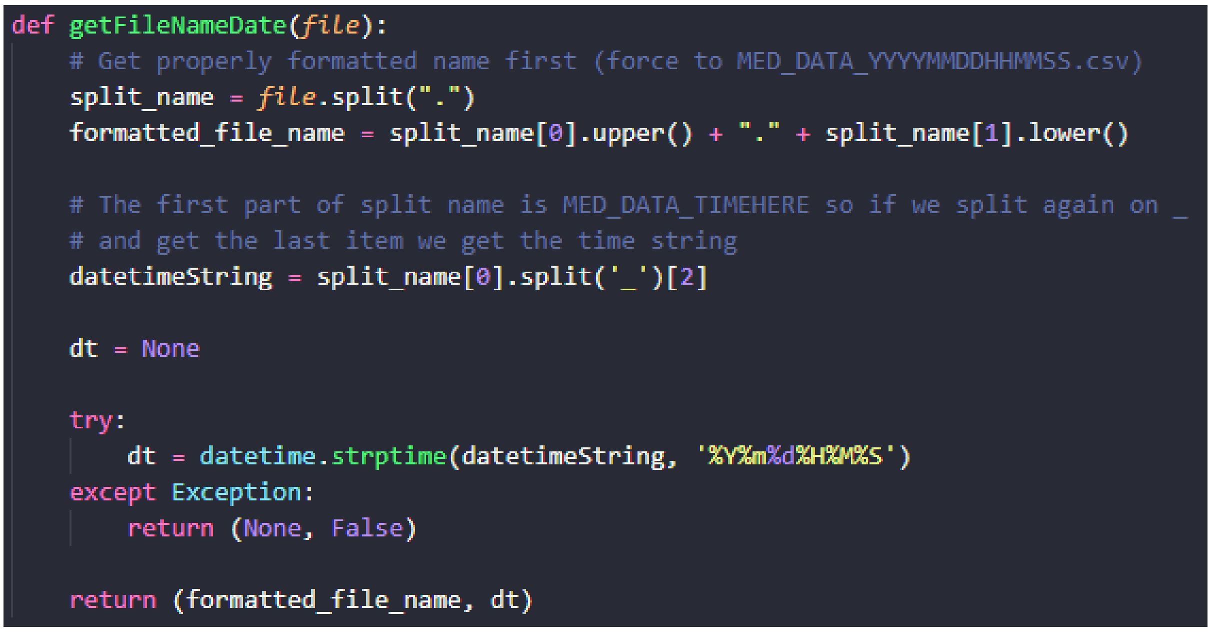


```python
def downloadFiles(ftp, local_temp_dir, remote_dir, limit_date=None):
    if not os.path.isdir(local_temp_dir):
        print("Creating temporary directory")
        os.mkdir(local_temp_dir)



    # Grabs all files in specified dir
    file_list = ftp.nlst(remote_dir)

    # Compiles a regex that matches the file name format.
    # YYYYMMDDHHMMSS is exactly 14 digits of 1-9 with MED_DATA_ before and .csv after
    # It's case-insensitive so MeD_DaTA_ and .Csv would be allowed

    # TODO: More specific to time and date digit limits
    pattern = re.compile("(?i)(MED_DATA_)([0-9]{14})\.(csv)")

    totalDown = 0

    for file_name in file_list:

        # Matches the file name against the regex pattern
        if (pattern.match(file_name)):
            print(f"Found valid medical data: {file_name}")

            formatted_file_name, dt = getFileNameDate(file_name)

            # Check file date against provided date if given and skip any that don't match
            if limit_date != None:
                if dt.date() != limit_date.date():
                    print(f"Skipping {file_name} as not for today")
                    continue


            # Opens a new file in write-binary mode, downloads the CSV files in binary and writes directly into the open file
            # Then closes file after finished resulting in the contents being transfered over
            with open(os.path.join(local_temp_dir, formatted_file_name), "wb") as f:
                ftp.retrbinary(f"RETR {file_name}", f.write)
                totalDown += 1

            print(f"Downloaded {file_name} with datetime: {dt}")

    print(f"Downloaded {totalDown} MED_DATA files")
```

FIGURE 1

Then the given file name is split up, so the program can collect the date.

