# !/usr/bin/python

# -*- coding: utf-8 -*-

import os import sys import time from os.path import join as pathjoin import debug.log as debug from urllib import unquote

from flask import Flask, redirect, render\_template, request, session, url\_for, send\_file

import helpers.constants as constants import helpers.general\_functions as general\_functions import managers.session\_manager as session\_manager from managers import utility

# ————

import managers.utility

app = Flask(**name**) app.config[‘MAX\_CONTENT\_LENGTH’] = constants.MAX\_FILE\_SIZE # convert into byte

@app.route(“/”, methods=[“GET”]) # Tells Flask to load this function when someone is at ‘/’ def base(): “”" Page behavior for the base url (‘/’) of the site. Handles redirection to other pages. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”"

return redirect(url\_for('upload'))

@app.route(“/downloadworkspace”, methods=[“GET”]) # Tells Flask to load this function when someone is at ‘/downloadworkspace’ def downloadworkspace(): “”" Downloads workspace that stores all the session contents, which can be uploaded and restore all the workspace. “”" fileManager = managers.utility.loadFileManager() path = fileManager.zipWorkSpace()

return send\_file(path, attachment\_filename=constants.WORKSPACE\_FILENAME, as\_attachment=True)

@app.route(“/reset”, methods=[“GET”]) # Tells Flask to load this function when someone is at ‘/reset’ def reset(): “”" Resets the session and initializes a new one every time the reset URL is used (either manually or via the “Reset” button) Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" session\_manager.reset() # Reset the session and session folder session\_manager.init() # Initialize the new session

return redirect(url\_for('upload'))

@app.route(“/upload”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/upload’ def upload(): “”" Handles the functionality of the upload page. It uploads files to be used in the current session. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”"

if request.method == "GET":  
  
 session\_manager.fix() # fix the session in case the browser is caching the old session  
  
 return render\_template('upload.html', MAX\_FILE\_SIZE=constants.MAX\_FILE\_SIZE,  
 MAX\_FILE\_SIZE\_INT=constants.MAX\_FILE\_SIZE\_INT,  
 MAX\_FILE\_SIZE\_UNITS=constants.MAX\_FILE\_SIZE\_UNITS)  
  
if 'X\_FILENAME' in request.headers: # X\_FILENAME is the flag to signify a file upload  
 # File upload through javascript  
 fileManager = managers.utility.loadFileManager()  
  
 # --- check file name ---  
 fileName = request.headers[  
 'X\_FILENAME'] # Grab the filename, which will be UTF-8 percent-encoded (e.g. '%E7' instead of python's '\xe7')  
 if isinstance(fileName, unicode): # If the filename comes through as unicode  
 fileName = fileName.encode('ascii') # Convert to an ascii string  
 fileName = unquote(fileName).decode(  
 'utf-8') # Unquote using urllib's percent-encoding decoder (turns '%E7' into '\xe7'), then deocde it  
 # --- end check file name ---  
  
 if fileName.endswith('.lexos'):  
 fileManager.handleUploadWorkSpace()  
  
 # update filemanager  
 fileManager = managers.utility.loadFileManager()  
 fileManager.updateWorkspace()  
  
 else:  
 fileManager.addUploadFile(request.data, fileName)  
  
 managers.utility.saveFileManager(fileManager)  
 return 'success'

@app.route(“/select\_old”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/select\_old’ def select\_old(): “”" Handles the functionality of the select page. Its primary role is to activate/deactivate specific files depending on the user’s input. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager() # Usual loading of the FileManager

if request.method == "GET":  
 activePreviews = fileManager.getPreviewsOfActive()  
 inactivePreviews = fileManager.getPreviewsOfInactive()  
  
 return render\_template('select\_old.html', activeFiles=activePreviews, inactiveFiles=inactivePreviews)  
  
if 'toggleFile' in request.headers:  
 # Catch-all for any POST request.  
 # On the select page, POSTs come from JavaScript AJAX XHRequests.  
 fileID = int(request.data)  
  
 fileManager.toggleFile(fileID) # Toggle the file from active to inactive or vice versa  
  
elif 'setLabel' in request.headers:  
 newLabel = (request.headers['setLabel']).decode('utf-8')  
 fileID = int(request.data)  
  
 fileManager.files[fileID].label = newLabel  
  
elif 'disableAll' in request.headers:  
 fileManager.disableAll()  
  
elif 'selectAll' in request.headers:  
 fileManager.enableAll()  
  
elif 'applyClassLabel' in request.headers:  
 fileManager.classifyActiveFiles()  
  
elif 'deleteActive' in request.headers:  
 fileManager.deleteActiveFiles()  
  
managers.utility.saveFileManager(fileManager)  
  
return '' # Return an empty string because you have to return something

@app.route(“/removeUploadLabels”, methods=[“GET”, “POST”]) # Tells Flask to handle ajax request from ‘/scrub’ def removeUploadLabels(): “”" Removes Scrub upload files from the session when the labels are clicked. “”" option = request.headers[“option”] session[‘scrubbingoptions’][‘optuploadnames’][option] = ’’ return “success”

@app.route(“/scrub”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/scrub’ def scrub(): “”" Handles the functionality of the scrub page. It scrubs the files depending on the specifications chosen by the user, with an option to download the scrubbed files. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager()

if request.method == "GET":  
 # "GET" request occurs when the page is first loaded.  
 if 'scrubbingoptions' not in session:  
 session['scrubbingoptions'] = constants.DEFAULT\_SCRUB\_OPTIONS  
  
 previews = fileManager.getPreviewsOfActive()  
 tagsPresent, DOEPresent = fileManager.checkActivesTags()  
  
 return render\_template('scrub.html', previews=previews, haveTags=tagsPresent, haveDOE=DOEPresent)  
  
# if 'preview' in request.form or 'apply' in request.form:  
# # The 'Preview Scrubbing' or 'Apply Scrubbing' button is clicked on scrub.html.  
# session\_manager.cacheAlterationFiles()  
# session\_manager.cacheScrubOptions()  
  
# # saves changes only if 'Apply Scrubbing' button is clicked  
# savingChanges = True if 'apply' in request.form else False  
  
# previews = fileManager.scrubFiles(savingChanges=savingChanges)  
# tagsPresent, DOEPresent = fileManager.checkActivesTags()  
  
