On Call
Technical Documentation

A. Archer (aa), K. Hosaka (kah), A. Huque (ash), L. Jim (lmj), M. Terry (mht)

3-08-2020 - v2.02

Table of Contents

1. Technical Documentation Revision History

Date	Author	Description
2-12-2020	lmj	Created the initial document
2-13-2020	lmj	Started specifying file and function names
2-19-2020	lmj	Updated section 5
2-27-2020	ash	Added information to 5.3
2-28-2020	ash	Finished writing 5.3
2-28-2020	kah	Rough Draft of 5.6
2-29-2020	kah	Updated 5.6
2-29-2020	ash	Updated 5.3
3-01-2020	kah	Finalized 5.6
3-01-2020	lmj	Updated section 5.2
3-02-2020	ash	Updated 5.4
3-03-2020	ash	Updated 5.4 deletePreferences
3-03-2020	aa	Added section 5.5
3-04-2020	lmj	Updated section 5.2
3-04-2020	ash	Wrote 5.3
3-05-2020	mht	Updated section 5.4, added section 5.7
3-04-2020	ash	Edited 5.3
3-05-2020	mht	Finished sections 5.4 and 5.7
3-06-2020	ash	Edited 5.3
3-06-2020	lmj	Finished section 2, 3, 4, and 5.2
3-06-2020	kah	Proofreading document
3-07-2020	lmj	Proofreading document, small edits, and suggestions.
3-07-2020	ash	Proofreading document, made edits/suggestions
3-08-2020	ash	Proofreading document

2. How to Install and Start *On Call*

Follow these steps:

1. Download the zip file containing *On Call*
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-Call/`"
4. Run "`python3.7 On-Call.py`"
   - This starts the application

3. Software Requirements

On Call has been tested to work on macOS High Sierra and macOS Catalina using Python 3.7.6 and Tkinter 8.6. Tkinter is a part of Python's standard library. Python does not require a compiler. *On Call* also uses the following from Python's standard library: ast, functools, importlib, and random.

4. Using *On Call*

The following sections correspond to the windows in *On Call*.

4.1. On Call - Home

- Start the software by using the command "`python3.7 On-Call.py`" in the terminal.
- From the Home window select either "RA Preferences" or "Schedule". Clicking these buttons will open their respective windows. These two windows can be open at the same time, but several iterations of the same window cannot be opened.

4.2. On Call - RA Preferences

- On the initial RA Preferences window, or the window that appears when no RAs are in the system, click "Import Preferences".
  - Select a CSV file to import. This file can either contain one line with a single RA, or several RAs on several lines of the file. You can continue to import more files later, but you cannot select more than one file during a single import.
  - After selecting a file, the RA Preferences window will close. If you would like to view a list of the imported RAs, click "RA Preferences" on the Home window to reopen it.

- To import more RAs, click “Import Preferences” again. This button is now located after the list of current RAs.
- You can also use “Import Preferences” to update an already imported RA’s preference information. The RA’s preferences will be updated to match the preferences in the newly imported file. Note: additions and updates made through “Import Preferences” cannot be undone using the “Undo” button. After selecting a file to import, the RA Preferences window will close.
- If an RA that was imported needs to be deleted, select the RA’s name in the dropdown menu that appears after the list of RAs. Once selected in the dropdown menu, click “Delete”. The system will confirm you want to permanently delete the selected RA. This action cannot be undone using the “Undo” button. The RA Preferences window will close.
- If you want to remove all of the RA preferences in the system, then click “Delete All”. This button is located after the list of current RAs and to the right of the single RA deletion button. This action cannot be undone. The RA Preferences window will close.
- To update a specific preference for an RA, select either the weekday preference or weekend off request that will be updated. The Edit RA Preference window will open, where the change can be made.
- Once a change has been made to an individual preference, the “Undo” button in the upper left of the window can be clicked. Clicking “Undo” will permanently revert the most recent change back to its previous state. Note: there is no redo button. Clicking this button will cause the window to close.

4.3. On Call - Edit RA Preference

- The name of the RA you are editing a preference for appears first. Then the preference you are changing for that RA is listed.
- When editing a weekday preference, a dropdown menu of the possible days is provided.
- When editing a weekend preference, a dropdown menu of the possible weeks is provided as well as a 0 which indicates no preference.
- Click “Save” once the new preference has been selected in the dropdown menu. This will cause the Edit RA Preference window to close as well as the RA Preferences window.

4.4. On Call - Schedule

- On the initial Schedule window, or when there is no schedule in the system, click “Generate New Schedule”.
    - There must be at least ten RAs in the system for a schedule to be generated. A message will be displayed if there are not enough RAs.

    - Each weekend must have at least half of the RAs present, so no more than half of the RAs can request the same weekend off. A message will be displayed if this has occurred in their preferences.
    - An RA must list different weekdays in their preferences. If an RA lists the same weekday more than once in their preferences, a message will be displayed to the user.
    - If all of the preferences are valid and a schedule can be generated, the Generate Schedule Settings window will open.
- When there is a schedule in the system, this window will display all of the shift assignments.
- Along the left hand side, there are labels for the rows indicating the week number and primary versus secondary slots. Along the top side, there are labels for the columns indicating the shifts for each week. The table is filled in with the names of the RAs who are assigned to each shift.
- To change the RA assigned to a specific shift, click on the name currently assigned to it. The Edit Schedule window will open, where the change can be made.
- Once a change has been made to a shift, the "Undo" button in the upper left of the window can be clicked. Clicking "Undo" will permanently revert the most recent change back to its previous state. Note: there is no redo button. Clicking this button will cause the window to close.
- When there is a schedule in the system, the "Generate New Schedule" button appears under the schedule. If RA preferences have changed, use this button to create a new schedule based on the current preferences in the system. If you want to change the settings used to produce the schedule, use this button to generate a new one.
- To export the schedule to a CSV file, click "Export Schedule". This will ask you where you would like to save the schedule to as well as a name for the file.
- To export a summary of each RA's shifts, click "Export Summary". This will ask you where you would like to save the summary to as well as a name for the text file.
- To delete the schedule in the system, click "Clear Schedule". This will permanently delete the schedule and cannot be undone.