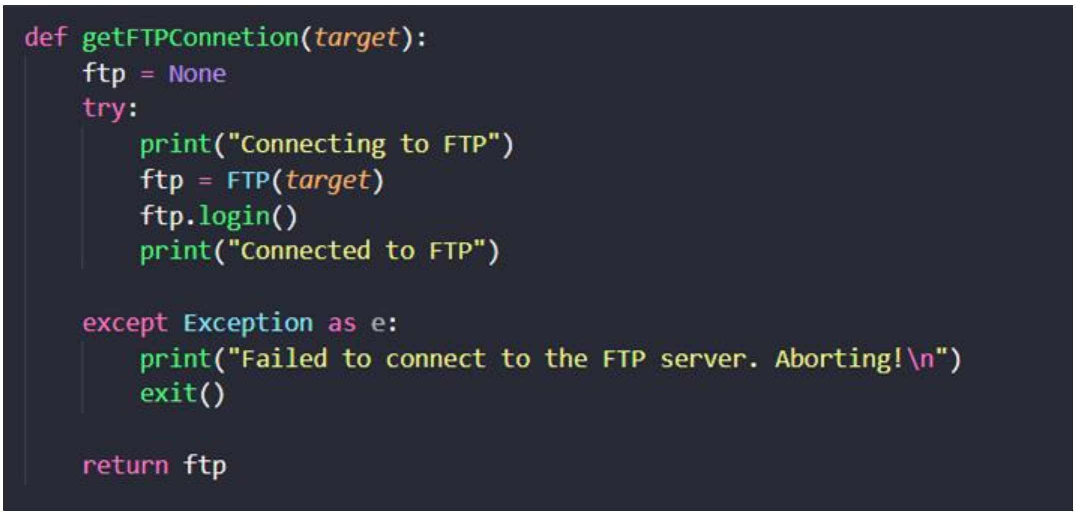
This function is used in downloadFiles() to check available files for the desired one (Figure 2).



```python
def getFileNameDate(file):
    # Get properly formatted name first (force to MED_DATA_YYYYMMDDHHMMSS.csv)
    split_name = file.split(".")
    formatted_file_name = split_name[0].upper() + "." + split_name[1].lower()

    # The first part of split name is MED_DATA_TIMEHERE so if we split again on _
    # and get the last item we get the time string
    datetimeString = split_name[0].split('_')[2]

    dt = None

    try:
        dt = datetime.strptime(datetimeString, '%Y%m%d%H%M%S')
    except Exception:
        return (None, False)

    return (formatted_file_name, dt)
```

FIGURE 2

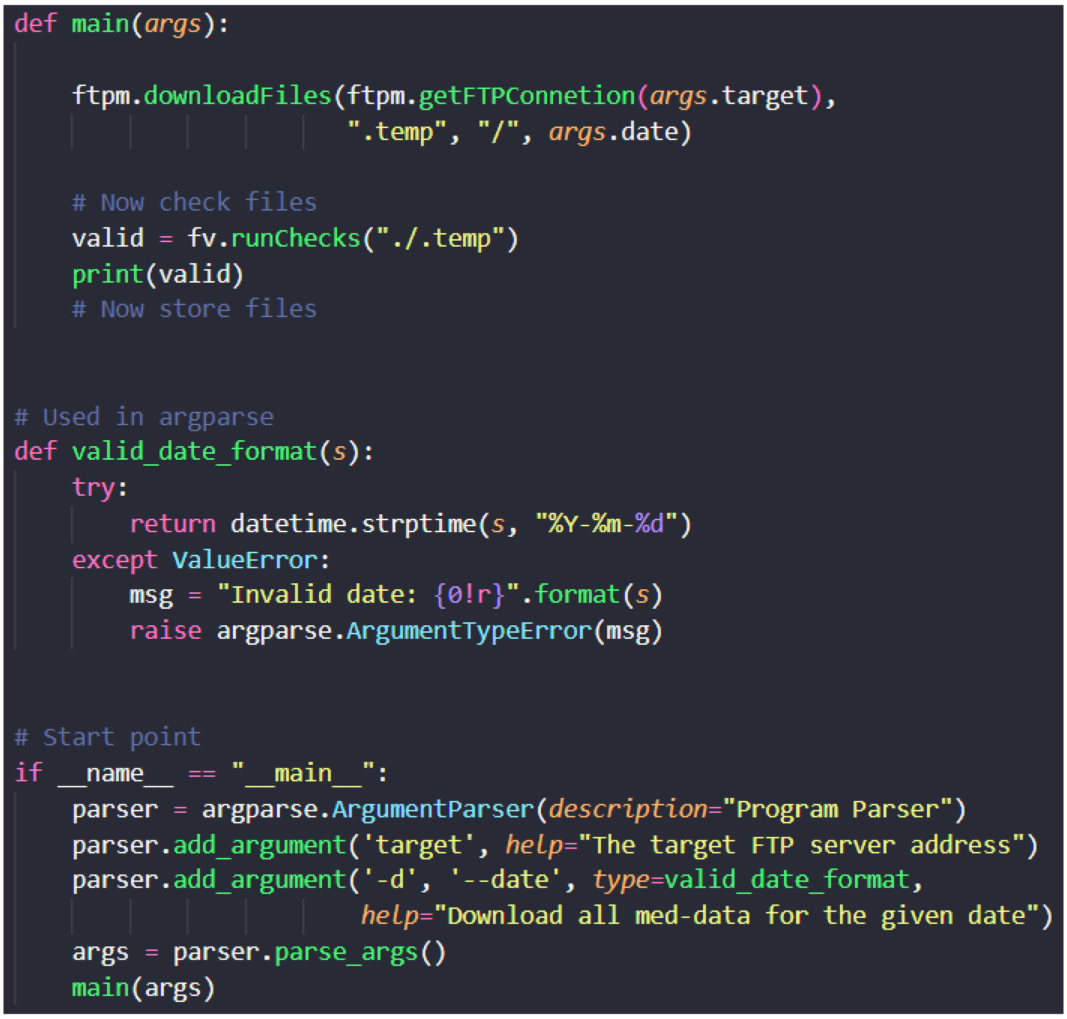
This function allows the user to connect to the target server or throws an error if it is unable to (Figure 3).



```python
def getFTPConnetion(target):
    ftp = None
    try:
        print("Connecting to FTP")
        ftp = FTP(target)
        ftp.login()
        print("Connected to FTP")

    except Exception as e:
        print("Failed to connect to the FTP server. Aborting!\n")
        exit()

    return ftp
```