# if savingChanges:  
# managers.utility.saveFileManager(fileManager)  
  
# return render\_template('scrub.html', previews=previews, haveTags=tagsPresent, haveDOE=DOEPresent)  
  
# if 'download' in request.form:  
# # The 'Download Scrubbed Files' button is clicked on scrub.html.  
# # sends zipped files to downloads folder.  
# return fileManager.zipActiveFiles('scrubbed.zip')

@app.route(“/cut”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/cut’ def cut(): “”" Handles the functionality of the cut page. It cuts the files into various segments depending on the specifications chosen by the user, and sends the text segments. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager()

active = fileManager.getActiveFiles()  
if len(active) > 0:  
  
 numChar = map(lambda x: x.numLetters(), active)  
 numWord = map(lambda x: x.numWords(), active)  
 numLine = map(lambda x: x.numLines(), active)  
 maxChar = max(numChar)  
 maxWord = max(numWord)  
 maxLine = max(numLine)  
 activeFileIDs = [lfile.id for lfile in active]  
  
else:  
 numChar = []  
 numWord = []  
 numLine = []  
 maxChar = 0  
 maxWord = 0  
 maxLine = 0  
 activeFileIDs =[]  
  
if request.method == "GET":  
 # "GET" request occurs when the page is first loaded.  
 if 'cuttingoptions' not in session:  
 session['cuttingoptions'] = constants.DEFAULT\_CUT\_OPTIONS  
  
 previews = fileManager.getPreviewsOfActive()  
  
  
 return render\_template('cut.html', previews=previews, num\_active\_files=len(previews), numChar=numChar, numWord=numWord, numLine=numLine, maxChar=maxChar, maxWord=maxWord, maxLine=maxLine, activeFileIDs = activeFileIDs)  
  
if 'preview' in request.form or 'apply' in request.form:  
  
 # The 'Preview Cuts' or 'Apply Cuts' button is clicked on cut.html.  
 session\_manager.cacheCuttingOptions()  
  
 savingChanges = True if 'apply' in request.form else False # Saving changes only if apply in request form  
 previews = fileManager.cutFiles(savingChanges=savingChanges)  
  
 if savingChanges:  
 managers.utility.saveFileManager(fileManager)  
 active = fileManager.getActiveFiles()  
 numChar = map(lambda x: x.numLetters(), active)  
 numWord = map(lambda x: x.numWords(), active)  
 numLine = map(lambda x: x.numLines(), active)  
 maxChar = max(numChar)  
 maxWord = max(numWord)  
 maxLine = max(numLine)  
 activeFileIDs = [lfile.id for lfile in active]  
  
 return render\_template('cut.html', previews=previews, num\_active\_files=len(previews), numChar=numChar, numWord=numWord, numLine=numLine, maxChar=maxChar, maxWord=maxWord, maxLine=maxLine, activeFileIDs = activeFileIDs)  
  
if 'downloadchunks' in request.form:  
 # The 'Download Segmented Files' button is clicked on cut.html  
 # sends zipped files to downloads folder  
 return fileManager.zipActiveFiles('cut\_files.zip')

@app.route(“/tokenizer”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/tokenize’ def tokenizer(): fileManager = managers.utility.loadFileManager() labels = fileManager.getActiveLabels() headerLabels = [] for fileID in labels: headerLabels.append(fileManager.files[int(fileID)].label) if ‘analyoption’ not in session: session[‘analyoption’] = constants.DEFAULT\_ANALYZE\_OPTIONS if ‘csvoptions’ not in session: session[‘csvoptions’] = constants.DEFAULT\_CSV\_OPTIONS csvorientation = session[‘csvoptions’][‘csvorientation’] csvdelimiter = session[‘csvoptions’][‘csvdelimiter’] cullnumber = session[‘analyoption’][‘cullnumber’] tokenType = session[‘analyoption’][‘tokenType’] normalizeType = session[‘analyoption’][‘normalizeType’] tokenSize = session[‘analyoption’][‘tokenSize’] norm = session[‘analyoption’][‘norm’] #csvdata = session[‘csvoptions’][‘csvdata’] print(“Session”) print(str(session[‘csvoptions’])) # Give the dtm matrix functions some default options data = {‘cullnumber’: cullnumber, ‘tokenType’: tokenType, ‘normalizeType’: normalizeType, ‘csvdelimiter’: csvdelimiter, ‘mfwnumber’: ‘1’, ‘csvorientation’: csvorientation, ‘tokenSize’: tokenSize, ‘norm’: norm} session\_manager.cacheAnalysisOption() dtm = utility.generateCSVMatrixFromAjax(data, fileManager, roundDecimal=True) del dtm[0] # delete the labels #Convert to json for DataTables matrix = [] for i in dtm: i = [str(j) for j in i] matrix.append(list(i)) numRows = len(matrix) draw = 1 return render\_template(‘tokenizer.html’, labels=labels, headerLabels=headerLabels, matrix=matrix, numRows=numRows, draw=draw)

@app.route(“/testA”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/tokenize’ def testA(): from datetime import datetime startTime = datetime.now() from operator import itemgetter import json form = request.json print(form) data = request.json fileManager = managers.utility.loadFileManager() labels = fileManager.getActiveLabels() headerLabels = [] for fileID in labels: headerLabels.append(fileManager.files[int(fileID)].label) if ‘analyoption’ not in session: session[‘analyoption’] = constants.DEFAULT\_ANALIZE\_OPTIONS if ‘csvoptions’ not in session: session[‘csvoptions’] = constants.DEFAULT\_CSV\_OPTIONS session\_manager.cacheAnalysisOption() dtm = utility.generateCSVMatrixFromAjax(data, fileManager, roundDecimal=True) del dtm[0] # delete the labels

# Get query variables  
page = request.json["page"]  
start = request.json["start"]  
end = request.json["end"]  
length = request.json["length"]  
draw = request.json["draw"] + 1  
search = str(request.json["search"])  
sortColumn = request.json["sortColumn"]  
order = request.json["order"]  
if order == "desc":  
 reverse = True  
else:  
 reverse = False  
  
# Sort and Filter the cached DTM by column  
# NB. Sorting needs to be run though a natsort function  
if len(search) != 0:  
 dtmSorted = filter(lambda x: x[0].startswith(search), dtm)  
 numRows = len(dtmSorted)  
 dtmSorted = sorted(dtmSorted,key=itemgetter(sortColumn), reverse=reverse)  
else:  
 dtmSorted = sorted(dtm,key=itemgetter(sortColumn), reverse=reverse)  
  
