On Deck
Technical Documentation

K. Hosaka (kah), A. Huque (ash), L. Jim (lmj), M. Terry (mht) - 2-3-2020 - v1.02

Table of Contents

1. Technical Documentation Revision History 3

2. How to Install and Start On Deck 3

3. Software Requirements 3

4. Using On Deck 3

5. Main Source Code 5

- **5.1. On-Deck.py** 5
- **5.2. interface.py** 5
  - **start** 5
  - **initiateSavedQueue** 5
  - **rosterMenu** 5
  - **newRoster** 6
  - **downloadRoster** 6
  - **updateRoster** 6
  - **exportSummary** 6
  - **initiateNewQueue** 7
  - **presentationView** 7
  - **getOnDeckNames** 7
  - **keyInputs** 7
  - **updateOnDeck** 8
  - **updateHighlight** 8
  - **settingsMenu** 8
  - **testingMode** 8
  - **resetAll** 8
  - **onClose** 8
  - **testButton and testPresenting** 9
- **5.3. queue.py** 9
  - **removeStudent** 9
  - **checkQueue** 9
  - **flagStudent** 10
  - **getOnDeck** 10

5.4. roster.py 10
increment_called 10
update_roster 11
increment_flagged 11
increment_test_called 11
5.5. fileInput.py 11
reading_dict_py 11
reading_new_roster 12
parser 12
instructor_update_roster 13
export_roster 14
5.6. fileOutput.py 14
daily_output 14
term_output 15
testing_output 15
5.7. randomTester.py 15
testRandomness 15
6. Main Source Code Dependencies 15
6.1. daily_logs.tsv 16
6.2. fresh_roster.py 16
6.3. queueState.py 16
6.4. students.py 16
6.5. testing_reports.tsv 16
6.6. images/ 16
7. Additional Files Provided 17
7.1. rosters/ 17
7.2. developerTests/ 17

1. Technical Documentation Revision History

Date	Author	Description
1-22-2020	lmj	Created the initial document
2-02-2020	lmj	Wrote sections 2, 3, 5.1, 5.2, 5.6, 6, and 7
2-02-2020	mht	Wrote sections 5.3, 5.4
2-02-2020	ash	Wrote section 5.5
2-03-2020	lmj	Wrote section 4 and made minor edits
2-03-2020	mht	Wrote section 5.7 and minor edits
2-03-2020	ash	Formatting, grammar edits

2. How to Install and Start *On Deck*

Follow these steps:

1. Download the zip file containing *On Deck*
2. Unzip the file
   - On Mac: right click zip file, select "Open With" and choose "Archive Utility" or a similar application
3. Open Terminal and navigate into the folder you just unzipped
   - For example, run "`cd /Desktop/On-Deck/`"
4. Run "`python3 On-Deck.py`"
   - This starts the application

3. Software Requirements

On Deck has been tested to work on macOS High Sierra and Catalina using Python 3.7.6 or 3.8.1 and Tkinter 8.6. Tkinter is a part of Python's standard library. Python does not require a compiler. *On Deck* does use the following from Python's standard library: time, datetime, ast, importlib, and random.

4. Using *On Deck*

Start the software by using the command "`python3 On-Deck.py`" in the terminal. When closing the program, use the 'x' to exit the Home window and not '⌘Q' or Quit Python from the macOS Menu Bar as this will not allow the queue state to save nor output the daily log information.

To input a roster into the system, click on the Roster Information button from the Home window. In the Roster window, select Choose New Roster or Update Current Roster. Choose New Roster will overwrite any student information already in the system. Update Current Roster will allow the number of times and dates when a student was called and flagged to be saved while adding, removing, or updating student information. There is a Download Current Roster in the Roster window to retrieve the roster already in the system.

To get a summary of the students' performance, click on Performance Summary Report in the Roster window. This will let you download a .tsv file that indicates how many times a student was called and flagged as well as the dates they were called on.

To get information on who was called on and flagged during a session, open the daily_logs.tsv file. This file updates every time the application is exited using the 'x' on the Home window. Do not use '⌘Q' or 'Quit Python' because this will not allow the daily log to get updated with information from that session.