FIGURE 3

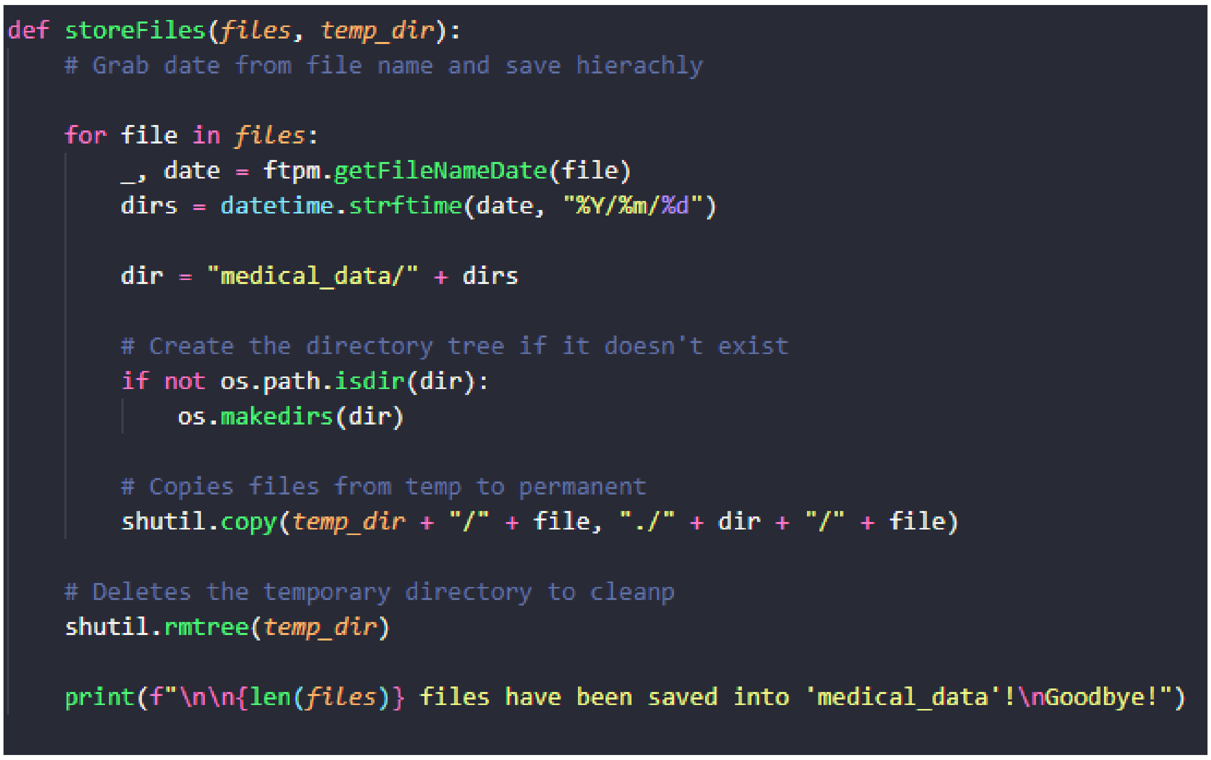
These functions are the core of the program. The user can use the command line to specify the server to be connected to, and the date of data to be fetched. It also validates the date format using validate_date_format() (Figure 4).



```python
def main(args):

    ftpm.downloadFiles(ftpm.getFTPConnetion(args.target),
                       ".temp", "/", args.date)

    # Now check files
    valid = fv.runChecks("./.temp")
    print(valid)
    # Now store files


# Used in argparse
def valid_date_format(s):
    try:
        return datetime.strptime(s, "%Y-%m-%d")
    except ValueError:
        msg = "Invalid date: {0!r}".format(s)
        raise argparse.ArgumentTypeError(msg)


# Start point
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Program Parser")
    parser.add_argument('target', help="The target FTP server address")
    parser.add_argument('-d', '--date', type=valid_date_format,
                        help="Download all med-data for the given date")
    args = parser.parse_args()
    main(args)
```