# Get the number of filtered rows  
numFilteredRows = len(dtmSorted)  
  
#Convert to json for DataTables  
matrix = []  
for i in dtmSorted:  
 i = [str(j) for j in i]  
 matrix.append(list(i))  
numRows = len(matrix)  
if int(data["length"]) == -1:  
 matrix = matrix[0:]  
else:   
 start = int(data["start"])  
 end = int(data["end"])  
 matrix = matrix[start:end]  
response = {"draw": draw, "recordsTotal": numRows, "recordsFiltered": numFilteredRows, "length": int(data["length"]), "headerLabels": headerLabels, "data": matrix}  
print("Script complete")  
print datetime.now() - startTime  
return json.dumps(response)

@app.route(“/tokenizer-old”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/tokenize’ def tokenizerOld(): “”" Handles the functionality on the tokenizer page. It analyzes the texts to produce and send various frequency matrices. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager()

if request.method == "GET":  
 if 'analyoption' not in session:  
 session['analyoption'] = constants.DEFAULT\_ANALYZE\_OPTIONS  
 if 'csvoptions' not in session:  
 session['csvoptions'] = constants.DEFAULT\_CSV\_OPTIONS  
 # "GET" request occurs when the page is first loaded.  
 labels = fileManager.getActiveLabels()  
 return render\_template('tokenizer.html', labels=labels, matrixExist=False)  
  
if 'gen-csv' in request.form:  
 # The 'Generate and Visualize Matrix' button is clicked on tokenizer.html.  
 session\_manager.cacheAnalysisOption()  
 session\_manager.cacheCSVOptions()  
 labels = fileManager.getActiveLabels()  
  
 matrixTitle, tableStr = utility.generateTokenizeResults(fileManager)  
 managers.utility.saveFileManager(fileManager)  
  
 return render\_template('tokenizer.html', labels=labels, matrixTitle=matrixTitle,  
 tableStr=tableStr, matrixExist=True)  
  
if 'get-csv' in request.form:  
 # The 'Download Matrix' button is clicked on tokenizer.html.  
 session\_manager.cacheAnalysisOption()  
 session\_manager.cacheCSVOptions()  
 savePath, fileExtension = utility.generateCSV(fileManager)  
 managers.utility.saveFileManager(fileManager)  
  
 return send\_file(savePath, attachment\_filename="frequency\_matrix" + fileExtension, as\_attachment=True)

@app.route(“/statistics”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/statsgenerator’ def statistics(): “”" Handles the functionality on the Statistics page … Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager() labels = fileManager.getActiveLabels()

if request.method == "GET":  
 # "GET" request occurs when the page is first loaded.  
 if 'analyoption' not in session:  
 session['analyoption'] = constants.DEFAULT\_ANALYZE\_OPTIONS  
 if 'statisticoption' not in session:  
 session['statisticoption'] = {'segmentlist': map(unicode, fileManager.files.keys())} # default is all on  
  
 return render\_template('statistics.html', labels=labels, labels2=labels)  
  
if request.method == "POST":  
  
 token = request.form['tokenType']  
  
 FileInfoDict, corpusInfoDict = utility.generateStatistics(fileManager)  
  
 session\_manager.cacheAnalysisOption()  
 session\_manager.cacheStatisticOption()  
 # DO NOT save fileManager!  
 return render\_template('statistics.html', labels=labels, FileInfoDict=FileInfoDict,  
 corpusInfoDict=corpusInfoDict, token= token)

@app.route(“/hierarchy”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/hierarchy’ def hierarchy(): “”" Handles the functionality on the hierarchy page. It analyzes the various texts and displays a dendrogram. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager() leq = ‘≤’.decode(‘utf-8’)

if request.method == "GET":  
 # "GET" request occurs when the page is first loaded.  
 if 'analyoption' not in session:  
 session['analyoption'] = constants.DEFAULT\_ANALYZE\_OPTIONS  
 if 'hierarchyoption' not in session:  
 session['hierarchyoption'] = constants.DEFAULT\_HIERARCHICAL\_OPTIONS  
 labels = fileManager.getActiveLabels()  
 thresholdOps = {}  
 return render\_template('hierarchy.html', labels=labels, thresholdOps=thresholdOps)  
  
if 'dendro\_download' in request.form:  
 # The 'Download Dendrogram' button is clicked on hierarchy.html.  
 # sends pdf file to downloads folder.  
 utility.generateDendrogram(fileManager)  
 attachmentname = "den\_" + request.form['title'] + ".pdf" if request.form['title'] != '' else 'dendrogram.pdf'  
 session\_manager.cacheAnalysisOption()  
 session\_manager.cacheHierarchyOption()  
 return send\_file(pathjoin(session\_manager.session\_folder(), constants.RESULTS\_FOLDER + "dendrogram.pdf"),  
 attachment\_filename=attachmentname, as\_attachment=True)  
  
if 'dendroSVG\_download' in request.form:  
 utility.generateDendrogram(fileManager)  
 attachmentname = "den\_" + request.form['title'] + ".svg" if request.form['title'] != '' else 'dendrogram.svg'  
 session\_manager.cacheAnalysisOption()  
 session\_manager.cacheHierarchyOption()  
 return send\_file(pathjoin(session\_manager.session\_folder(), constants.RESULTS\_FOLDER + "dendrogram.svg"),  
 attachment\_filename=attachmentname, as\_attachment=True)  
  
if 'getdendro' in request.form:  
 # The 'Get Dendrogram' button is clicked on hierarchy.html.  
  
 pdfPageNumber, score, inconsistentMax, maxclustMax, distanceMax, distanceMin, monocritMax, monocritMin, threshold = utility.generateDendrogram(  
 fileManager)  
 session['dengenerated'] = True  
 labels = fileManager.getActiveLabels()  
  
 inconsistentOp = "0 " + leq + " t " + leq + " " + str(inconsistentMax)  
 maxclustOp = "2 " + leq + " t " + leq + " " + str(maxclustMax)  
 distanceOp = str(distanceMin) + " " + leq + " t " + leq + " " + str(distanceMax)  
 monocritOp = str(monocritMin) + " " + leq + " t " + leq + " " + str(monocritMax)  
  
 thresholdOps = {"inconsistent": inconsistentOp, "maxclust": maxclustOp, "distance": distanceOp,  
 "monocrit": monocritOp}  
  
 managers.utility.saveFileManager(fileManager)  
 session\_manager.cacheAnalysisOption()  
 session\_manager.cacheHierarchyOption()  
 return render\_template('hierarchy.html', labels=labels, pdfPageNumber=pdfPageNumber, score=score,  
 inconsistentMax=inconsistentMax, maxclustMax=maxclustMax, distanceMax=distanceMax,  
 distanceMin=distanceMin, monocritMax=monocritMax, monocritMin=monocritMin,  
 threshold=threshold, thresholdOps=thresholdOps)