4.5. On Call - Edit Schedule

- The label before the dropdown menu indicates which shift is getting changed.
- In the dropdown menu, select the RA you want to assign to the shift.
- Click "Save" once the appropriate RA has been selected in the dropdown menu. This will cause the Edit Schedule window to close as well as the Schedule window.

4.6. On Call - Generate Schedule Settings

- The first dropdown menu in this window is to select an RA who has earned a gold star. The RA selected here is guaranteed to get their first choice in weekday preferences. The option of "None" is default, but it is encouraged to consider which RA is most deserving of their top preference.
- The second dropdown menu in this window is to select the tiebreaker option. This is to guarantee a schedule can be made even if most of the RAs prefer working on the same weekdays.
    - Random is the default option and will randomize which RA does not get any of their weekday preferences when a conflict occurs. The gold star RA is exempt from this randomization.
    - Alphabetical Order (Last Name) will choose the first conflicting RA based on last name. They will not get assigned to any of their weekday preferences.
    - Numerical Order (ID Number) will choose the first conflicting RA based on their ID number. They will not get assigned to any of their weekday preferences.
- The next two sets of dropdown menus are for selecting a pair of RAs who cannot share a shift together. The coordinator can use this at their discretion if they think two RAs do not work well together. The default option is "None". If an RA is selected in one menu and "None" is still selected in the other, no dis-allowed pairing will be marked in the system. If the same name is selected in both menus, then the user will be given a message when they try to save that it needs to be changed.
- Once all of the options are set how the coordinator would like, click "Save". This will close the Generate Schedule Settings window as well as the Schedule window. To view the generated schedule, click "Schedule" on the Home window.

5. Main Source Code

These files contain the code written by Team LLC.

5.1. On-Call.py

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

5.2. onCallViewer.py

The onCallViewer.py file contains the code for the graphical user interface. This is the file that uses Tkinter, and it relies on all of the other code files with the exception of On-Call.py.

5.2.1. Main Window

home

`home` acts as the main function for the user interface. It creates the root, or home, window. The home screen has two buttons. The first button opens the RA preferences window by calling `preferencesView`. The second button opens the schedule window by calling `scheduleView`. When this window is closed, the application stops.

5.2.2. RA Preferences Window

preferencesView

`preferencesView` creates the actual window pertaining to RA preferences. If this window is already open, a new one will not be created and the existing one will be pulled to the front. The raPreferences dictionary is refreshed and the window is sized based on the number of RAs in the system. If there are no RAs the window is small and simply displays a message to the user stating there are no RAs along with a button to import RAs. This label and button are centered horizontally in the window. If there are more than eighteen RAs, the window is simply larger than normal.

When there is at least one RA in the system, the user is presented with an undo button in the upper left corner. This button calls `undoPreferences` when clicked. It is disabled when there is nothing to undo. Then the headers for the columns are added to the screen. Weekday Preferences spans three columns because all of these columns correspond to this header. The same is true for Weekend Off Requests. These labels are configured to use size fourteen font and an underline.

The RAs in the system and their preferences are added to this window. `preferencesView` iterates through the raPreferences dictionary. If the key is not ‘1’, ‘2’, or ‘3’ then the key refers to an RA. The ID numbers and names are appended to separate lists. The index is the same in both of these lists, so when the user selects a name, it can get matched to the correct ID number quickly. In addition to creating these lists, a label is created for each RA’s name and six buttons are created for their preferences. These buttons call `editRA` and pass the index of the RA in these separate lists as well as a number indicating which preference slot the button corresponds to.

When there are RAs in the system, additional buttons are added to the window under the list of RAs. The same import button from when there are no RAs in the system is displayed here, and it calls `importPreferences`. Then there is a label for a dropdown menu indicating that the dropdown menu is for deleting an RA. The dropdown menu uses the list of names created from the raPreferences dictionary and relies on `updateDeletionChoice` to track which RA is selected. Then there is a button called "Delete" that the user clicks to delete the RA selected in the dropdown menu. This button calls `deleteRA`. There is also a "Delete All" button which calls `deleteAllRAs` and empties the dictionary of information.

A special protocol is specified for when this window is closed. `closePreferences` gets called.

undoPreferences

`undoPreferences` calls `undo` in input.py. `closePreferences` is also called to force the user to reopen the preferences window to see the updated information.

importPreferences

`importPreferences` refreshes the raPreferences dictionary. If there are already twenty-five RAs in the system, then a message is displayed to the user notifying them they cannot import more until some are deleted. If the limit has not been reached, then a message is displayed to the user explaining how this button can be used. The message differs based on if there are any RAs in the system already.

When the user has not reached the limit of RAs and has chosen to continue with the import, the computer's file dialog system is used. This requires the use of CSV files. If a file is chosen, the path and file name are passed to input.py's `importFile` function. If an error is registered, a message is displayed to the user; otherwise, `closePreferences` is called to force the user to reopen the window for the newly updated information.

deleteRA

`deleteRA` checks if an RA has been selected using the dropdown menu. If no RA is selected, a message is displayed to the user to select an RA. If an RA has been chosen, a message is displayed to the user to make sure they want to permanently delete the RA. Provided the user chooses to continue, input.py's `deletePreferences` is called followed by `closePreferences` to require the refreshing of the window.

deleteAllRAs

`deleteAllRAs` checks with the user that they truly want to permanently delete all of the saved RA information. If this is the case, input.py's `resetPreferences` is called followed by `closePreferences` to require the refreshing of the window.

closePreferences

`closePreferences` checks if the Edit RA Preference window is open. If it is, then this window is closed by calling `closeEditRA`. The preferences window is also closed. Then all of the variables set when opening this window are reset to None.