FIGURE 4

The user can also save the files from the server (Figure 5).



```
def storeFiles(files, temp_dir):
    # Grab date from file name and save hierachly

    for file in files:
        _, date = ftpm.getFileNameDate(file)
        dirs = datetime.strftime(date, "%Y/%m/%d")

        dir = "medical_data/" + dirs

        # Create the directory tree if it doesn't exist
        if not os.path.isdir(dir):
            os.makedirs(dir)

        # Copies files from temp to permanent
        shutil.copy(temp_dir + "/" + file, "./" + dir + "/" + file)

    # Deletes the temporary directory to cleanp
    shutil.rmtree(temp_dir)

    print(f"\n\n{len(files)} files have been saved into 'medical_data'!\nGoodbye!")
```

FIGURE 5

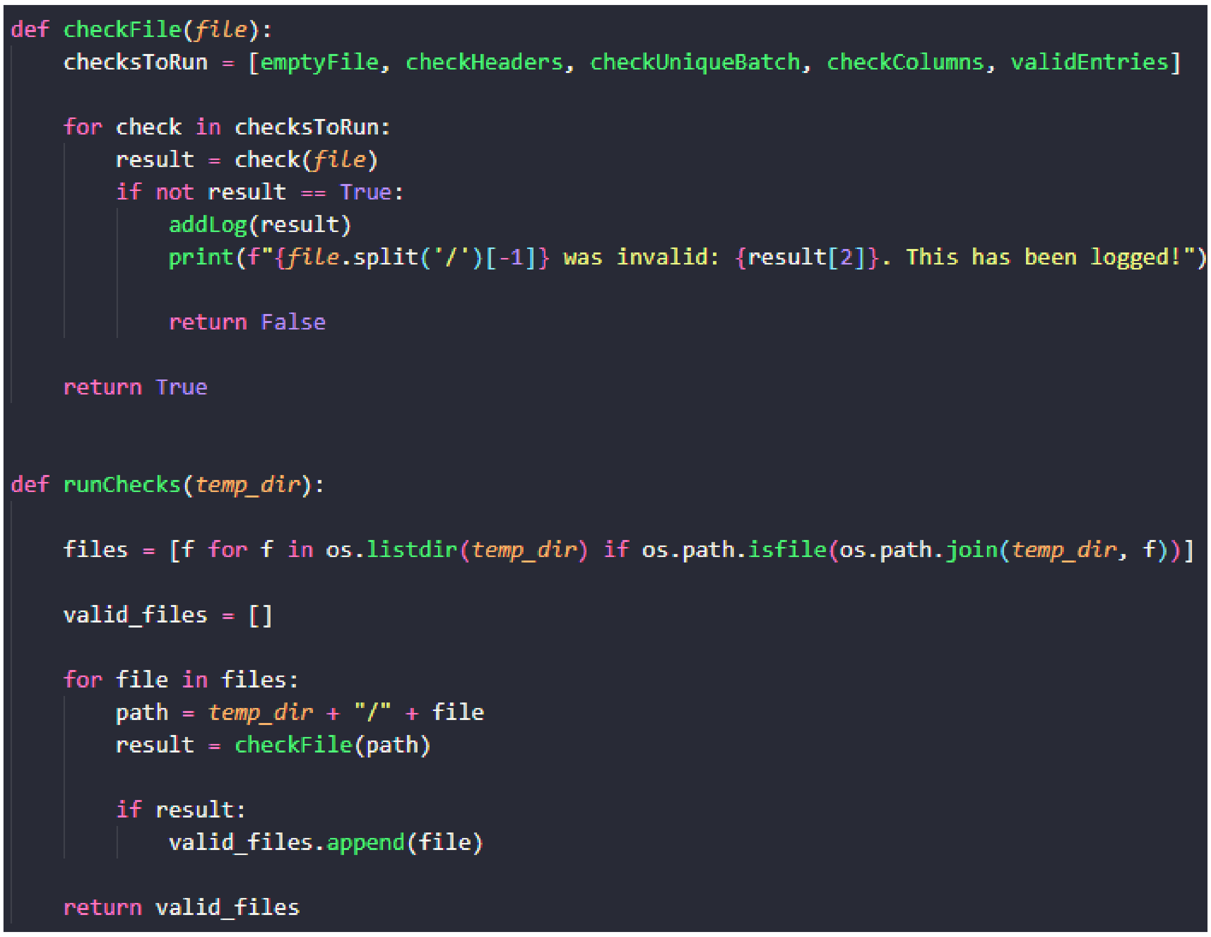
File Validation:

fileValidation.py

Once all the required files have been downloaded, the program ensures that they are all valid, checking against the criteria:

- The file isn't empty
- All the column headings follow standard naming conventions and order
- Each batch in the file has a unique ID
- There are no missing columns
- The data entered is valid (0 < data <= 9.999)