@app.route(“/dendrogramimage”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/dendrogramimage’ def dendrogramimage(): “”" Reads the png image of the dendrogram and displays it on the web browser. \*dendrogramimage() linked to in analysis.html, displaying the dendrogram.png Note: Returns a response object with the dendrogram png to flask and eventually to the browser. “”" # dendrogramimage() is called in analysis.html, displaying the dendrogram.png (if session[‘dengenerated’] != False). imagePath = pathjoin(session\_manager.session\_folder(), constants.RESULTS\_FOLDER, constants.DENDROGRAM\_FILENAME) return send\_file(imagePath)

@app.route(“/kmeans”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/kmeans’ def kmeans(): “”" Handles the functionality on the kmeans page. It analyzes the various texts and displays the class label of the files. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager() labels = fileManager.getActiveLabels() defaultK = int(len(labels) / 2)

if request.method == 'GET':  
 # 'GET' request occurs when the page is first loaded  
 if 'analyoption' not in session:  
 session['analyoption'] = constants.DEFAULT\_ANALYZE\_OPTIONS  
 if 'kmeanoption' not in session:  
 session['kmeanoption'] = constants.DEFAULT\_KMEAN\_OPTIONS  
  
 return render\_template('kmeans.html', labels=labels, silhouettescore='', kmeansIndex=[], fileNameStr='',  
 fileNumber=len(labels), KValue=0, defaultK=defaultK,  
 colorChartStr='', kmeansdatagenerated=False)  
  
if request.method == "POST":  
 # 'POST' request occur when html form is submitted (i.e. 'Get Graphs', 'Download...')  
  
 if request.form['viz'] == 'PCA':  
 kmeansIndex, silhouetteScore, fileNameStr, KValue, colorChartStr = utility.generateKMeansPCA(fileManager)  
  
 session\_manager.cacheAnalysisOption()  
 session\_manager.cacheKmeanOption()  
 managers.utility.saveFileManager(fileManager)  
 return render\_template('kmeans.html', labels=labels, silhouettescore=silhouetteScore,  
 kmeansIndex=kmeansIndex,  
 fileNameStr=fileNameStr, fileNumber=len(labels), KValue=KValue, defaultK=defaultK,  
 colorChartStr=colorChartStr, kmeansdatagenerated=True)  
  
 elif request.form['viz'] == 'Voronoi':  
  
 kmeansIndex, silhouetteScore, fileNameStr, KValue, colorChartStr, finalPointsList, finalCentroidsList, textData, maxVal = utility.generateKMeansVoronoi(  
 fileManager)  
  
 session\_manager.cacheAnalysisOption()  
 session\_manager.cacheKmeanOption()  
 managers.utility.saveFileManager(fileManager)  
 return render\_template('kmeans.html', labels=labels, silhouettescore=silhouetteScore,  
 kmeansIndex=kmeansIndex, fileNameStr=fileNameStr, fileNumber=len(labels),  
 KValue=KValue, defaultK=defaultK, colorChartStr=colorChartStr,  
 finalPointsList=finalPointsList, finalCentroidsList=finalCentroidsList,  
 textData=textData, maxVal=maxVal, kmeansdatagenerated=True)

@app.route(“/kmeansimage”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/kmeansimage’ def kmeansimage(): “”" Reads the png image of the kmeans and displays it on the web browser.

\*kmeansimage() linked to in analysis.html, displaying the kmeansimage.png  
  
Note: Returns a response object with the kmeansimage png to flask and eventually to the browser.  
"""  
# kmeansimage() is called in kmeans.html, displaying the KMEANS\_GRAPH\_FILENAME (if session['kmeansdatagenerated'] != False).  
imagePath = pathjoin(session\_manager.session\_folder(), constants.RESULTS\_FOLDER, constants.KMEANS\_GRAPH\_FILENAME)  
return send\_file(imagePath)

@app.route(“/rollingwindow”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/rollingwindow’ def rollingwindow(): “”" Handles the functionality on the rollingwindow page. It analyzes the various texts using a rolling window of analysis. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager() labels = fileManager.getActiveLabels()

if request.method == "GET":  
 # "GET" request occurs when the page is first loaded.  
 if 'rwoption' not in session:  
 session['rwoption'] = constants.DEFAULT\_ROLLINGWINDOW\_OPTIONS  
  
 # default legendlabels  
 legendLabels = [""]  
  
 return render\_template('rwanalysis.html', labels=labels, legendLabels=legendLabels,  
 rwadatagenerated=False)  
  
if request.method == "POST":  
 # "POST" request occurs when user hits submit (Get Graph) button  
  
 dataPoints, dataList, graphTitle, xAxisLabel, yAxisLabel, legendLabels = utility.generateRWA(fileManager)  
  
 if 'get-RW-plot' in request.form:  
 # The 'Generate and Download Matrix' button is clicked on rollingwindow.html.  
  
 savePath, fileExtension = utility.generateRWmatrixPlot(dataPoints, legendLabels)  
  
 return send\_file(savePath, attachment\_filename="rollingwindow\_matrix" + fileExtension, as\_attachment=True)  
  
 if 'get-RW-data' in request.form:  
 # The 'Generate and Download Matrix' button is clicked on rollingwindow.html.  
  
 savePath, fileExtension = utility.generateRWmatrix(dataList)  
  
 return send\_file(savePath, attachment\_filename="rollingwindow\_matrix" + fileExtension, as\_attachment=True)  
  
 session\_manager.cacheRWAnalysisOption()  
 return render\_template('rwanalysis.html', labels=labels,  
 data=dataPoints,  
 graphTitle=graphTitle,  
 xAxisLabel=xAxisLabel,  
 yAxisLabel=yAxisLabel,  
 legendLabels=legendLabels,  
 rwadatagenerated=True)