To test the randomization of the queue, click on the Settings button from the Home window. Then click on the Test Randomizer button. The application may need to be restarted if a new or updated roster was loaded into the application during the current session. The results can be viewed in the testing_report.tsv file.

To reset the entire application, click on the Reset Application button in the Settings window. This will clear all the saved information pertaining to students. This includes clearing the daily_logs.tsv file and the testing_report.tsv file. No roster will be in the system after using this button. A new roster will need to be loaded using the Choose New Roster button in the Roster window. The Reset Application button is not required to choose a brand new roster. However, this button allows the daily log file to also get reset.

To get the students who are on deck, click the Present button on the Home window. If the roster was updated during the current session, the application may need to be restarted. The four names on deck will be displayed in the On Deck window. This window will always be in front of any other windows on the screen. This includes when applications such as PowerPoint are in full screen mode. If desired, the window size can be reduced small enough to only show one name at a time. The currently selected student will be highlighted yellow. To switch between students, use the left and right arrow keys. Make sure the On Deck window is active when using the arrow keys. When the student to dismiss is selected on the screen, use the up arrow key to dismiss them from being on deck. However, if the student should also be flagged, use the down arrow key.

5. Main Source Code

These files contain the code written by the On Deck Development Team.

5.1. On-Deck.py

The On-Deck.py file contains the main function to run the entire program. It relies on interface.py and launches the application window.

5.2. interface.py

The interface.py file contains the code that affects the way the user interacts with the program. This is the file that utilizes tkinter and creates the user interface. It relies on all of the other python files with the exception of On-Deck.py.

start

The initial startup of the application begins with the function `start` which initializes the Home window. It sets the size, location, and title of the Home window. If a previously imported roster is detected, it calls `initiateSavedQueue` to load the saved queue information. This function relies on the content in the images folder to create three buttons. The buttons call the functions `rosterMenu`, `presentationView`, and `settingsMenu`. The closing protocol is also defined by calling the `onClose` function.

initiateSavedQueue

When the application already has a saved queue, `initiateSavedQueue` is called. First it accesses the saved information in students.py and starts the studentQueue. To start the studentQueue, instances of Roster and Queue are initialized. Then it accesses the saved information in queueState.py and sets the studentQueue's activeRoster and passiveRoster to the saved information.

rosterMenu

When the Roster Information button is clicked, `rosterMenu` is called. This starts a new window (setting the size, location, and title) and adds the buttons pertaining to rosters to it. These buttons call the functions `newRoster`, `downloadRoster`, `updateRoster`, and `exportSummary`.

newRoster

When the Choose New Roster button is clicked, `newRoster` is called. It first refreshes the students.py file to check if a roster is already present. If a roster is present, a message box opens to warn the user that this will override the current roster and the application will need to be restarted. Assuming the user says ok and not cancel, or is uploading a roster for the first time, the macOS file dialog window opens. It is set to require a file in .tsv format. If the user clicks cancel, the file dialog window closes and nothing else happens. If a file is actually selected, then restartRequired or restartRecommended is set to True depending on if there was already a roster in the system or not. The file name is then passed into `parser` in fileInput. If `parser` returns 1, then there was an error in the file and a message is displayed to the user. If there is no detected error in the file, `initiateNewQueue` is called.

downloadRoster

When the Download Current Roster button is clicked, `downloadRoster` is called. If there is a roster in the system, then the macOS file dialog window opens. This allows the user to choose the name of the file they want to save the performance summary as and select the location it saves to. The roster can only be downloaded in .tsv format. If there was no roster in the system, then it tells the user there is no roster and does not prompt them to save any file.

updateRoster

When the Update Current Roster button is clicked, `updateRoster` is called. It reloads the students.py file, so it has the most current information. If there is no roster already in the system, it tells the user to use Choose New Roster instead of Update Current Roster. If there is a roster in the system, it opens a message box asking the user if they are sure they want to continue. If they do not click cancel, then the macOS file dialog window opens. Similar to choosing a new roster, it requires a .tsv file, sets restartRequired to True, will display an error message if the file is bad, and call `initiateNewQueue` if the file was good. The main difference is that it calls fileInput's `instructor_update_roster` function instead of parser.