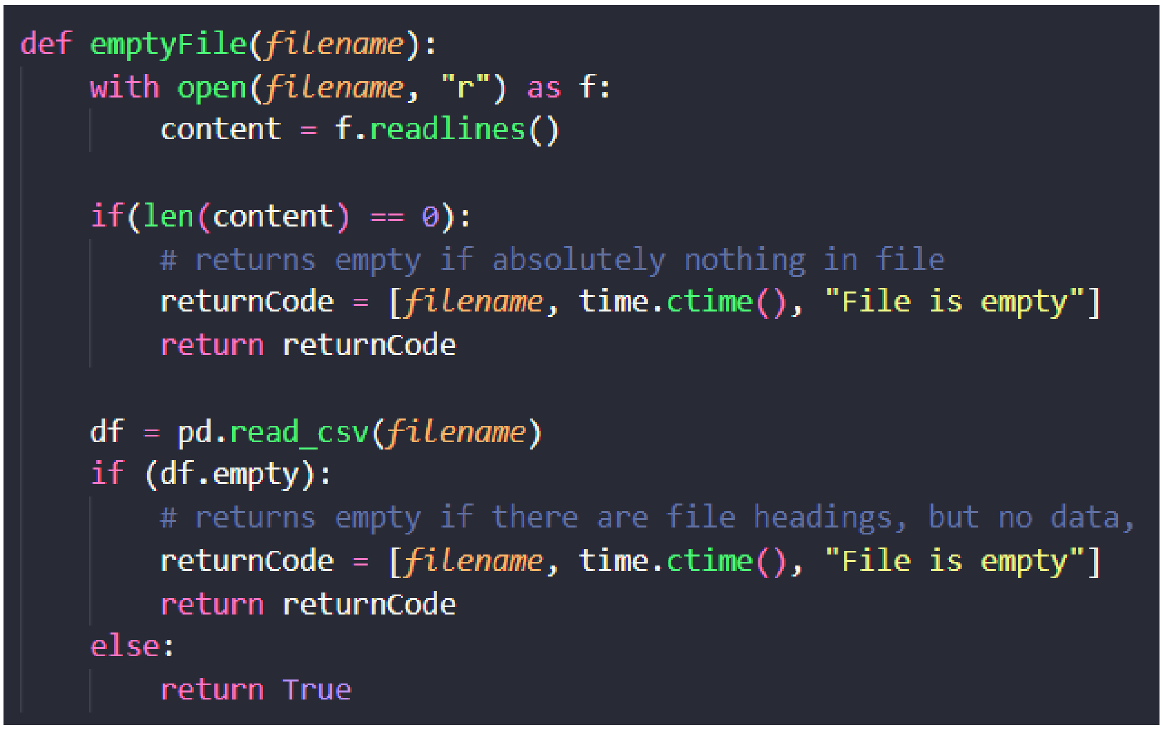
If an error is found, a log is created and added to the log file (Figure 6).



```python
def checkFile(file):
    checksToRun = [emptyFile, checkHeaders, checkUniqueBatch, checkColumns, validEntries]

    for check in checksToRun:
        result = check(file)
        if not result == True:
            addLog(result)
            print(f"{file.split('/')[-1]} was invalid: {result[2]}. This has been logged!")

            return False

    return True


def runChecks(temp_dir):

    files = [f for f in os.listdir(temp_dir) if os.path.isfile(os.path.join(temp_dir, f))]

    valid_files = []

    for file in files:
        path = temp_dir + "/" + file
        result = checkFile(path)

        if result:
            valid_files.append(file)

    return valid_files
```