5.2.3. Edit RA Preferences Window

editRA

`editRA` creates a small window that lets the user edit the specified preference field of a specific RA. If one of these Edit RA Preference windows is already open, a new one is not opened and the existing one is pulled to the front. A label with the selected RA is displayed at the top of the window. Then a label indicating which preference will be changed is displayed. Under these labels, a dropdown menu is added to the window. The dropdown menu is either the weekday options or weekend options, depending on which preference is being updated. The dropdown menus rely on `updateWeekdayChoice` and `updateWeekendChoice`, respectively. Finally, a save button is added to the window. The button calls `updateRA` when clicked and passes the preference's list index of the RA as well as the number corresponding to which preference is being updated. The labels, dropdown menu, and button are centered horizontally. If the user chooses to click 'x' to exit this window, then a protocol has been set and `closeEditRA` will be called.

updateRA

`updateRA` first checks to see if a choice has actually been made in the dropdown menu. If no choice has been selected, a message is sent to the user. Depending on which preference field is being updated, either a weekday choice or weekend choice will be passed to input.py's updatePreferences function. Additionally, this function will only be called if a new choice is being made. This prevents someone from clicking undo several times without seeing changes because they cannot change a value to the same value. After updating the dictionary, `closeEditRA` and `closePreferences` are called to require a refresh of the preferences window.

closeEditRA

`closeEditRA` closes the Edit RA Preferences window and resets all of the variables pertaining to this window to None.

5.2.4. Schedule Window

scheduleView

`scheduleView` creates the window pertaining to the shift assignments information. If this window is already open, a new one will not be opened and the existing one will be pulled to the front. The shiftAssignments dictionary is refreshed for current information. If there is no schedule in the system, then a small window is opened. This window shows a short message telling the user to generate a schedule and provides the generate button. This button calls `generateNewSchedule`.

If there is a schedule in the system, then the standard window will open. This places an undo button in the top left of the window. The button calls `undoShiftChange`. If there are no changes to be undone, then this button is disabled. Next, the headers are added as labels to the window. They are given a font of size fourteen text and an underline. The row labels are given the same text size but no underline. The row labels are added in the same for loop that adds all of the shift information to the window.

The shift information is added to the window by iterating through the shiftAssignments dictionary. Each of these are buttons that call `editSchedule`. This function is passed the week number, a 0 or 1 to indicate primary or secondary, and the index corresponding to which shift is being changed.

Under the schedule, some more buttons are given. There is a button to generate a new schedule, so if RA preferences are changed, a new schedule can be produced. This button calls `generateNewSchedule`. Then there is the button to export the schedule which calls `exportSchedule`. There is also an export summary button which calls `exportSummary`. Finally, there is a button to clear the schedule from the system which calls `clearSchedule`.

A special protocol is specified for when this window is closed. `closeSchedule` gets called.

undoShiftChange

`undoShiftChange` simply calls output.py's `undo` function. Then `closeSchedule` is called to force the user to reopen the window, refreshing it.

generateNewSchedule

`generateNewSchedule` checks to see if a settings window is already open. If this window is already open, a new one is not created, and the existing window is pulled to the front. Then, input.py's `generateCheck` is called. This ensures there are enough RAs in the system and all of the preferences are set acceptably. This function returns a 1, 2, or 3 when there is an issue. Depending on the number, a message is sent to the user informing them of the issue that exists.

A 0 means there are enough RAs and all of the preferences are allowed. If there is already a schedule in the system, a warning will be displayed to the user telling them the current schedule will be overwritten. If they choose to continue, the settings window is opened. Once the settings window is closed via the save button, output.py's `generateSchedule` is called. If this function returns an error indicator, a message is sent to the user.

exportSchedule

`exportSchedule` refreshes the shiftAssignments dictionary to have the most recent information. If there is no schedule in the system, then a message is sent to the user notifying them there is nothing to export. If there is a schedule, then the computer's file dialog system is used. The coordinator must save the file as a CSV but can choose the location and name of it. If a filename and path are chosen, then output.py's `exportFile` function is called. If this returns an error indicator a message is displayed to the user.

exportSummary

`exportSummary` refreshes the shiftAssignments dictionary to have the most recent information. If there is no schedule in the system, then a message is sent to the user notifying them there is nothing to export. If there is a schedule, then the computer's file dialog system is used. The coordinator must save the file as a generic text file but can choose the location and name of it. If a filename and path are chosen, then exportSummary.py's `exportShiftInfo` function is called.

clearSchedule

`clearSchedule` checks with the user that they want to permanently delete the schedule in the system. If this is the case, then output.py's `resetAssignments` is called. `closeSchedule` is also called to force the user to reopen the schedule window.

closeSchedule

`closeSchedule` checks to see if the Edit Schedule window is open. If this window is open, it gets closed by calling `closeEditSchedule`. It also checks to see if the settings window is open for generating a new schedule. If this window is open, it gets closed by calling `closeSettings`. The schedule window is closed and corresponding variables are reset to None.

5.2.5. Edit Schedule Window

editSchedule

`editSchedule` checks to see if an existing Edit Schedule window is open. If this is the case, then a new window is not opened, and the existing one is pulled to the front. The raPreferences dictionary is refreshed for the most current RA information. The names of all of the RAs are put into a list to create a dropdown menu. A label is created to inform the user which shift is being changed. Then a dropdown menu is provided. This dropdown menu is a list of all of the RAs in the system to choose from. The function `updateChangeRaChoice` keeps track of which RA is selected. Finally, there is a save button. This calls `updateShift`. All of the widgets are centered horizontally in the window. A special protocol is set for when the window closes. This calls `closeEditSchedule`.

updateShift

`updateShift` checks if an RA has been selected in the dropdown menu. If there has not been a selection, the user is notified. If the selected RA is not the same RA who currently has the shift, then output.py's `updateSchedule` is called. `closeEditSchedule` and `closeSchedule` are also called to ensure the schedule gets refreshed.

closeEditSchedule

`closeEditSchedule` simply closes the Edit Schedule window and resets the variables to None.

5.2.6. Settings Window

settingsView

`settingsView` refreshes raPreferences for the most current information. It puts 0 and 'None' into separate lists which are then appended by the id numbers and names of RAs respectively. The index between these two lists match the same option and are tracked this way to create the dropdown menus.

A label and dropdown menu are added to the window for the user to select the gold star RA. This dropdown menu relies on `updateGoldStarChoice` to track which RA has been selected.

A label and dropdown menu are added to the window for the user to select a tiebreaker choice. This dropdown menu relies on `updateTiebreakerChoice` to track which option has been selected.