exportSummary

When the Performance Summary Report button is clicked, `exportSummary` is called. If there is no roster in the system, it will notify the user. Otherwise, it will open the macOS file dialog window and the user can choose the output file name as well as set the save location. The file format is set as a .tsv file. If the user actually entered a file name, then students.py is reloaded for the most up to date information and fileOutput's function `term_output` is called.

initiateNewQueue

When a valid new or updated roster is imported, `initiateNewQueue` is called. It reloads students.py for the most up to date information. Then it reads the contents and converts the string type dictionary to be an actual dictionary. The dictionary is then used to initialize an instance of Roster which is used to initialize an instance of Queue. This queue instance is set to the interface's studentQueue.

presentationView

When the Present button is clicked, `presentationView` is called. First, it needs to check whether a restart is required. If a restart is required (from a new roster being imported), then a message box is displayed to the user that they need to restart the application. If a restart is not required, then it will check that there is actually a roster in the system. If no roster is found, then it will display a message box notifying the user. If no restart is required and there is a roster in the system, then exportDailyLog is set to true. This allows the program to know that presentation mode was entered and upon exiting the home menu, the daily log should be written. A new window is started, and the size, location, and title are set. This window is unique because an attribute of "-topmost" is set. This makes the window remain in the foreground even when the application is not the active window.

The function then calls `getOnDeckNames` which returns a list of four names. It then sets the interface's four on deck names which are set through tkinter's StringVar(). This allows the names to be updated. All four names are displayed in the window as labels. Since the labels need to be next to each other horizontally, the side is set to left. The first name always starts as the selected name. It gets configured with raised relief and a yellow background. The interface's highlighted name and studentQueue's studentSelected variables are set to 0 to reflect that the first name is highlighted. The other three names have relief set to flat and background set to white as they are not the currently highlighted name. Finally, the listeners for the four arrow keys are set.

getOnDeckNames

When present mode first starts, getOnDeckNames is called. Using the interface's instance of the studentQueue, `getOnDeck` is called from the Queue class.

keyInputs

While present mode is running, `keyInputs` is called if an arrow key is pressed. If the right or left arrow keys are pressed and the highlighted name is not already the rightmost of leftmost, `updateHighlight` is called, and the studentQueue's studentSelected is also updated. If the up arrow key is pressed, then the studentQueue's `removeStudent`

function is called. If the down arrow key is pressed, then the studentQueue's `flagStudent` function is called. Both the up and down arrow portions retrieve the new list of names using the studentQueue's `getOnDeck` function and pass the returned list to `updateOnDeck`.

updateOnDeck

When the up or down arrow keys are pressed, `updateOnDeck` is called. This function simply sets the StringVar() for all four on deck names to the names that were passed to the function. Then it updates the actual present window.

updateHighlight

When the left or right arrow keys are pressed, `updateHighlight` is called. It first sets the relief and background of the label for the name that is currently highlighted to flat and white. Then, using the position number passed to the function, it sets what the new highlighted position is and sets the label's relief to raised and background to yellow. It then updates the actual present window.

settingsMenu

When the Settings button is clicked, `settingsMenu` is called. This starts a new window (setting the size, location, and title) and adds buttons to it. It relies on one image in the images folder. The Test Randomizer button calls `testingMode`, and the Reset Application button calls `resetAll`.

testingMode

When the Test Randomizer button is clicked, `testingMode` is called. If a restart is required, a message box will notify the user. If a restart is not required, then it will reload the students.py file for the most up to date information. If there is no roster, then a message box will notify the user that there is no roster. Before calling the randomTester's `testRandomness`, it will ask the user if they are sure they want to continue. The `testRandomness` function handles the rest if the user says okay.