FIGURE 6

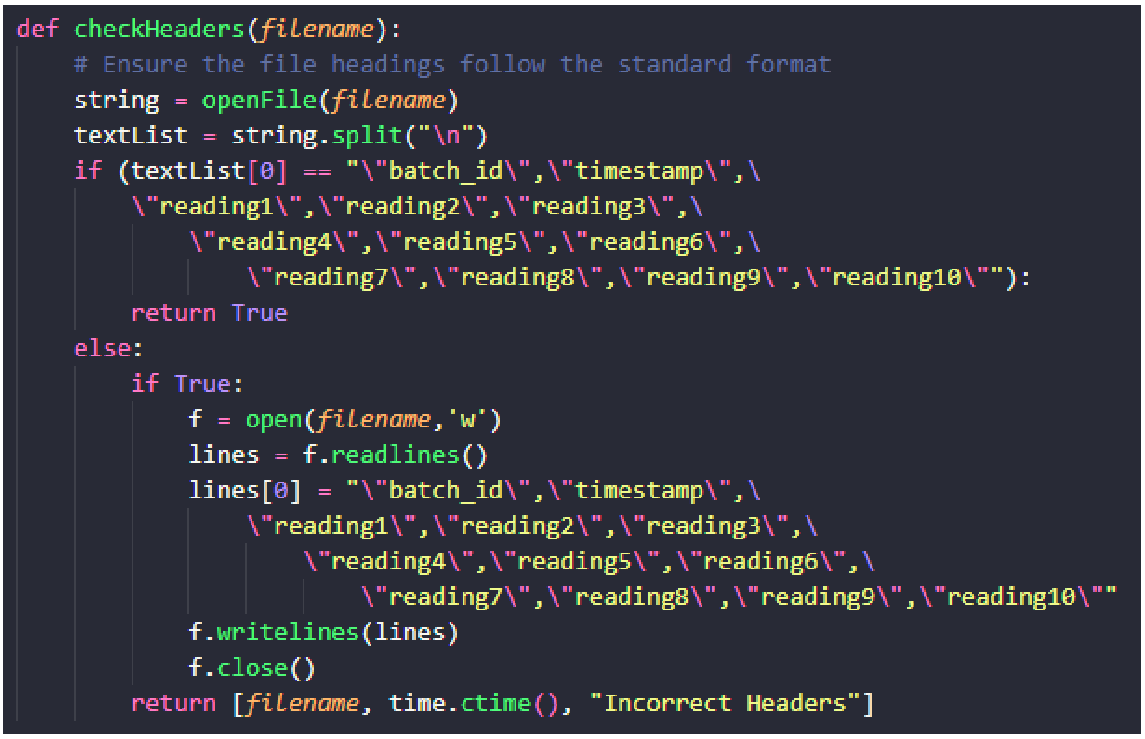
There are two interpretations of an empty file – entirely blank, or with the correct headers but no readings taken. This function checks for both possibilities (Figure 7).



```python
def emptyFile(filename):
    with open(filename, "r") as f:
        content = f.readlines()

    if(len(content) == 0):
        # returns empty if absolutely nothing in file
        returnCode = [filename, time.ctime(), "File is empty"]
        return returnCode

    df = pd.read_csv(filename)
    if (df.empty):
        # returns empty if there are file headings, but no data,
        returnCode = [filename, time.ctime(), "File is empty"]
        return returnCode
    else:
        return True
```

FIGURE 7

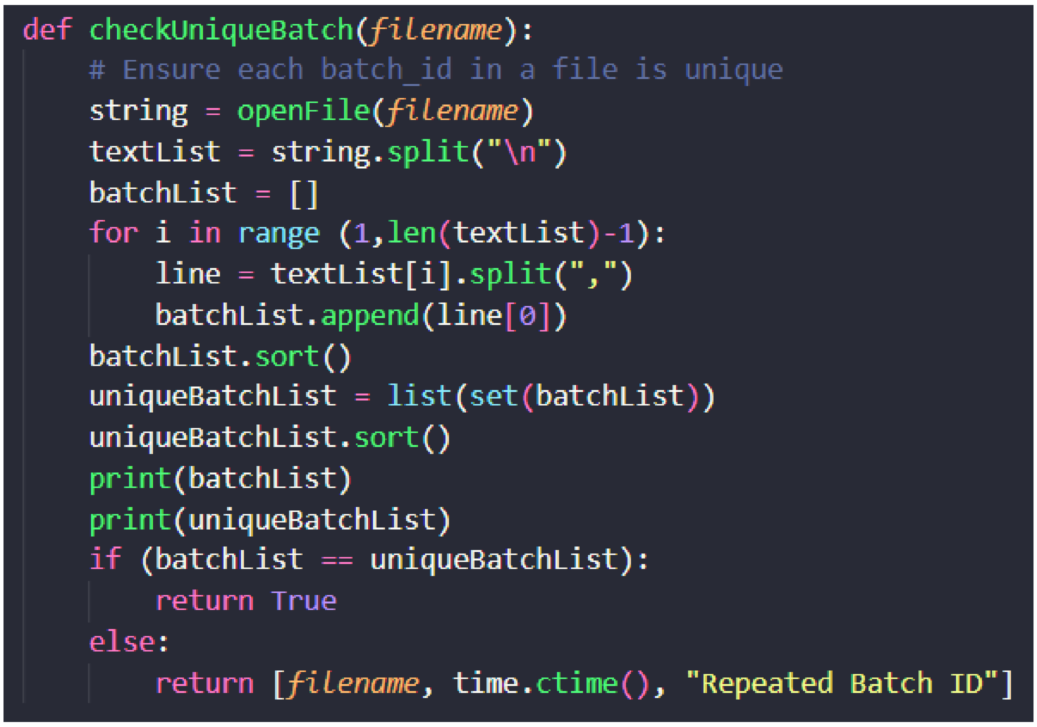
Once we ensure the file has content and is not empty, we make sure that the headings have the correct names in the correct orders (Figure 8).



```python
def checkHeaders(filename):
    # Ensure the file headings follow the standard format
    string = openFile(filename)
    textList = string.split("\n")
    if (textList[0] == "\"batch_id\",\"timestamp\",\
        \"reading1\",\"reading2\",\"reading3\",\
            \"reading4\",\"reading5\",\"reading6\",\
                \"reading7\",\"reading8\",\"reading9\",\"reading10\""):
        return True
    else:
        if True:
            f = open(filename,'w')
            lines = f.readlines()
            lines[0] = "\"batch_id\",\"timestamp\",\
                \"reading1\",\"reading2\",\"reading3\",\
                    \"reading4\",\"reading5\",\"reading6\",\
                        \"reading7\",\"reading8\",\"reading9\",\"reading10\""
            f.writelines(lines)
            f.close()
        return [filename, time.ctime(), "Incorrect Headers"]
```