A label for the first dis-allowed pairing and two dropdown menus are added to the window for the user to select a bad pairing. The two dropdown menus rely on `updatePairingOne` and `updatePairingTwo` to track the RAs that have been selected.

A label for the second dis-allowed pairing and two dropdown menus are added to the window for the user to select an additional bad pairing. The two dropdown menus rely on `updatePairingThree` and `updatePairingFour` to track the RAs that have been selected.

The five dropdown menus that have RAs as options all use the same list. This means the RAs will be displayed in the same order in each menu. Each of these dropdown menus is set to 'None' to start. The tiebreaker menu is set to 'Random' to start.

A save button is also added to the window. This calls `saveSettingChoices`. A special protocol is set to close the window which calls `closeSettings`.

saveSettingsChoices

`saveSettingsChoices` gets the ID number of the selected RA for gold star (or a 0 if 'None' is selected) as well as the number that corresponds to the tiebreaker choice. This information is then passed to input.py's `setGoldStar` and `setTiebreaker` functions when they are called.

The bad pairing options are all initially set to zero. If both the first and second pairing choices have selected an RA, then a check will be done to make sure the same name was not chosen in both dropdown menus. If this is the case, a message will be sent to the user and the settings will not save and close. If two different RAs are chosen in the dropdown menus, then the corresponding bad pairings are set to the ID numbers of the RAs.

If both the third and fourth pairing choices have selected an RA, then a check will be done to make sure the same name was not chosen in both dropdown menus. If this is the case, a message will be sent to the user and the settings will not save and close. If two different RAs are chosen in the dropdown menus, then the corresponding bad pairings are set to the ID numbers of the RAs.

All four of the bad pairing options come together and are passed to input.py's `setBadPairings` function.

The variable settingsSaved is set to true so `generateNewSchedule` knows the settings window was closed using the save button. It also calls `closeSettings` and `closeSchedule`.

closeSettings

`closeSettings` closes the settings window and resets all of the related variables back to None. The variable settingsClosed is a special tkinter variable that uses the method set and in this case is True, not None.

5.2.7. Button Testers

testButton, testRaEdit, testSchedEdit
These three functions simply print to the terminal. `testButton` was used as a placeholder before all of the buttons could be mapped to their current functions. `testRaEdit` was used to make sure the arguments would be passed correctly before writing the current function used. `testSchedEdit` was also used to make sure arguments were passed correctly before mapping the current function to the buttons.

5.2.8. Update Dropdown Menu Choice

updateDeletionChoice, updateWeekdayChoice, updateWeekendChoice, updateChangeRaChoice, updateGoldStarChoice, updateTiebreakerChoice, updatePairingOne, updatePairingTwo, updatePairingThree, updatePairingFour
All of these functions relate to tracking the choice selected in the dropdown menus in the various windows.

5.3. input.py

The input.py file contains the functionality of the RA Preferences module. This code takes in a new or updated CSVfile of RA preferences and processes it into the internal raPreferences dictionary. It also contains functions to delete a specific RA, update RA information, undo updates, store information about selected settings, download the current information stored in the system, and reset the dictionary to be empty.

5.3.1. Input

inputPreferences
`inputPreferences` is called in `importFile` within input.py. `inputPreferences` has one argument, the name of a file. This function parses the given file and adds the information to the internal raPreferences dictionary. This input file should be formatted as: <student ID>,<first and last name>,<weekday preference 1>,<weekday preference 2>,<weekday preference 3>,<weekend requested off 1>,<weekend requested off 2>,<weekend requested off 3>.

`inputPreferences` 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 by the newline character and then splits by commas. The function then iterates through the now stripped and split list containing the contents of the file. `inputPreferences` then does six error checks to ensure the contents of the file are the right format to be processed:

- The first field of each line is a valid student ID by checking if it starts with “951,” as all student IDs at the University of Oregon do.
- There are exactly eight fields on each line.
- There is an additional loop that iterates through the three weekday preference fields on each line and makes sure that the given weekday preference is a valid weekday (Sunday-Thursday) by comparing it to the list weekdays which is defined within `inputPreferences`.
- The weekend requested off falls within the range of 0-10.
- The weekend requested off is actually a number and not words or a symbol.
- The three given weekday preferences are all unique weekdays. This is to ensure that someone does not type “Monday, Monday, Monday” for example.

If any of these error checks fail the function returns a 1 to notify the GUI an error has occurred. Assuming all the error checks pass and the input file is correctly formatted, the input file is then parsed into the raPreferences dictionary. `inputPreferences` now closes the file and returns the raPreferences dictionary.

readingDictPy

`readingDictPy` is called in `importFile, deletePreferences`, `resetPreferences`, `setGoldStar`, `setTiebreaeker`, `setBadPairings`, `exportRApreferences`, `generateCheck`, `updatePreferences`, and `undo` within input.py. `readingDictPy` has one argument, the name of a Python file. This Python file is expected to contain only “raPreferences = {}” and the preferences for each RA inside the dictionary. `readingDictPy` opens the file that has been passed in and reads through the contents of the file. This function then strips “raPreferences = ” within the file so that the variable, raPreferences, 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. `readingDictPy` returns the dictionary for `importFile`, `deletePreferences`, `resetPreferences`, `setGoldStar`, `setTiebreaeker`, `setBadPairings`, `exportRApreferences`, `generateCheck`, `updatePreferences`, and `undo` to use.

save

`save` is called in `updatePreferences` within input.py. This function saves changes that have occured to a global list of lists, called inputUpdates, in order for the `undo` function to call from that list. `save` has three arguments: current_dictionary, idNum, and index. current_dictionary is an instance of the raPreferences dictionary. idNum is a student ID number, a key within current_dictionary. index is a number which is the designated point within a list, that is the value to the idNum key, at which a change has occurred. Having these three arguments, `save` first accesses the information currently stored within current_dictionary at key idNum at index and saves it to the variable old. Then, this function creates a new list which contains the idNum key, index, and old. This list is saved to the variable change. change is then appended to the global dictionary inputUpdates and zero is returned. Now the global dictionary inputUpdates has a list to keep track of what the information at the specified key and value index used to be.