resetAll

When the Reset Application button is clicked, `resetAll` is called. A message box will ask the user if they are sure. If they say okay, five files are reset to their initial state. These files are: students.py, fresh_roster.py, queueState.py, daily_logs.tsv, and testing_report.tsv.

onClose

When the 'x' for the Home window is clicked to exit, `onClose` is called. If exportDailyLog is set to true, then the presentation mode was entered, and the daily log should be updated. It

reloads students.py to have the most up to date information. Then it calls fileOutput's `daily_output`. If there is a roster in the system, then the queue gets saved in queueState.py. Finally, all of the windows are closed and the application is no longer running.

testButton and testPresenting

There are also two functions that were created for testing purposes called `testButton` and `testPresenting`. The function `testButton` simply prints a message to the terminal. If a button is not mapped to another function, this function is great to test that the button is clickable. The function `testPresenting` contains sleeps and calls `updateOnDeck` and `updateHighlight` to test that they work.

5.3. queue.py

The queue.py file contains the code that generates and maintains a random list of student names for cold-calling. The queue is initialized with a roster provided by the user. The names of each student are taken from the roster and randomly put into a list. Each name is then appended to activeRoster, a list that maintains the ordering of all students who have yet to be called on. Students who are called/flagged are removed from activeRoster and placed in passiveRoster until the active queue is empty. This file imports random to shuffle lists and generate random numbers, as well as imports datetime for accurate flagging.

removeStudent

When students are dismissed or flagged for removal, `removeStudent` gets called. This function takes in a single parameter, testing, which indicates whether or not to increment the active student dictionary. The default value of this variable is false. The student that is removed from the queue is saved and returned as removeStudent. Once a student is removed, the datetime module is utilized to call `today`, which saves the current date. If the testing variable is false, the roster function `increment_called` is used to indicate which day the student was cold-called. removedStudent is then added to the passiveRoster so as to not lose track of the student. The activeRoster is then checked against `checkQueue` to make sure it is not empty. This function returns the information of the student that was removed.

checkQueue

Whenever a student is removed from the queue, this function gets called. Its main purpose is to make sure the activeRoster has enough students in it for the queue to keep running. This function first checks to see if there are more than 4 students in the activeRoster. If there are fewer than 4, then that means that almost all of the students have been called on. Should this be the case, passiveRoster, which contains the remaining students, is shuffled using the random module's `shuffle` function. The passiveRoster is then appended to the end of the activeRoster. In this

manner, all students will be called once before running into a student a second time. The passiveRoster variable is then reset to an empty list.

flagStudent

Whenever the user presses the "down" key on a name while in the main presentation view, `flagStudent` will be called. Since flagged students are to be removed from the queue, `removeStudent` is called. The results of this function call is saved in the variable removedStudent. This function then passes the unique student ID of the removed student to the roster function `increment_flagged` to register in the dictionary that they have been flagged.

getOnDeck

Whenever the state of the queue updates, namely through dismissing or flagging students, this function is called to get a list of four names of "on deck" students. This is done by iterating through the student roster dictionary using their unique student ID numbers. Each student's name is then pulled out and saved into a names list. The first four values of the names list are then returned and reflected in the interface. These four names comprise the "on deck" students.

5.4. roster.py

The roster.py file is utilized to create and modify instances of class rosters. A dictionary containing all of the students information (student_dict), with unique student IDs as the keys, is required for initialization. A flagged_list variable is provided to keep track of which students on the roster have been flagged for further review. A session_list variable is provided to track which students got called on in that particular session of running the *On Deck* software. The test_dict variable is used to differentiate the roster in the presentation state and the testing state. Changes made to the roster in the testing state are not reflected in the presentation state.

increment_called

Whenever a student is dismissed from the queue, this function is called from the Queue function `removeStudent`. This function takes in two variables as parameters. The first being the unique student ID of the student in string form, and the second being the date that the student was called in string form. The roster's student_dict is then modified to reflect the call by incrementing the number of calls to that student by 1. Additionally, the date of the call is saved in the dictionary. The student is also appended to the session_list to designate that they were called on in that instance of the program. `update_roster` is then called to save these changes in the additional files.

