CoolCall: A Cold-Call Assist Program Programmer Documentation

Arden Butterfield, Quinn Fetrow, Derek Martin, Amy Reichhold, Madison Werries

January 28, 2022

0. Programmer Documentation Revision History

Date	Author	Description
1-28-2022	Amy	Outlined the documentation based on the examples.
1-29-2022	Amy	Completed the first draft.
1-30-2022	Amy	Finished the document based on input from the team.
1-30-2022	Arden	Reviewed for final submission.

1. Introduction

This document provides necessary information on the CoolCall cold-calling assist program in terms of its source code, including important classes and their member variables and methods. The documentation is based on the structure and content of the EyeDraw v3.0 Programmer's Documentation [1] and World Tax Planner Programmer's Documentation [2] examples provided in the CIS 422 course, as taught by Professor Anthony Hornof at the University of Oregon.

All code is written to be runnable in Python 3.7 through 3.10, and was written for the ColdCall v1.0 initial release on January 30, 2022.

The documentation assumes a sophomore-level understanding of Python and object-oriented programming.

2. Program Files

The program is structured very loosely around a model-view-controller (MVC) design pattern. The instructor_interaction_model.py file represents the main program entry point and provides the majority of the core program logic, and acts as a model, while communicating with the back end modules such as the queue, roster, and log manager. The display.py file represents the view and thus manages the graphical user interface (GUI). While there is not a designated module for the controller, control happens through the Tkinter main window in the Display class, where events such as key and button presses are bound to functions in the InstructorInteractionModel class.
Generally, each source code file implements one class. The key_sequence.py and random_distribution_verification.py files implement the Random Distribution Verification mode, and the log_manager.py file implements daily logging and summary file maintenance. The student.py, student_queue.py, and student_roster.py files respectively implement classes for student information, the randomized student cold-calling queue, and the student roster. Finally, the constants.py file stores configurable constants used by the program.

2.1 instructor_interaction_model.py

The instructor_interaction_model.py receives keyboard input from the user, then interacts with other modules to execute the input. The keyboard input includes arrow keys (up, down, left, and right) and import and export roster buttons.

2.1(a) index → int

Used for storing the index across the on-deck display of the currently selected student. For instance, and index of 0 represents the first student from the left.

2.1(b) roster → (new instance of StudentRoster Module)

Used for storing the student roster.

2.1(c) display → (new instance of Cold Call Display Module)

Used for interacting with the display.

2.1(d) queue → (new instance of StudentQueue Module)

Used for storing the student queue.

2.1(e) __init__():

This function is called on startup. First it creates instances of the Key Sequence, Student Roster, Cold Call Display and Student Queue objects. Then it checks to see if there is already a queue file. If there is, it calls load_queue_from_file() on Student Queue Manager, and if there is not it checks to see if there is a roster file. If there is no roster file, import_roster() is self-called. The queue is then filled with queue_from_roster(Roster).

2.1(f) ensure_directories_exist(self):

This function will try to create directories for student_data and the output files, so that we can create files within these directories later on. If the directories already exist, then the function will accept the error and pass.

2.1(g) initial_loads(self):

This function calls _initial_load_roster to get a roster. It then calls _initial_load_queue(new_roster) to get the queue.

2.1(h) _inital_load_roster(self):

This function will check if there is a roster saved in the internal storage location. If there is no roster saved, then it will prompt the user to import a roster.

2.1(i) _initial_load_queue(self, make_new):

When called, this function will check for a queue saved in the internal storage location. If there is no queue saved, then it will create one with the current roster.

2.1(j) shift_index(self, event):

This is called when the left or right arrow keys are pressed. When the left arrow is pressed, the index variable is reduced by 1 or set to 0, whichever is greater, to ensure it doesn't go off the end of the screen. When the right arrow is pressed, the index variable is increased by 1 or set to the length of the on deck queue, whichever is smaller, to ensure it doesn't go off the end of the screen. draw_main_screen(index) is then called on the Cold Call Display Manager to update the display.

2.1(k) remove(self, event):

Called when either the down or up arrow keys are pressed. This function will get the student that was removed from the on deck queue. When the up arrow is pressed the flag is set to True, and when the down arrow is pressed the flag is set to False. call_on(flag) is called to update the students information. take_off_deck(student) removes the student from on deck. write(student_queue, student, flag) on the Log Manager to create a new summary.txt with the students current information and append the removed student to the daily log file. draw_main_screen(index, get_on_deck()) is called on the Cold Call Display Manager to update the display.