5.3.2. Preferences

importFile

`importFile` writes the contents of a given file to raPreferences.py, a file that contains the contents of the raPreferences dictionary. This function has one argument, the name of a file. The input file should be formatted as: <student ID>,<first and last name>,<weekday preference 1>,<weekday preference 2>,<weekday preference 3>,<weekend requested off 1>,<weekend requested off 2>,<weekend requested off 3>. `importFile` calls `inputPreferences` to parse the file and check for errors. Then `importFile` checks if the returned value from `inputPreferences` is a one, meaning an error has occurred. If it is, then `importFile` also returns one so the GUI knows that there was an error with the given file. If no errors occur, `inputPreferences` returns a dictionary which is stored in the variable updated_dict. `importFile` then calls `readingDictPy` to read and retrieve the dictionary that is currently stored in raPreferences.py. The current dictionary is stored in the variable original_dict. Then a new variable, dictionary, contains a copy of original_dict so no modifications are made to original_dict while processing the information.

First, `importFile` looks for the difference in keys between updated_dict and original_dict. The keys that are unique to updated_dict are added to a list called added_RAs. Then, for each key in added_RAs, a new key is made in the dictionary, with its values being the relevant key/value pair from updated_dict.

Second, `importFile` checks for keys that exist in both original_dict and updated_dict. If there are keys that exist in both dictionaries, the variable dictionary overwrites its information, that was received when it was copied from original_dict, with the new information in updated_dict.

Third, `importFile` checks if the total number of keys in the dictionary are greater than 25. If so, it returns a two to notify the user that given input has exceeded the maximum size of an RA team.

Finally, this function opens raPreferences.py, writes dictionary into it, closes the file, and returns zero so the GUI knows no error has occurred.

deletePreferences

`deletePreference` removes the key/value pair of the given RA and rewrites the raPreferences dictionary without the specified key. This function takes in one argument, a string that is a key in the dictionary. Each key is a student ID so this input should be a student ID of someone in the dictionary. First, `deletePreferences` calls `readingDictPy` to read the dictionary that is currently in raPreferences.py. This dictionary is stored in the variable current_dictionary. Then, this function deletes the inputted key from current_dictionary. `deletePreferences` then iterates through the global dictionary inputUpdates, which stores the information when changes occur for the `undo` function. If the deleted student ID has any information in inputUpdates, it pops that index of the list. This is to prevent the user from attempting to undo a change that occured for an RA that was later deleted. `deletePreferences` then opens raPreferences.py, writes the now modified current_dictionary into it, closes the file, and returns zero so the GUI knows no error has occurred.

resetPreferences

`resetPreference` rewrites the `raPreferences` dictionary in `raPreferences.py` to be an empty dictionary. This function takes in no arguments. `resetPreference` opens `raPreferences.py,` writes "raPreferences = {}" into it and closes the file. Then, `resetPreference` takes the global list inputUpdates, which stores the information when changes occur for the `undo` function, sets it to an empty list, and returns zero.

setGoldStar

`setGoldStar` stores the "gold star" setting information to key "1" in the raPreferences dictionary. This function takes in one argument, a string that is a student ID. `setGoldStar` first calls `readingDictPy` to read the dictionary that is currently in raPreferences.py. This dictionary is stored in the variable current_dictionary. Then, this function adds a key "1" to current_dictionary with the value being the inputted student ID. `setGoldStar` opens raPreferences.py, writes the now modified current_dictionary into it, closes the file, and returns zero.

setTiebreaker

`setTiebreaker` stores the tiebreaker setting information to key "2" in the raPreferences dictionary. This function takes in one argument, an integer that is either 0, 1, or 2. 0 correlates to the random setting being selected, 1 is alphabetical by last name, and 2 is numerical by student ID. `setTiebreaker` first calls `readingDictPy` to read the dictionary that is currently in raPreferences.py. This dictionary is stored in the variable current_dictionary. Then, this function adds a key "2" to current_dictionary with the value being the integer relating to the selected option. `setTiebreaker` opens raPreferences.py, writes the modified current_dictionary into it, closes the file, and returns zero.

setBadPairings'

`setBadPairings` stores the "bad pairings" setting information to key "3" in the raPreferences dictionary. This function takes in four arguments, student1-student4, which are four strings of different student IDs. `setBadPairings` first calls `readingDictPy` to read the dictionary that is currently in raPreferences.py. This dictionary is stored in the variable current_dictionary. Then, this function adds a key "3" to current_dictionary with the value being a list of lists. The first list is of student1 and student2 and the second list is of student3 and student4. Each list represents two RAs that can not be scheduled together. Both lists are put in a single list which is the value for the key "3." `setBadPairings` opens raPreferences.py, writes the modified current_dictionary into it, closes the file, and returns zero.

exportRaPreferences

`exportRaPreferences` writes the contents currently stored in raPreferences.py to a file for the user to read. This function takes in one argument, the name of a file to write the raPreferences dictionary into. `exportRaPreferences` first calls `readingDictPy` to read the dictionary that is currently in raPreferences.py. This dictionary is stored in the variable current_dictionary. Then, this function opens the inputted filename to begin writing the raPreferences.py dictionary to. First, however, `exportRaPreferences` tries to delete the setting keys "1", "2", and "3" from current_dictionary so that it won't be written to the file. If those keys do not exist yet and a KeyError rises, the function continues. `exportRaPreferences` iterates through each key in current_dictionary and first writes the key and then a comma. Then, the function iterates through the length of the value of the key and writes the value followed by a comma. Each key/value information is written on a new line. Finally, `exportRaPreferences` closes the file and returns zero.

generateCheck

`generateCheck` does error checking before generating a schedule. This checks that there are a minimum of ten RAs that have been inputted to *On Call*, the smallest size an RA team could be, and that there are no weekends that more than half the team has requested off. This function takes in no arguments. `generateCheck` contains a list, weekends_off, to keep track of the number of times each weekend is requested off by an RA. weekends_off has ten zeros to start off with. The index of the list plus one corresponds to weekend requested off (so the number of students who requested weekend one off can be found at index zero of the list). This function also has a list called requests to keep track of every single weekend off an RA has requested.