update_roster

Whenever changes are made to the central roster dictionary, student_dict, `update_roster` is called. This function opens up a file called "students.py." In the first instance of the program, this file will be empty until a roster is loaded in. The most recent instance of the student_dict is then written to students.py, and the file is closed. In this manner, the information of the dictionary (including who has been dismissed/flagged and when) is saved in an additional file so it can be reloaded in future instances of the program

increment_flagged

If the "down" key is pressed in the active queue screen, then a student is flagged and this function gets called from queue.py. This function takes only one parameter: the student's unique student ID number as a string. It will first update the roster's student_dict by incrementing the number of times the student has been flagged by 1. The student's ID number is then appended to the global flag list, flagged_students. At the end of the function, `update_roster` is called to reflect the changes made to the student_dict in the additional files.

increment_test_called

When the user initiates a test by clicking the "test randomizer" button, this function is called from testRandomizer.py when students are "called on" in the testing queue. It takes the student's unique ID number as a parameter. Upon the dismissal of a student in test mode, this function will increment the roster's test_dict to reflect how many times a student was called. This is made separate from `increment_called` so as to not have testing mode create changes in the actual student_dict.

5.5. fileInput.py

The fileInput.py file contains the code that takes in a new or updated .tsv file and processes it into the internal roster format. It also contains a function to download the current roster for the user to update in another text editor.

reading_dict_py

`reading_dict_py` is called in `instructor_update_roster` and `export_roster` within the fileInput.py file. `reading_dict_py` has one argument, the name of a python file. This python file is expected to contain only "students_dict = {dictionary}".
`reading_dict_py` opens the file that has been passed in and reads through the contents of the python. This function then strips "students_dict = " within the file so that the variable, roster_dict, only contains the dictionary with the roster information. The file is then closed and ast.literal_eval is called to evaluate the string form of the dictionary as a data type dictionary.

`reading_dict_py` returns the data type dictionary for `instructor_update_roster` and `export_roster` to use.

reading_new_roster

`reading_new_roster` is called in `parser` and `instructor_update_roster` within the fileInput.py file. `reading_new_roster` has one argument, the name of a tab-delimited file that has a class roster. This file should be formatted as:
<first_name><tab><last_name><tab><UO_ID><tab><email_address><tab><phonetic_spelling>
The roster may have more fields after <phonetic_spelling> but `reading_new_roster` will not parse those additional fields. This function first opens the given file and reads the contents of the file. It then iterates through the contents and strips the contents of the file first by the newline character and then splits by tabs. `reading_new_roster` then iterates to the now stripped and split list, checking that the file is properly formatted. The function will return a one if there are less than five fields or if the third field is not a student ID that starts with '951,' as all student IDs at the University of Oregon do. If a one is returned, the GUI will raise a warning for the user that the input file is formatted incorrectly. Assuming one is not returned, `reading_new_roster` then inserts an empty list at the end of each student's information and two zeros at the beginning of each student's information. The empty list is for the dates of participation, the first zero is for the number of times a student participates, and the second zero is for the time of times a student has been flagged. This function then appends the students' information, including the three new fields, to a dictionary where the student ID is the key and the value is a list of all the student's information. `reading_new_roster` then closes the file that is being parsed and returns the dictionary for `parser` and `instructor_update_roster` to utilize.

parser

`parser` is called upon in `newRoster` in interface.py to parse through the given roster. `parser` has one argument, a tab-delimited file that is formatted as the same input as `reading_new_roster`. `parser` calls `reading_new_roster` to parse the input file and create a dictionary. The function then checks if the returned value is a one, meaning there was an error in parsing the file, and returns a one if this is the case so the GUI can raise a warning to the user. Assuming one is not returned, the `parser` function then opens a new file named "students.py" and writes the dictionary that was returned from `reading_new_roster`. It then closes "students.py." The function then opens another new file named "fresh_roster.py" and writes the same dictionary before closing the file. "students.py" will be utilized for the fileOutput.py, interface.py, and roster.py to update student participation dates, student flags, and the number of times a student participates. "fresh_roster.py" will be utilized by rosterTester.py, interface.py, and `export_roster` to prevent any overwrite of "students.py" and maintaining an unmodified roster. `reading_new_roster` then returns a zero so the GUI is aware there was no errors during this function call.