2.1(l) import_roster(self, initial_import=False):

Called when the import button is pressed, this function opens a window that prompts the user to select a file to import from. The selected file is saved to internal data, and the Student Roster Manager is sent the new_data(filename) command with the selected file. If the file is formatted correctly, The Student Roster Manager returns an array of differences between new and old data. These differences are displayed and the user is prompted to continue the importing process.
If, however, the file is not formatted correctly, the InstructorInteractionModel opens a window telling the user that the file was not readable. At user input, it loops back to the start of this function, and opens the file selection window again.
The initial_import parameter changes the message displayed to the user slightly.

2.1(m) export_roster(self):

Called when the export button is pressed, this function opens a window that prompts the user to select a filename and location, and calls export_roster(filename) on the Student Roster Manager.

2.1(n) _format_names(self, students):

This function ensures that the roster file is in alphabetical order by last name and is separated with spaces and commas.

2.2 display.py

The Display object is provided by display.py, and displays and updates the program window, which presents the on deck students and the import and export roster buttons. The InstructorInteractionModel object instantiates the Display object when the CoolCall program launches, and invokes draw_main_screen() on the Display object after each pertinent keystroke.

2.2(a) main_window → (instance of Tk object, provided by tkinter module)

The program window used for displaying on deck students and the import and export roster buttons. Several configuration parameters such as the window color, title, size, and resizability are set by the constructor. Also, importantly, the keys used for navigating the on deck students and for removing a student from being on deck (with/without flag) are bound to main_window by the constructor, using the key configuration from constants.py.

2.2(b) draw_main_screen(index → int, on_deck_students → [Student])

Called from the InstructorInteractionModel object any time the User interacts with the system. Updates the display with the first names of the list of on_deck_students as well as highlights the name of the student at index.

2.2(c) rdv → (instance of RandomDistributionVerification object)

Stores the RandomDistributionVerification object to be used by the Display object for main_window.

2.3 key_sequence.py

The KeySequence object keeps track of key presses from the Random Verification Manager. This is accomplished by storing the key presses in an array and checking the array with the target sequence array (10 left key presses). If the 10 most recent key presses match the target sequence, the KeySequence.check_for_match() method returns True to the Random Verification Manager and starts the random verification process.

2.3(a) target_sequence → [String]:

Array that holds the target sequence of keystrokes.

2.3(b) add_key(key):

Called from the Random Distribution Verification Manager, appends the given key to the array.

2.3(c) check_for_match() → Bool:

Called from the Random Distribution Verification Manager, returns True if the array matches the target sequence.

2.3(d) reset():

Called from the Random Distribution Verification Manager, resets the data in the target_sequence array.

2.4 random_distribution_verification.py

The Random Verification Manager relies on the object KeySequence that stores an array containing user input. Each user input is sent from the Instructor Interaction Model to the Random Verification Manager which appends the key to the KeySequence object. If the key sequence array matches the target sequence, random verification mode is launched, which randomizes the queue by cold calling students 100 times and writing to an output file.

2.4(a) key_sequence → (instance of KeySequence object)

Used for storing the most recent key sequences and allows checking for the target key sequence. (See 2.3 key_sequence.py)

2.4(b) test_queue → (instance of Queue object)

Used for storing a temporary queue to test the Random Distribution Verification Mode.

2.4(c) add_and_check_for_random_verfication(key → int):

Appends new key to Key Sequence object and checks to see if it is equal to the target sequence, if it is, start() is called.

2.4(d) start():

Prompts the user to start the program with a tkinter dialogue box. If the user accepts, calls write(), create_test_queue(), and run()

2.4(e) create_test_queue():

Creates an instance of Student Queue with information from the saved Queue file if it exists. If the Queue file does not exist, a new queue is created from the stored roster.

2.4(f) random_call():

Randomly chooses a student from the queue and calls test_queue.take_off_deck(chosen student). Then calls write_line(student).

2.4(g) run():

Simulates an application restart by calling shuffle_front_and_back() on a test queue 100 times, and each time calling random_call() to simulate calling on a random student 100 times for a total of 10,000 operations.

2.4(h) write_header():

Called on startup, writes a header that indicates what the output file contains and when it was written

2.4(i) write_line():

Writes information about each cold call to the output file.