`generateCheck` calls `readingDictPy` to read the dictionary that is currently in raPreferences.py. This dictionary is stored in the variable current_dictionary. If a schedule has already been generated once, the dictionary will contain the keys "1", "2", and "3" for the setting information. `generateCheck` will first try to delete these keys if they exist to prevent any errors arising while error checking. If those keys do not exist a KeyError arises. An exception is created for KeyError which will allow the function to continue. `generateCheck` then creates a list called key_list of all the keys in current_dictionary. First, the function checks if the length of key_list is less than ten, meaning less than ten RAs were inputted into *On Call*. If so, 1 is returned so the GUI can notify the user that not enough RAs have been inputted. Next, for each key in key_list, the last three values containing the requested weekends off at the related key in current_dictionary are added to the list requests. Within this loop, there is an additional check to make sure no RA has been given more than one of the same preference (such as "Monday, Monday, Tuesday" for example). If an RA has been given repeated preferences the function returns a three so the GUI can notify the user that at least one RA has been assigned multiple of the same preferences. Then, for each number in requests, the number is converted to an int. If the item is a 0 the function will pass since this means the RA did not request a weekend off. If the number is any number 1-10 the relevant indices is incremented by one.

Finally, `generateCheck` checks if more than half the RA team has requested the same weekend off. If the RA team is an odd number of RAs, it is half the RA team rounded up. The function will iterate through weekends_off and if any of the indices is greater than the length of half of key_list plus one it returns a two so the GUI can notify the user that too many RAs have requested the same weekend off. Else, meaning the RA team is an even number of RAs, the function will iterate through weekends_off and if any of the indices is greater than the length of half of key_list it returns a two. If no errors occur, `generateCheck` returns zero.

updatePreferences

`updatePreferences` updates the raPreferences dictionary with new information. This function takes in three arguments, idNum, index, newPref. idNum is a string of a student ID, the key in the raPreferences dictionary. Index is an int relating to the index within the value of idNum that the change is occuring. newPref is a string of what the index will be changed to. `updatePreferences` first calls `readingDictPy` to read the dictionary that is currently in raPreferences.py. This dictionary is stored in the variable current_dictionary. Then, this function calls `save` to store the key, index, and what the value was before the change to the global dictionary inputUpdates. `updatePreferences` now accesses the related key and index and overwrites the current value with newPref. Finally, `updatePreferences` opens raPreferences.py, writes current_dictionary into it, closes the file, and returns zero.

undo
`undo` pulls from the global list inputUpdates to rewrite the raPreferences dictionary with a previous action. This function accepts no arguments. `undo` first tries to call `readingDictPy` to read the dictionary that is currently in raPreferences.py. This dictionary is stored in the variable current_dictionary. Then, this function removes the last index of the global dictionary inputUpdates, utilizing the builtin pop() method, and assigns the value to a variable called previous. If an IndexError arises a one is returned to let the GUI know the list is empty. If no error arises, then the function continues and current_dictionary overwrites its current values stored in previous, reverting current_dictionary to the previous state. Finally, `undo` opens raPreferences.py, writes current_dictionary into it, closes the file, and returns zero.

5.4. weekdayScheduler.py

The weekdayScheduler.py file contains the functionality of the weekday scheduler module. Upon initialization, the `init` function will utilize the dictionary in raPreferences.py to get a dictionary of all raPreferences, as well as the user's settings. This file will then assign every RA to a weekday that they will work for the entire term. Lastly, RAs are given primary/secondary shifts based on that weekday information.

weekdayShifts
The `generateSchedule` function in output.py will trigger this function to start. `weekdayShifts` is the main driver of this file. It will first check that the raPreferences dictionary was loaded correctly, returning an error if it was not. Once it is clear that there is data to work with, this function will call a myriad of other functions throughout the file to generate the scheduling information. First, RAs are assigned to weekdays using the `weekdayShifts` function. These assignments are then passed to `scheduleShifts` to assign each RA to primary and secondary shifts throughout the term. A 3D list of weekday assignments is then returned.

getRaInformation
`getRaInformation` is called once: upon initialization of the entire class. The purpose of this function is to get rid of unnecessary information from the main raPreferences dictionary returned from raPreferences.py. First, the raPreferences dictionary is loaded from the raPreferences.py file. Each key in this dictionary is iterated over. For each RA, their weekend preferences are removed. This is so as to limit the amount of data to be worked with throughout the function. For each RA, two extra integers are appended. The first represents the total number of shifts, the second represents the total number of primary shifts. Both of these values are incremented later. If a key in the raPreferences dictionary is not a student ID number, namely if the key is in the list [‘1’,’2’,’3’], then that means it is a user setting. This gets saved in a separate dictionary so as to

keep the preferences apart from the user settings. Both of these dictionaries are returned in the form of a tuple.

assignDays

`assignDays` is called from the main `weekdayShifts` driver function. This function takes in no variables, and will export a dictionary where each key is a day of the week, and each value is a list containing RAs that are assigned to that day of the week. To start, this function first calculates the minimum for how many RAs should be scheduled each day (raPerDay). An raList is created from the keys (student ID numbers) of the raPreferences dictionary so as to keep track of who has already been assigned a day of the week. If the user did not input a gold star RA, meaning that no RA is saved in the settings dictionary, then the main loop starts. If a gold star RA is present, they get their first choice of weekday and are removed from the raList.

There are 3 major loops that occur at this point in `assignDays`. Each loop will iterate over every RA that is left in the raList, checking their first, second, and then third preferences respectively. In each iteration, the RAs preference of day is pulled out of the raPreferences dictionary. If that day does not have the minimum number of RAs (raPerDay) scheduled to it yet, and if a bad pair is not present on that day, then the RA will be scheduled to work on that weekday. To check for bad pairs throughout this process, the `badPairInDay` function is called, which will return True if an RA cannot be scheduled that day due to bad pairing inputs from the user. Upon the completion of each of these main loops, all RAs that were assigned throughout that loop are removed from the raList.

Once all possible preferences are set, it is time to schedule all of the remaining RAs. However, before we give a certain day extra RAs, we first check to make sure that all days have the minimum number (raPerDay). While each day of the week is not full, RAs will be selected using the tiebreaker setting input from the user. The function `tiebreaker` is called to get the ID number of a single RA. If `badPairInDay` returns False, then that RA will be added to the weekday.