@app.route(“/wordcloud”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/wordcloud’ def wordcloud(): “”" Handles the functionality on the visualisation page – a prototype for displaying single word cloud graphs. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager() labels = fileManager.getActiveLabels()

if request.method == "GET":  
 # "GET" request occurs when the page is first loaded.  
 if 'cloudoption' not in session:  
 session['cloudoption'] = constants.DEFAULT\_CLOUD\_OPTIONS  
  
 # there is no wordcloud option so we don't initialize that  
 return render\_template('wordcloud.html', labels=labels)  
  
if request.method == "POST":  
 # "POST" request occur when html form is submitted (i.e. 'Get Dendrogram', 'Download...')  
 JSONObj = utility.generateJSONForD3(fileManager, mergedSet=True)  
  
 # Create a list of column values for the word count table  
 from operator import itemgetter  
  
 terms = sorted(JSONObj["children"], key=itemgetter('size'), reverse=True)  
  
 columnValues = []  
  
 for term in terms:  
 rows = [term["name"], term["size"]]  
 columnValues.append(rows)  
  
 session\_manager.cacheCloudOption()  
 return render\_template('wordcloud.html', labels=labels, JSONObj=JSONObj, columnValues=columnValues)

@app.route(“/multicloud”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/multicloud’ def multicloud(): “”" Handles the functionality on the multicloud pages. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager()

if request.method == 'GET':  
 # 'GET' request occurs when the page is first loaded.  
 if 'cloudoption' not in session:  
 session['cloudoption'] = constants.DEFAULT\_CLOUD\_OPTIONS  
 if 'multicloudoptions' not in session:  
 session['multicloudoptions'] = constants.DEFAULT\_MULTICLOUD\_OPTIONS  
  
 labels = fileManager.getActiveLabels()  
  
 return render\_template('multicloud.html', jsonStr="", labels=labels)  
  
if request.method == "POST":  
 # 'POST' request occur when html form is submitted (i.e. 'Get Graphs', 'Download...')  
 labels = fileManager.getActiveLabels()  
 JSONObj = utility.generateMCJSONObj(fileManager)  
  
 session\_manager.cacheCloudOption()  
 session\_manager.cacheMultiCloudOptions()

# return render\_template(‘multicloud.html’, JSONObj=JSONObj, labels=labels, loading=‘loading’)

return render\_template('multicloud.html', JSONObj=JSONObj, labels=labels)

@app.route(“/viz”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/viz’ def viz(): “”" Handles the functionality on the alternate bubbleViz page with performance improvements. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager()

if request.method == "GET":  
 # "GET" request occurs when the page is first loaded.  
 if 'cloudoption' not in session:  
 session['cloudoption'] = constants.DEFAULT\_CLOUD\_OPTIONS  
 if 'bubblevisoption' not in session:  
 session['bubblevisoption'] = constants.DEFAULT\_BUBBLEVIZ\_OPTIONS  
  
 labels = fileManager.getActiveLabels()  
  
 return render\_template('viz.html', JSONObj="", labels=labels)  
  
if request.method == "POST":  
 # "POST" request occur when html form is submitted (i.e. 'Get Dendrogram', 'Download...')  
 labels = fileManager.getActiveLabels()  
 JSONObj = utility.generateJSONForD3(fileManager, mergedSet=True)  
  
 session\_manager.cacheCloudOption()  
 session\_manager.cacheBubbleVizOption()

# return render\_template(‘viz.html’, JSONObj=JSONObj, labels=labels, loading=‘loading’)

return render\_template('viz.html', JSONObj=JSONObj, labels=labels)

@app.route(“/extension”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/extension’ def extension(): “”" Handles the functionality on the External Tools page – a prototype for displaying possible external analysis options. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" return render\_template(‘extension.html’)

@app.route(“/similarity”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/extension’ def similarity(): “”" Handles the similarity query page functionality. Returns ranked list of files and their cosine similarities to a comparison document. “”"

fileManager = managers.utility.loadFileManager()  
encodedLabels = {}  
labels = fileManager.getActiveLabels()  
for i in labels:  
 encodedLabels[str(i)] = labels[i].encode("utf-8")  
  
if request.method == 'GET':  
 # 'GET' request occurs when the page is first loaded  
 if 'analyoption' not in session:  
 session['analyoption'] = constants.DEFAULT\_ANALYZE\_OPTIONS  
 if 'similarities' not in session:  
 session['similarities'] = constants.DEFAULT\_SIM\_OPTIONS  
  
 return render\_template('similarity.html', labels=labels, encodedLabels=encodedLabels, docsListScore="", docsListName="",  
 similaritiesgenerated=False)  
  
if 'gen-sims'in request.form:  
 # 'POST' request occur when html form is submitted (i.e. 'Get Graphs', 'Download...')  
 docsListScore, docsListName = utility.generateSimilarities(fileManager)  
  
 session\_manager.cacheAnalysisOption()  
 session\_manager.cacheSimOptions()  
 return render\_template('similarity.html', labels=labels, encodedLabels=encodedLabels, docsListScore=docsListScore, docsListName=docsListName,  
 similaritiesgenerated=True)  
if 'get-sims' in request.form:  
 # The 'Download Matrix' button is clicked on similarity.html.  
 session\_manager.cacheAnalysisOption()  
 session\_manager.cacheSimOptions()  
 savePath, fileExtension = utility.generateSimsCSV(fileManager)  
 managers.utility.saveFileManager(fileManager)  
  
 return send\_file(savePath, attachment\_filename="similarity-query" + fileExtension, as\_attachment=True)

@app.route(“/topword”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/topword’ def topword(): “”" Handles the topword page functionality. “”" fileManager = managers.utility.loadFileManager() labels = fileManager.getActiveLabels()

if request.method == 'GET':  
 # 'GET' request occurs when the page is first loaded  
  
 if 'topwordoption' not in session:  
 session['topwordoption'] = constants.DEFAULT\_TOPWORD\_OPTIONS  
 if 'analyoption' not in session:  
 session['analyoption'] = constants.DEFAULT\_ANALYZE\_OPTIONS  
  
 # get the class label and eliminate the id (this is not the unique id in filemanager)  
 ClassdivisionMap = fileManager.getClassDivisionMap()[1:]  
  