FIGURE 8

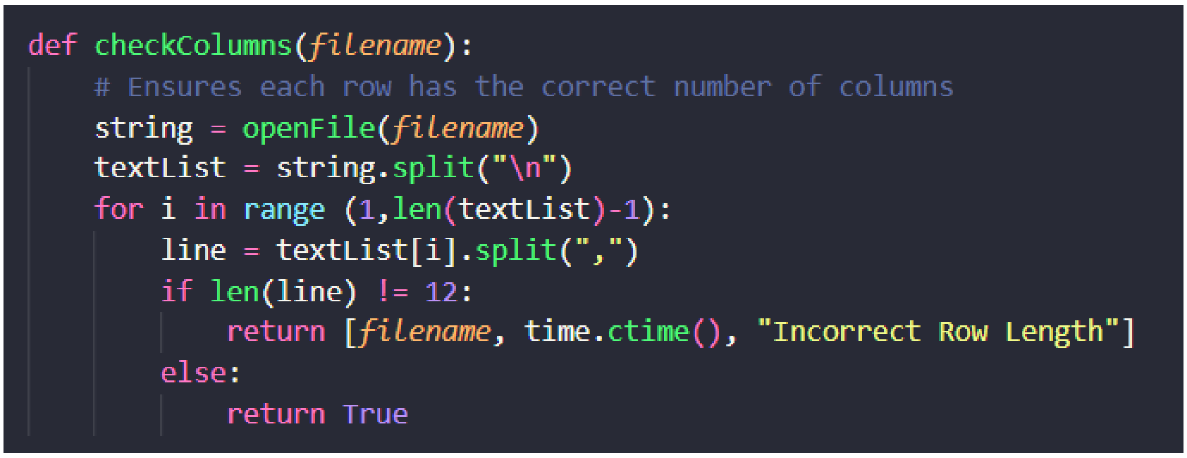
Each file consists of multiple batches. We validate that there are no duplicated batch numbers in the file (Figure 9).



```python
def checkUniqueBatch(filename):
    # Ensure each batch_id in a file is unique
    string = openFile(filename)
    textList = string.split("\n")
    batchList = []
    for i in range (1,len(textList)-1):
        line = textList[i].split(",")
        batchList.append(line[0])
    batchList.sort()
    uniqueBatchList = list(set(batchList))
    uniqueBatchList.sort()
    print(batchList)
    print(uniqueBatchList)
    if (batchList == uniqueBatchList):
        return True
    else:
        return [filename, time.ctime(), "Repeated Batch ID"]
```

FIGURE 9

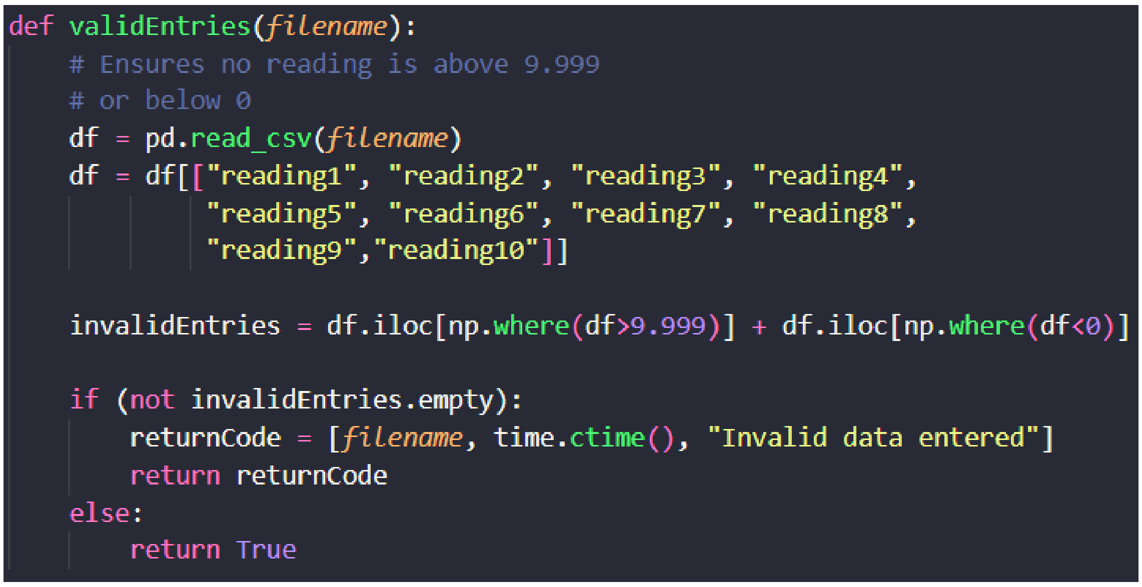
Inside the file, each row must have the same number of columns. This function checks this is correct (Figure 10).



```python
def checkColumns(filename):
    # Ensures each row has the correct number of columns
    string = openFile(filename)
    textList = string.split("\n")
    for i in range (1,len(textList)-1):
        line = textList[i].split(",")
        if len(line) != 12:
            return [filename, time.ctime(), "Incorrect Row Length"]
        else:
            return True
```