Once every weekday has the minimum number of RAs assigned to it, a check is done to see if there are any additional RAs remaining. These leftover RAs can be assigned to any day, as every day will already have the minimum number of RAs required. As such, their preferences are looped through again. If an RA cannot be added to one of their preferences, then they will be assigned to the first day possible. After this, a dictionary containing each weekday and the RAs assigned to that day is returned.

tiebreaker

`tiebreaker` is called from `assignDays` to generate an RA to be assigned when there are no preferences left. There are two parameters: tieSetting, which is an integer correlating to which tiebreaker mode to use based on the user's input. The second parameter is raList, which constitutes the available RAs to choose from. If the tieSetting is 0, then a random RA is selected from raList and is returned to the user. If the tieSetting is 1, then a list is created holding all of the last names of the RAs. This list is then sorted and the RA with the lowest name alphabetically is returned to the user. The final tiebreaker is when tieSetting is 2. Here, each student ID number is iterated over. The RA with the lowest student ID number is returned to the user.

badPairInDay

`badPairInDay` is called from `assignDays` to check whether or not an RA can be scheduled on a certain day based on the presence/absence of a bad pair. The bad pairs are provided by the user before generating the schedule. This function takes in a list of all RAs that are currently assigned to a given day, as well as the RA to check. At the beginning of the function, a check is done to see if the RA is present in either of the bad pair lists. If they are not, then the function will return False. If the RA is present in one of the bad pair lists, then we iterate through every RA in the weekday list. If any of these RAs are in a bad pair list with the original RA, then the function will return True. This indicates that the given RA cannot be scheduled on that day. Otherwise the function will return False.

scheduleShifts

`scheduleShifts` is called from `weekdayScheduler` once all RAs are assigned to a day. This function takes one parameter, which correlates to the dictionary of weekdays and the RAs assigned to them. A copy of this dictionary is made to avoid aliasing. The function then begins to step through a loop 10 times, once for each week. For each day in a week, we iterate over every single RA so as to find the RA with the fewest shifts. This is the RA that will be scheduled next. If there is only one RA with the fewest shifts, then another RA is just randomly selected.

Once the RAs with the fewest shifts are pulled out, a similar loop will pull out the RA with the fewest number of primary shifts. This RA will then be added to the primary shift list for that week. The shift counter and primary shift counter sections of raPreferences are then incremented. The secondary shift is assigned randomly. Once the day has been filled out, the function will move to the next day, then the next week. All the while adding these elements to a cumulative list to be returned back to the user.

5.5. weekendScheduler.py

The weekendScheduler.py file contains the functionality of the Weekend Scheduler module. This file accesses the dictionary in raPreferences.py and assigns RAs to fill the weekend shifts in the schedule.

weekendShifts

`weekendShifts` is called by output.py. This function accepts no input and returns a 3D list, a list of two lists containing RA names, that correspond to the weekend shifts in the schedule. The function imports and copies the raPreferences.py dictionary and adds new index variables corresponding to the number of total shifts as well as primary and secondary shift counts. The shift count is used to sort RAs fairly by ensuring no RA has more shifts than average. The primary and secondary shift counts are used to ensure a balance between how often an RA is the primary RA on call. The RAs are sorted into weekend shifts based on their given preferences and the disallowed pairings.

The schedule output is formatted as: [[[week 1 primary schedule], [week 1 secondary schedule]], [[week 2 primary schedule], [week 2 secondary schedule]], … , [[week 10 primary schedule], [week 10 secondary schedule]]]. The primary schedule consists of the primary RAs on call for each weekend shift during that week. Likewise, the secondary schedule consists of the secondary RAs on call for each weekend shift during that week. Each week's primary and secondary schedule list is in the format of: [Friday RA, Saturday Day RA, Saturday Night RA, Sunday RA]. Uniquely, week one has no RAs assigned to the Sunday shift, so this shift is filled with an "X".

5.6. output.py

The output.py file contains the functionality of the Shift Assignments module. This file compiles the weekday and weekend shift information into one schedule. It takes in two 3D arrays of schedules returned by weekendScheduler.py and weekdayScheduler.py. This module contains the functions to generate the shift schedule, export the schedule into a CSV file, save the current state of the schedule, update changes to the schedule, and undo updates.

generateSchedule

`generateSchedule` is called by onCallViewer.py. This function calls `weekendShifts` in weekendScheduler.py and `weekdayShifts` in weekdayScheduler.py. These functions both return a 3D list of the shift assignments they generated. This input is formatted as: [[[week 1 primary schedule], [week 1 secondary schedule]], [[week 2 primary schedule], [week 2 secondary schedule]], … , [[week 10 primary schedule], [week 10 secondary schedule]]]. The primary schedule consists of the primary RAs on call for each shift during that week. Likewise, the secondary schedule consists of the secondary RAs on call for each shift during that week.

`generateSchedule` creates a temporary dictionary called assignments which contains the generated schedule containing the weekend and weekday shift assignments. The keys in assignments are the weeks in a term (1 - 10). Each key's value is a list of the primary and secondary schedules. The format of assignments is: {week number: [[week 1 primary schedule], [week 1 secondary schedule]], [[week 2 primary schedule], [week 2 secondary schedule]], …, [[week 10 primary schedule], [week 10 secondary schedule]]}. Primary and secondary schedules are a list of the shifts assignments in the format: [Sunday Day Assignment, Sunday Night Assignment, Monday Assignment, Tuesday Assignment, Wednesday Assignment, Thursday Assignment, Friday Assignment, Saturday Day Assignment, Saturday Night Assignment].

`generateSchedule` adds the weekday shift assignments to assignments by iterating through the 3D list of weekday assignments and appending the values to the assignments dictionary. To add the weekend shifts to assignments, the function iterates through the 3D list of weekend assignments. First, the primary and secondary Sunday Day shift assignments are prepended to the beginning of the schedules. Following this prepend, the rest of the primary and secondary weekend assignments are added to the dictionary. This is done to follow the formatting order as specified in the last sentence of the previous paragraph.