 # get number of class  
 try:  
 num\_class = len(ClassdivisionMap[1])  
 except IndexError:  
 num\_class = 0  
  
 return render\_template('topword.html', labels=labels, classmap=ClassdivisionMap,  
 numclass=num\_class, topwordsgenerated='class\_div')  
  
if request.method == "POST":  
 # 'POST' request occur when html form is submitted (i.e. 'Get Graphs', 'Download...')  
  
 if request.form['testInput'] == 'classToPara':  
 header = 'Comparing Class To All The Paragraph Not Within This Class'  
 elif request.form['testInput'] == 'allToPara':  
 header = 'Compare Each Paragraph To The Whole Corpus'  
 elif request.form['testInput'] == 'classToClass':  
 header = 'Compare Class To Each Other Class'  
 else:  
 raise IOError('the value of request.form["testInput"] cannot be understood by the backend')  
  
 result = utility.GenerateZTestTopWord(fileManager) # get the topword test result  
  
 if 'get-topword' in request.form: # download topword  
 path = utility.getTopWordCSV(result,  
 csv\_header=header)  
  
 session\_manager.cacheAnalysisOption()  
 session\_manager.cacheTopwordOptions()  
 return send\_file(path, attachment\_filename=constants.TOPWORD\_CSV\_FILE\_NAME, as\_attachment=True)  
  
 else:  
 # get the number of class  
 num\_class = len(fileManager.getClassDivisionMap()[2])  
  
 # only give the user a preview of the topWord  
 for i in range(len(result)):  
 if len(result[i][1]) > 20:  
 result[i][1] = result[i][1][:20]  
  
 session\_manager.cacheAnalysisOption()  
 session\_manager.cacheTopwordOptions()  
  
 return render\_template('topword.html', result=result, labels=labels, header=header, numclass=num\_class,  
 topwordsgenerated='True', classmap=[])

# =================== Helpful functions ===================

def install\_secret\_key(fileName=‘secret\_key’): “”" Creates an encryption key for a secure session. Args: fileName: A string representing the secret key. Returns: None “”" fileName = os.path.join(app.static\_folder, fileName) try: app.config[‘SECRET\_KEY’] = open(fileName, ‘rb’).read() except IOError: print ‘Error: No secret key. Create it with:’ if not os.path.isdir(os.path.dirname(fileName)): print ‘mkdir -p’, os.path.dirname(fileName) print ‘head -c 24 /dev/urandom >’, fileName sys.exit(1)

# ================ End of Helpful functions ===============

# =========== Temporary development functions =============

@app.route(“/manage”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/select’ def manage(): “”" Handles the functionality of the select page. Its primary role is to activate/deactivate specific files depending on the user’s input. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager() # Usual loading of the FileManager

if request.method == "GET":  
  
 rows = fileManager.getPreviewsOfAll()  
 for row in rows:  
 if row["state"] == True:  
 row["state"] = "selected"  
 else:  
 row["state"] = ""  
  
 return render\_template('manage.html', rows=rows, itm="best-practices")  
  
if 'previewTest' in request.headers:  
 fileID = int(request.data)  
 fileLabel = fileManager.files[fileID].label  
 filePreview = fileManager.files[fileID].getPreview()  
 previewVals = {"id": fileID, "label": fileLabel, "previewText": filePreview}  
 import json  
  
 return json.dumps(previewVals)  
  
if 'toggleFile' in request.headers:  
 # Catch-all for any POST request.  
 # On the select page, POSTs come from JavaScript AJAX XHRequests.  
 fileID = int(request.data)  
  
 fileManager.toggleFile(fileID) # Toggle the file from active to inactive or vice versa  
  
elif 'toggliFy' in request.headers:  
 fileIDs = request.data  
 fileIDs = fileIDs.split(",")  
 fileManager.disableAll()  
  
 fileManager.togglify(fileIDs) # Toggle the file from active to inactive or vice versa  
  
elif 'setLabel' in request.headers:  
 newName = (request.headers['setLabel']).decode('utf-8')  
 fileID = int(request.data)  
  
 fileManager.files[fileID].setName(newName)  
 fileManager.files[fileID].label = newName  
  
elif 'setClass' in request.headers:  
 newClassLabel = (request.headers['setClass']).decode('utf-8')  
 fileID = int(request.data)  
 fileManager.files[fileID].setClassLabel(newClassLabel)  
  
elif 'disableAll' in request.headers:  
 fileManager.disableAll()  
  
elif 'selectAll' in request.headers:  
 fileManager.enableAll()  
  
elif 'applyClassLabel' in request.headers:  
 fileManager.classifyActiveFiles()  
  
elif 'deleteActive' in request.headers:  
 fileManager.deleteActiveFiles()  
  
elif 'deleteRow' in request.headers:  
 fileManager.deleteFiles(request.form.keys()) # delete the file in request.form  
  
managers.utility.saveFileManager(fileManager)  
return '' # Return an empty string because you have to return something

@app.route(“/selectAll”, methods=[“GET”, “POST”]) def selectAll(): fileManager = managers.utility.loadFileManager() fileManager.enableAll() managers.utility.saveFileManager(fileManager) return ‘success’

@app.route(“/deselectAll”, methods=[“GET”, “POST”]) def deselectAll(): fileManager = managers.utility.loadFileManager() fileManager.disableAll() managers.utility.saveFileManager(fileManager) return ‘success’

@app.route(“/enableRows”, methods=[“GET”, “POST”]) def enableRows(): fileManager = managers.utility.loadFileManager() for fileID in request.json: fileManager.enableFiles(fileID) managers.utility.saveFileManager(fileManager) return ‘success’

@app.route(“/disableRows”, methods=[“GET”, “POST”]) def disableRows(): fileManager = managers.utility.loadFileManager() for fileID in request.json: fileManager.disableFiles(fileID) managers.utility.saveFileManager(fileManager) return ‘success’

@app.route(“/getPreview”, methods=[“GET”, “POST”]) def getPreviews(): fileManager = managers.utility.loadFileManager() fileID = int(request.data) fileLabel = fileManager.files[fileID].label filePreview = fileManager.files[fileID].loadContents() previewVals = {“id”: fileID, “label”: fileLabel, “previewText”: filePreview} import json return json.dumps(previewVals)

@app.route(“/setLabel”, methods=[“GET”, “POST”]) def setLabel(): fileManager = managers.utility.loadFileManager() fileID = int(request.json[0]) newName = request.json[1].decode(‘utf-8’) fileManager.files[fileID].setName(newName) fileManager.files[fileID].label = newName managers.utility.saveFileManager(fileManager) return ‘success’