instructor_update_roster

`instructor_update_roster` is called upon in `updateRoster` in interface.py to allow the user to modify the roster. `instructor_update_roster` has one argument, a tab-delimited file that formatted as the same input as `reading_new_roster`. This input is intended to be an updated version of the roster currently utilized by the *On Deck* application. `Reading_new_roster` then calls `reading_dict_py` on "students.py," the file containing the current dictionary, in order to parse the file and create a dictionary data type of the current dictionary in *On Deck*. This dictionary is assigned to the variable original_roster. Then `reading_new_roster` is called on the given file to parse that file and store the roster in a dictionary which is assigned to the variable updated_roster. The function then checks if `reading_new_roster` returned one, meaning there was an error with parsing the file. If one was returned the function `instructor_update_roster` returns one as well so the GUI can notify the user there is an issue with the input. Assuming one is not returned, the original dictionary, from "students.py," is copied to a new variable dictionary in order to allow for modifications with prematurely overwriting "students.py."

`instructor_update_roster` then checks if any new students have been added to the updated roster dictionary by looking for the differences in keys between the updated_roster dictionary and the original roster dictionary. If there are new keys that have been added, a set of those keys are created and assigned to the variable added_students. The function iterates through added_students and adds the key with the relevant value to the dictionary variable.

`instructor_update_roster` then checks if any students have dropped the class, meaning they have been removed from the roster, by looking for the differences in keys between the original roster dictionary and the updated_roster dictionary. If there are any removed keys, a set of those keys are created and assigned to the variable removed_students. The function iterates through removed_students and removes the key/value pairs from the dictionary variable.

`instructor_update_roster` will then iterate through all keys that exist in both updated_ditionary and original_dictionary and overwrite all student information in the dictionary except for student IDs, number of times participated, number of times flagged, and the days of participation. This is to account for any modifications the instructor makes to the roster such as adjusting phonetic spelling or correcting an error with name spelling.

Finally, the dictionary variable will contain a dictionary that has been modified with the updated_roster while maintaining the students and their participation records of the original roster. `instructor_update_roster` then opens "students.py" and writes in the dictionary variable and then closes the file. The function also opens "fresh_roster.py" and writes in the

dictionary variable and then closes the file. Finally the function returns 0 meaning there were no errors.

export_roster

`export_roster` is called by `downloadRoster` in interface.py so the user can export the roster currently utilized by *On Deck*. This function takes two arguments, filename, the name of the .py file that has the dictionary with student information, and exportFile, the name specified by the user for what they want the exported roster file name to be.

First, `export_roster` calls on `reading_dict_py` to parse the filename variable and have a dictionary data type of the roster information that is assigned to roster_dict. In practice, filename will always be "fresh_roster.py" but this was created as a variable since, during the development process, there was discussion on whether this function would be used for also producing the daily logs. At the end, our team decided not to but the filename input variable remained.

`export_roster` then opens a file named to what the user has specified and iterates through the keys of roster_dict, writing the key and the key's value on the same line. Since every key's value is a list, there is a second for loop that iterates through the indices of the list and writes them as tab separated values. For every new key a newline character is added. After the function finishes writing the file, the file is closed and the function returns None.