FIGURE 10

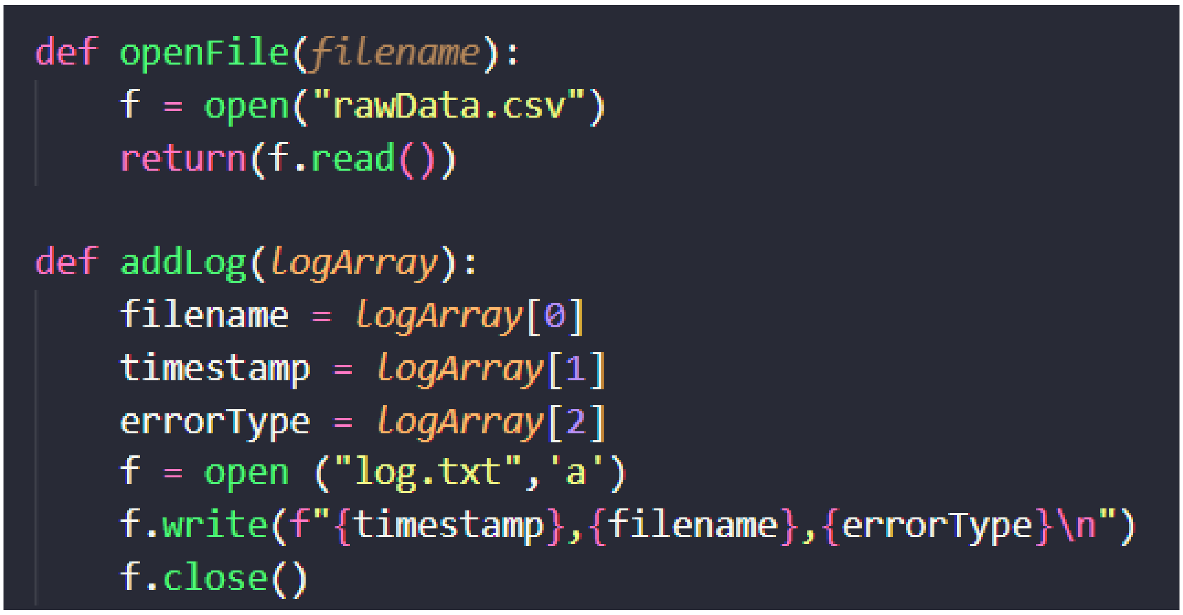
Finally, we ensure each item of data in the file is valid within the correct range (Figure 11).



```python
def validEntries(filename):
    # Ensures no reading is above 9.999
    # or below 0
    df = pd.read_csv(filename)
    df = df[["reading1", "reading2", "reading3", "reading4",
            "reading5", "reading6", "reading7", "reading8",
            "reading9","reading10"]]

    invalidEntries = df.iloc[np.where(df>9.999)] + df.iloc[np.where(df<0)]

    if (not invalidEntries.empty):
        returnCode = [filename, time.ctime(), "Invalid data entered"]
        return returnCode
    else:
        return True
```

FIGURE 11

These functions are open the file and create the log file to store when any problems arise (Figure 12).



```python
def openFile(filename):
    f = open("rawData.csv")
    return(f.read())

def addLog(logArray):
    filename = logArray[0]
    timestamp = logArray[1]
    errorType = logArray[2]
    f = open ("log.txt",'a')
    f.write(f"{timestamp},{filename},{errorType}\n")
    f.close()
```

FIGURE 12

What have we learnt:

This has been a new experience for us - working within a group with a big range in our individual skillsets. We had a variety of levels of programming, teamwork and project management skills however we managed to work well together so that everyone found a role benefiting their strengths.

It was also interesting using GitHub to manage the project. It took some time to learn when to push and pull for people not so used to version control systems, but by the end we had a good flow working together.

Trying to divide the project up into separate sections for each of us presented an interesting problem. We had to make sure that everyone had an equal share of work while also everyone played to their strengths. After having a group meeting we understood these factors and came to a conclusion of what role each person would be best suited to undertake.

On reflection if we repeated a project like this there would be a few things we would try to improve for next time. Firstly, we would meet more often to make sure that everyone is on schedule and working towards the same idea. Secondly, we would set better conventions and standards before starting the project. For example, before starting we did not choose a consistent variable name convention so some variable names are CamelCase whereas others are snake_case.

Overall, we really enjoyed this project and have learnt lots of useful new skills for our futures.