2.5 log_manager.py

The Log Manager maintains the summary file by overwriting (updating) it after every cold call, using the write() method, and records cold calls to the daily logs by appending each cold call to the current daily log file (creating a daily log file as necessary).

2.5(a) write(students, called_student, flagged):

Each time a student is cold called, write(students, called_student, flag) is called from the InstructorInteractionModel object. This function will overwrite the previous summary file with updated student data from the given student queue. The summary file is in the format of <total times called><number of flags><first name><last name><UO ID><email address> <phonetic spelling><reveal code><list of dates> with a tab between each field and a linefeed at the end of each line. write_logfile(called_student, flagged) is then called to record the student that was cold called to the daily log file.

2.5(b) write_logfile(student, flagged):

Appends a line of the following form <response_code><tab><first name><last name>"<"<email address>">" to the daily log file, creating the file as necessary. If the flag argument is false, the <response_code> field is blank, and if the flag is true, then the <response_code> field is the character "X" to indicate that the cold call is flagged.

2.6 student.py

The Student object stores student data, including student identification information as provided by the user. The Student object also stores an integer to keep track of how many times a student has been called on, and an array which stores the dates the student has been called on. The StudentRosterManager and StudentQueueManger objects create instances of the student object.

2.6(a) Strings: first_name, last_name, UO_ID, email_address, phonetic_spelling

2.6(b) reveal_code → String :

String that is used by include_on_deck() to determine if the student should be included in the queue.

2.6(c) total_num_flags → Integer:

Is incremented by call_on() function if boolean flagged = True. This integer keeps track of how many times the student has been flagged

2.6(d) dates_called → [Date]:

Every time the call_on() function is called, the current date is appended. This date is received from the datetime module.

2.6(e) call_on(flag):

Called from the Student Queue Manager, indicates the instructor has called on the student. If flag is true, the total_num_flags integer is incremented. The current date is appended to dates_called

2.6(f) get_name():

Returns a string of the first and last name of the student.

2.6(g) include_on_deck():

Called from the Student Queue Manager, returns true if the student's reveal_code is equal to "0". This indicates the student should be included in the queue

2.6(h) __members(self):

The __members() function returns the students first name, last name, email address, phonetic spelling, and reveal code. It doesn't return the total number of flags or dates called on because they are mutable, or the UO ID per project specification.

2.6(i) __hash__(self):

This function creates a unique hash of the student, which allows the students to be stored in a set.

2.6(j) __eq__(self, other):

This function checks to see if there is a duplicate student by checking if the two student objects have the same first and last name, email address, phonetic spelling, reveal code, and UO ID.

2.7 student_queue.py

The StudentQueue object provides the functionality of creating and storing a queue of student.Student objects, either from a previously saved (pickled) queue file or from an instance of the student_roster.StudentRoster object. The StudentQueue object manages adding and dropping students from the on deck list and maintains queue randomization.

2.7(a) student_queue [array of student objects]

Used for storing the most recent student queue.

2.7(b) queue_from_roster(roster → Roster):

Called from Instructor Interaction Model if there is no save file, generates queue from the given roster module

2.7(c) save_queue_to_file(file_name):

Self-called from take_off_deck(), saves the entire queue to a pickle file.

2.7(d) load_queue_from_file(file_name):

Reads and stores array of student objects to student_queue from pickle file file_name

2.7(e) get_on_deck():

Returns the students who are currently on deck.

2.7(f) shuffle_queue():

Randomizes the order of all elements in the Queue

2.7(g) shuffle_front_and_back():

Called on startup, randomizes the front and back of the queue separately

2.7(h) randomized_enqueue(student → Student):

Called from take_off_deck(), enqueues a student back into the queue at a random index in the bottom N percent

2.7(i) dequeue_student():

Called from take_off_deck() , removes a student from the queue

2.7(j) take_off_deck(student → Student):

Called from the Instructor Interaction Model, if a name has been cold-called, the queue removes the selected name and reinserts it randomly in the bottom "n" percent of names by calling dequeue() and randomized_enqueue(). Finally, save_queue_to_file() is called to store the updated queue.

2.8 student_roster.py

The StudentRoster object collects student information from the imported roster file, then creates and sends arrays of student objects to the Student Queue module.