5.6. fileOutput.py

The fileOutput.py file contains the code that writes the daily log, performance summary report, and testing report.

daily_output

The function `onClose` in interface.py calls `daily_output`. If the file daily_logs.tsv does not exist in the On-Deck folder, then the file is created. Any data already in the file is read and saved. The file header with that day's date is appended to the file. Then students.py is reloaded for the most up to date information. The entire dictionary of students is then looped through. If the student is in the session list, then their name and email are written to the daily output file. Before writing to the file, it checks if the student is also in the flagged list. If the student is in the flagged list, they will have an 'X' before their name in the log. Finally, the data that was already in the file is appended. This results in the appearance that the new information was prepended to the file.

term_output

The function `exportSummary` in interface.py calls `term_output`. The students.py file is reloaded for the most up to date information. The interface passes the name of the file the output should be written to. A header is initially written to the file. Then all the students in the dictionary are looped through. The following information is written to the file for each student: Times Called, Times Flagged, First Name, Last Name, UO ID, Email, Phonetic Spelling, and List of Dates Called.

testing_output

The function `testRandomness` in randomTester.py calls `testing_output`. It opens the testing_report.tsv file and writes the header to it. Then it loops through all of the students in the dictionary. It calculates the percentage the student was called on, then writes the number of times they were called on, their first and last name, and the percentage to the report.

5.7. randomTester.py

The randomTester.py file contains the code that tests the randomness of the queue. The variable totalCalls represents how many random students are called in the test. The roster is imported from the fresh_roster.py code dependency. The originalStudentSelected variable maintains the selected state of the queue from before the test begins.

testRandomness

Whenever a randomizer test is initiated from the settings menu, this function gets called. It takes in a copy of the queue through the parameter studentQueue. Next, the student_dict, which comes from studentQueue, is iterated through. Each name is assigned to a results dictionary index so that their calls can be tracked. Once the results dictionary is set up, the function enters a loop wherein it generates 1000 random integers between 0 and 3. These numbers correlate to the first through fourth spot of "on deck" students. On each iteration, `removeStudent` is called in testing mode to remove the student at the randomly generated index. To keep track of how many times a student was called, `increment_test_called` is used on each iteration. Finally, once 1000 calls are executed, the function will call `testing_output` to generate the testing output file. The active queue is then restored and the test ends.

6. Main Source Code Dependencies

These files are required for the software to run.

6.1. daily_logs.tsv

This file is used by fileOutput.py. It starts off initially empty and will eventually contain a log of which students were called and if they were flagged on any given day.

6.2. fresh_roster.py

This file is used by fileInput.py, randomTester.py, and roster.py. This file is initially an empty dictionary. When a roster is imported, this file maintains a clean copy of the roster information. This clean copy gets utilized when testing the queue randomization.

6.3. queueState.py

This file is used by interface.py. This file initially contains an empty activeRoster list and an empty passiveRoster list. When starting the application, if there is a roster in the system, then the initialized queue utilizes the information from these lists. When exiting the application, if there is a roster in the system, the queue state is saved into these lists. While the queue is ultimately what needs this saved information, the interaction happens through interface.py.

6.4. students.py

This file is used by fileInput.py, fileOutput.py, interface.py, and roster.py. This file initially contains an empty dictionary. When a roster is imported, the information is saved in this file. The number of times a student has been called on and flagged as well as the dates the student was called are stored in this file.

6.5. testing_reports.tsv

This file is used by fileOutput.py. This file starts empty. This is the file where the results of the randomization test are put when the user runs it. The results are overridden each time the test is run.

6.6. images/

This folder contains four icons that are used for buttons by interface.py. There is one additional image that was intended to be the application icon.

7. Additional Files Provided

These files can be used by a developer to test the software.

7.1. rosters/

This folder contains four rosters that can be utilized to test the software. The file badRoster.tsv is intended to make the fileInput.py functions return an error and test how the application handles it. The file exampleRoster.tsv contains eight entries for a simplified test of the system. The file roster.tsv has thirty-three students in it and allows for a more realistic test of the application. The file updatedRoster.tsv has removed some names from roster.tsv and added some entries from exampleRoster.tsv. This file allows for the testing of the Update Current Roster button to make sure added and removed names are handled properly.

7.2. developerTests/

This folder contains files used by developers during the development process. The file fake_queue_calls.py was used to test the functionality of saving the number of times a student was called on and flagged as well as test the output functions. The Makefile was simply used to clean the directory during the testing phase. The file originalTkTest.py was used to see how tkinter works.