@app.route(“/setClass”, methods=[“GET”, “POST”]) def setClass(): fileManager = managers.utility.loadFileManager() fileID = int(request.json[0]) newClassLabel = request.json[1].decode(‘utf-8’) fileManager.files[fileID].setClassLabel(newClassLabel) managers.utility.saveFileManager(fileManager) return ‘success’

@app.route(“/deleteOne”, methods=[“GET”, “POST”]) def deleteOne(): fileManager = managers.utility.loadFileManager() fileManager.deleteFiles(request.data) managers.utility.saveFileManager(fileManager) return ‘success’

@app.route(“/deleteSelected”, methods=[“GET”, “POST”]) def deleteSelected(): fileManager = managers.utility.loadFileManager() fileManager.deleteActiveFiles() managers.utility.saveFileManager(fileManager) return ‘success’

@app.route(“/setClassSelected”, methods=[“GET”, “POST”]) def setClassSelected(): fileManager = managers.utility.loadFileManager() rows = request.json[0] newClassLabel = request.json[1].decode(‘utf-8’) for fileID in list(rows): fileManager.files[int(fileID)].setClassLabel(newClassLabel) managers.utility.saveFileManager(fileManager) return ‘success’

@app.route(“/manage-old”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/manage’ def manageOld(): “”" Handles the functionality of the manage page. Its primary role is to activate/deactivate specific documents depending on the user’s input. Note: Returns a response object (often a render\_template call) to flask and eventually to the browser. “”" fileManager = managers.utility.loadFileManager() # Usual loading of the FileManager

if request.method == "GET":  
  
 rows = fileManager.getPreviewsOfAll()  
 for row in rows:  
 if row["state"] == True:  
 row["state"] = "selected"  
 else:  
 row["state"] = ""  
  
 return render\_template('manage.html', rows=rows, itm="best-practices")  
  
if 'previewTest' in request.headers:  
 fileID = int(request.data)  
 fileLabel = fileManager.files[fileID].label  
 filePreview = fileManager.files[fileID].getPreview()  
 previewVals = {"id": fileID, "label": fileLabel, "previewText": filePreview}  
 import json  
  
 return json.dumps(previewVals)  
  
if 'toggleFile' in request.headers:  
 # Catch-all for any POST request.  
 # On the select page, POSTs come from JavaScript AJAX XHRequests.  
 fileID = int(request.data)  
  
 fileManager.toggleFile(fileID) # Toggle the file from active to inactive or vice versa  
  
elif 'toggliFy' in request.headers:  
 fileIDs = request.data  
 fileIDs = fileIDs.split(",")  
 fileManager.disableAll()  
  
 fileManager.togglify(fileIDs) # Toggle the file from active to inactive or vice versa  
  
elif 'setLabel' in request.headers:  
 newName = (request.headers['setLabel']).decode('utf-8')  
 fileID = int(request.data)  
  
 fileManager.files[fileID].setName(newName)  
 fileManager.files[fileID].label = newName  
  
elif 'setClass' in request.headers:  
 newClassLabel = (request.headers['setClass']).decode('utf-8')  
 fileID = int(request.data)  
 fileManager.files[fileID].setClassLabel(newClassLabel)  
  
elif 'disableAll' in request.headers:  
 fileManager.disableAll()  
  
elif 'selectAll' in request.headers:  
 fileManager.enableAll()  
  
elif 'applyClassLabel' in request.headers:  
 fileManager.classifyActiveFiles()  
  
elif 'deleteActive' in request.headers:  
 fileManager.deleteActiveFiles()  
  
elif 'deleteRow' in request.headers:  
 fileManager.deleteFiles(request.form.keys()) # delete the file in request.form  
  
managers.utility.saveFileManager(fileManager)  
return '' # Return an empty string because you have to return something

@app.route(“/gutenberg”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/module’ def gutenberg(): “”" Generic module for saving text stored as a variable to the file manager. It mostly just illustrates how to access the file manager. “”" fileManager = managers.utility.loadFileManager()

if request.method == "GET":  
 # "GET" request occurs when the page is first loaded.  
  
 # Get a dictionary of the currently active files' labels.  
 labels = fileManager.getActiveLabels()  
  
 message = "Submit to load file"  
  
 return render\_template('gutenberg.html', message=message)  
  
if request.method == "POST":  
 # "POST" request occur when html form is submitted  
 labels = fileManager.getActiveLabels()  
  
 # Get the request variable  
 s = request.form["urls"]  
 formLines = [l for l in s.split("\n") if l]  
  
 #import os, urllib # imported by lexos.py  
 import re, urllib  
  
 remove = ["Produced by","End of the Project Gutenberg","End of Project Gutenberg"]  
 savedFiles = "<ol>"  
  