`generateSchedule` then opens the file shiftAssignments.py, which contains the compiled and finalized shift assignments for the other modules in the software architecture to interact with. `generateSchedule` writes the assignments dictionary to shiftAssignments.py and renames the dictionary to shiftAssignments.

`generateSchedule` returns 0 upon success and 1 if an error is raised while attempting to open the files.

exportFile

`exportFile` is called by onCallViewer.py, which passes in a filename as specified by the user. This function is called when a user wants to export the shift assignments schedule into a CSV file.

`exportFile` writes a header for the output file indicating the shifts in the order: Sunday day, Sunday night, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday Day, Saturday Night. `exportFile` iterates through the shiftAssignments dictionary in shiftAssignments.py and writes the contents into the output file. With each iteration, the week numbers are written to the output file. The primary and secondary shift assignments are written next to its corresponding week number.

`exportFile` will return 0 upon success and 1 if an error is raised while attempting to open the files.

rewriteSchedule

`rewriteSchedule` is a helper function that is passed an updated dictionary of shift assignments that will overwrite the existing shiftAssignments dictionary in shiftAssignments.py. This function is called by the `updateSchedule` and `undo`.

save

`save` tracks the updates made to the shiftAssignments dictionary in shiftAssignments.py. It saves every state that the shiftAssignments dictionary has been in into a global list, outputUpdates. `save` is used by the `undo` function that returns the shiftAssignments dictionary to a previous state.

`save` makes a copy of the current shiftAssignments dictionary from shiftAssignments.py and names the copy old_assignments. The function then appends the old_assigments to outputUpdates.

updateSchedule

`updateSchedule` is called by onCallViewer.py if the user wants to update a shift assignment. It is passed the week number of the shift the user is changing, a 0 or 1 for whether the primary or secondary shift assignments are being modified, the index of the name being changed, and the new RA's name that will be assigned to that shift.

`updateSchedule` calls the `save` function as described above to save the current state of the shiftAssignments dictionary in shiftAssignments.py. `updateSchedule` then copies the current shiftAssignments dictionary to a temporary dictionary called new_assignments. The dictionary new_assignments is modified by the changes as specified by the parameters passed in by onCallViewer.py. The function calls `rewriteSchedule` and passes in the new_assignments dictionary to overwrite the current shiftAssignments dictionary in shiftAssignments.py.

undo

`undo` is called by onCallViewer.py when the user wants to undo the updates that they made during the current session. When `undo` is called, shiftAssignments dictionary in shiftAssignments.py gets overwritten with the previous state of the dictionary before the last dictionary modification.

The previous state of shiftAssignments is popped from outputUpdates, a list that contains all of the states that shiftAssignments has been in, and is saved as a variable called last_state.

`rewriteSchedule` is called with last_state as the parameter to undo the current state and replace it with the most previous state before modifications were made.

resetPreferences

`resetPreference` rewrites the shiftAssignments dictionary to shiftAssignments.py to be an empty dictionary. This function takes in no arguments. `resetPreference` opens shiftAssignments.py, writes "shiftAssignments = {}" into it, closes the file, and returns zero.

5.7. exportSummary.py

The purpose of this file is to create a summary of each RA's shift information for the coordinator. This differs from the schedule produced and exported by output.py.Instead, a text file is generated containing each RA's name, total shifts, what weekdays they were assigned, how many primary shifts they have for weekdays, what weekends were assigned, and how many weekend primary shifts they have.

exportShiftInfo

All of the functionality for this file is done in this function. It takes in one parameter, a string containing the desired file name. First, the function loads in the last generated schedule from shiftAssignments.py. A dictionary, raDict, is created to count a number of values including: weekday/weekend primary and secondary shifts, weekdays assigned, and weekends assigned.

To get these values, the function iterates through all ten weeks of assignments. For each week, there is a primary list and a secondary list. Each of these lists are iterated through. Each time an RA is encountered, one of the weekday primary, weekday secondary, weekend primary, or weekend secondary counters gets incremented. The weekdays are differentiated from weekends by checking the index of the list. Since the lists are ordered, everything from index 1–5 are weekdays and the rest are weekends. While the function increments these counters, additional checks are done to keep track of the days an RA has been assigned to. Two lists are used to keep track of these assigned days.

Once all of this information is tracked, the function opens the file specified by the parameter in write mode. For each RA in raDict, a myriad of information is written to the file for the user. Each RA is separated by a blank line following their information, then the name of the next RA. The output file is a text file. Nothing is returned from this function as the text file occurs as a side effect.

6. Main Source Code Dependencies

These files are required for the software to run.

6.1. raPreferences.py

This file is used by onCallViewer.py, input.py, weekdayScheduler.py, and weekendScheduler.py. This file is initially an empty dictionary. When RA preferences are imported or updated, the information is saved in this file. This allows the information to be persistent between sessions.

6.2. shiftAssignments.py

This file is used by exportSummary.py, onCallViewer.py, and output.py. This file is initially an empty dictionary. When a schedule is generated, the shift assignment information is saved in this file. This allows the information to be persistent between sessions.

7. Additional Files Provided

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

7.1. developerTestFiles/

This folder contains an example dictionary of raPreferences. This dictionary was used to test the functionality of pieces of code before actual RA preferences could be imported and saved into the dictionary. Similarly, test_end.py and test_week.py are static lists that can be used instead of calling the schedulers. These files were used before the schedulers were finished so other parts of the code could start testing. Finally, there is a makefile which simply removes a file that was produced during the testing phase of output.py.

7.2. testInputs/

This folder contains several different CSV files that can be used as inputs. RA 1-16 are all CSV files containing a single line. These sixteen files contain sixteen different RAs. These same sixteen RAs are the ones in 16 RAs.csv. This file represents the typical staff size. There are also two CSVs with more RAs than normal: 20 RAs.csv and 25 RAs.csv. The maximum number of RAs the system can handle is twenty-five. The example.csv file contains five RAs and the example 1-5 files are each of those RAs individually. The file Same weekday chosen.csv gives an example where the weekday preference for at least one RA is the same as another one of their weekday preferences. The file Same Weekend Off.csv gives an example of too many RAs all requesting the same weekend off.