 ''' Reads a raw Project Gutenberg etext, reformat paragraphs,  
 and removes fluff. Determines the title of the book and uses it  
 as a filename to write the resulting output text. '''  
 for url in formLines:  
 f = urllib.urlopen(url)  
 data = f.readlines()  
 f.close()  
 lines = [line.strip() for line in data]  
 collect = False  
 lookforsubtitle = False  
 outlines = []  
 startseen = endseen = False  
 authorLastName = ""  
 title=""  
 one="<?xml version=\"1.0\" encoding=\"utf-8\"?><TEI xmlns=\"http://www.tei-c.org/ns/1.0\" version=\"5.0\"><teiHeader><fileDesc><titleStmt>"  
 two = "</titleStmt><publicationStmt><publisher></publisher><pubPlace></pubPlace><availability status=\"free\"><p>Project Gutenberg</p></availability></publicationStmt><seriesStmt><title>Project Gutenberg Full-Text Database</title></seriesStmt><sourceDesc default=\"false\"><biblFull default=\"false\"><titleStmt>"  
 three = "</titleStmt><extent></extent><publicationStmt><publisher></publisher><pubPlace></pubPlace><date></date></publicationStmt></biblFull></sourceDesc></fileDesc><encodingDesc><editorialDecl default=\"false\"><p>Preliminaries omitted.</p></editorialDecl></encodingDesc></teiHeader><text><body><div>"  
 for line in lines:  
 if line.startswith("Author: "):  
 author = line[8:]  
 authorLastName = author  
 authorTemp = line[8:]  
 continue  
 if line.startswith("Title: "):  
 title = line[7:]  
 titleTemp = line[7:]  
 lookforsubtitle = True  
 continue  
 if lookforsubtitle:  
 if not line.strip():  
 lookforsubtitle = False  
 else:  
 subtitle = line.strip()  
 subtitle = subtitle.strip(".")  
 title += ", " + subtitle  
 if ("\*\*\* START" in line) or ("\*\*\*START" in line):  
 collect = startseen = True  
 paragraph = ""  
 continue  
 if ("\*\*\* END" in line) or ("\*\*\*END" in line):  
 endseen = True  
 break  
 if not collect:  
 continue  
 if (titleTemp) and (authorTemp):  
 outlines.append(one)  
 outlines.append("<title>")  
 outlines.append(titleTemp)  
 outlines.append("</title>")  
 outlines.append("<author>")  
 outlines.append(authorTemp)  
 outlines.append("</author>")  
 outlines.append(two)  
 outlines.append("<title>")  
 outlines.append(titleTemp)  
 outlines.append("</title>")  
 outlines.append("<author>")  
 outlines.append(authorTemp)  
 outlines.append("</author>")  
 outlines.append(three)  
 authorTemp = False  
 titleTemp = False  
 continue  
 if not line:  
 paragraph = paragraph.strip()  
 for term in remove:  
 if paragraph.startswith(term):  
 paragraph = ""  
 if paragraph:  
 paragraph = paragraph.replace("&", "&")  
 outlines.append(paragraph)  
 outlines.append("</p>")  
 paragraph = "<p>"  
 else:  
 paragraph += " " + line  
  
 # Get author lastname  
 authorLastName = authorLastName.split(" ")  
 authorLastName = authorLastName[-1].lower()  
  
 # Get short title  
 shortTitle = title.replace(":", "\_")  
 shortTitle = shortTitle.replace(",", "\_")  
 shortTitle = shortTitle.replace(" ", "")  
 first\_cap\_re = re.compile('(.)([A-Z][a-z]+)')  
 all\_cap\_re = re.compile('([a-z0-9])([A-Z])')  
 shortTitle = first\_cap\_re.sub(r'\1\_\2', shortTitle)  
 shortTitle = all\_cap\_re.sub(r'\1\_\2', shortTitle).lower()  
 shortTitle = shortTitle.replace("\_\_", "\_")  
  
 # Compose a filename. Replace some illegal file name characters with alternatives.  
 filename = url.split("/")  
 ofn = filename[-1]  
 ofn = authorLastName + "\_" + shortTitle[:150] + ".xml"  
 ofn = ofn.replace("&", "")  
 ofn = ofn.replace("/", "")  
 ofn = ofn.replace("\"", "")  
 ofn = ofn.replace(":", "")  
 ofn = ofn.replace(",", "")  
 ofn = ofn.replace(" ", "")  
 ofn = ofn.replace("txt", "xml")  
  
 outlines.append("</div></body></text></TEI>")  
 text = "\n".join(outlines)  
 text = re.sub("End of the Project Gutenberg .\*", "", text, re.M)  
 text = re.sub("Produced by .\*", "", text, re.M)  
 text = re.sub("<p>\s+<\/p>", "", text)  
 text = re.sub("\s+", " ", text)  
  
 # Save the file to the file manager  
 savedFiles += "<li>" + ofn + "</li>"  
 fileManager.addUploadFile(text, ofn)  
  
 # Read from a list of urls  
 #outputDir = "/Path/to/your/ProjectGutenberg/TEI/Output/files/"  
 #urls = ['http://www.gutenberg.org/cache/epub/42324/pg42324.txt']  
 #for url in urls:  
 # ofn, text = beautify(url, outputDir, url)  
 # print(ofn+":")  
 # print(text[:10000])  
  
 # Save the file to the file manager  
 #fileManager.addUploadFile(doc, fileName)  
  
 message = savedFiles + "</ol>"  
  
 # Save the file manager  
 managers.utility.saveFileManager(fileManager)  
  
 return render\_template('gutenberg.html', message=message)

@app.route(“/downloadScrubbing”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/module’ def downloadScrubbing(): # The ‘Download Scrubbed Files’ button is clicked on scrub.html. # Sends zipped files to downloads folder. fileManager = managers.utility.loadFileManager() return fileManager.zipActiveFiles(‘scrubbed.zip’)

@app.route(“/doScrubbing”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/module’ def doScrubbing(): fileManager = managers.utility.loadFileManager() # The ‘Preview Scrubbing’ or ‘Apply Scrubbing’ button is clicked on scrub.html. session\_manager.cacheAlterationFiles() session\_manager.cacheScrubOptions()

# saves changes only if 'Apply Scrubbing' button is clicked  
savingChanges = True if request.form["formAction"] == "apply" else False  
previews = fileManager.scrubFiles(savingChanges=savingChanges)  
#tagsPresent, DOEPresent = fileManager.checkActivesTags()  
  
if savingChanges:  
 managers.utility.saveFileManager(fileManager)  
  
data = {"data": previews}  
import json  
data = json.dumps(data)  
return data

@app.route(“/getAllTags”, methods=[“GET”, “POST”]) # Tells Flask to load this function when someone is at ‘/module’ def getAllTags(): “”" Returns a json object with a list of all the element tags in an XML file. “”"  
 fileManager = managers.utility.loadFileManager() text = “” for file in fileManager.getActiveFiles(): text = text + " " + file.loadContents()

from bs4 import BeautifulSoup  
soup = BeautifulSoup(text, 'xml')  
tags = []  
[tags.append(tag.name) for tag in soup.find\_all()]  
tags = list(set(tags))  
  
sorted(tags, key=unicode.lower)  
import json  
data = json.dumps(tags)  
return data

# ======= End of temporary development functions =======

install\_secret\_key() app.debug = True app.jinja\_env.filters[‘type’] = type app.jinja\_env.filters[‘str’] = str app.jinja\_env.filters[‘tuple’] = tuple app.jinja\_env.filters[‘len’] = len app.jinja\_env.filters[‘unicode’] = unicode app.jinja\_env.filters[‘time’] = time.time() app.jinja\_env.filters[‘natsort’] = general\_functions.natsort

# app.config[‘PROFILE’] = True

# app.wsgi\_app = ProfilerMiddleware(app.wsgi\_app, restrictions = [300])

if **name** == ‘**main**’: app.run()