2.8(a) students: [array of Student Objects]

Used for storing the most recent student roster.

2.8(b) import_roster_from_file(filename):

Called from the Instructor Interaction Model, after checking the file format with get_errors(), saves an array of student objects created from parsing information in the filename to students.

2.8(c) export_roster_to_file(filename):

Called from the Instructor Interaction Model when the user presses the export button, creates a new file and writes data from the current roster to a new file "[filename]".

2.8(d) compare(roster, other_roster):

Called from the Instructor Interaction Model when the user wants to upload a new roster, returns an array of Student objects containing changes that will be made to the old roster.

2.8(e) save_internally():

Writes the contents of the roster to an internal file.

2.8(f) _get_path_name(self, directory):

This function gets the path to a filename to save the student roster file to. _get_path_name(directory) checks to see if the roster file already exists in the given path. If it does exist, it will create a new file called roster<i> in the directory where i starts at the smallest possible natural number.

2.8(g) add_student(self, student):

Adds the specified Student object to the student roster.

2.8(h) remove_student(self, student):

Removes the specified Student object from the student roster.

2.8(i) num_students(self):

Returns the number of students that are currently in the roster.

2.8(f) get_errors():

Called from import_roster_from_file(), verifies that the provided input file is in the correct format

2.9 constants.py

The constants.py file stores constants and variables used widely throughout the program. Having a constants file allows the user to easily change any constant in one location instead of in multiple places. For each of the constants and variables below, the default value is shown.

2.9(a) NUM_ON_DECK = 4

The number of students the user would like to have on deck. This value can be changed to a positive integer up to 10.

2.9(b) INSERT_DELAY = 0.35

Specifies the percentage of the front of the student queue which will be avoided when inserting cold called students back into the queue, to ensure that there will be a delay between when a given student is cold called. This can be a real value between 0 and 1.

2.9(c) ROSTER_DELIMITER = "\t"

Allows the user to change how text fields in the roster file should be delimited (separated). This can be any character that is not present in any of the data fields.

The key values for moving left and right, and removing with or without flags, can be any valid Tkinter keysym value, as described in the manual here: https://www.tcl.tk/man/tcl8.4/TkCmd/keysyms.html.

2.9(d) MOVE_LEFT_KEY = "Left" # Left arrow

Allows the user to specify which key should be used for moving the selection to the left in the on deck student display.

2.9(e) MOVE_RIGHT_KEY = "Right" # Right arrow

Allows the user to specify which key should be used for moving the selection to the right in the on deck student display.

2.9(f) REMOVE_WITH_FLAG_KEY = "Up" # Up arrow

Allows the user to specify which key should be used for removing the currently selected student from the on deck display, with flagging them.

2.9(g) REMOVE_WITHOUT_FLAG_KEY = "Down" # Down arrow

Allows the user to specify which key should be used for removing the currently selected student from the on deck display, without flagging them.

2.9(h) LOGS_LOCATION = (os.path.join(os.path.dirname(__file__), "../logs"))

The path location where log files will be stored. Can be any valid file path.

2.9(i) DAILY_LOG_HEADING = "Daily Log File for Cold Call Assist program."

The heading used for daily log files. Can be any string.

2.9(j) DAILY_LOG_FILE_NAME_PREFIX = "daily_log"

The file name prefix used when creating daily log files. Can be any string of characters allowable in a file name.

2.9(k) INTERNAL_ROSTER_LOCATION = (os.path.join(os.path.dirname(__file__), "student_data/roster.txt"))

The location in which the program will store the internal (imported/loaded) student roster between program runs (persistence). Can be any valid file path.

2.9(l) INTERNAL_QUEUE_LOCATION = (os.path.join(os.path.dirname(__file__), "student_data/student_queue"))

The location in which the program will store the student queue between program runs (persistence). Can be any valid file path.

3. Outstanding Issues

- On the initial import, there is no button to quit the program. The only way to quit the program is to type “command q”.
- The program does not always automatically work on top of full screen. The CoolCall screen has to be dragged to the full screen when using specific applications, like Google Slides.

4. Conclusion

This documentation is intended to provide a programmer with the necessary information for modifying, extending, and refactoring the CoolCall program. Additional minor details are present within the source code, in the form of comments and doctrings. The source code comments together with this document should provide a clear overall structure of the program source code.

References

[1] https://classes.cs.uoregon.edu/10W/cis422/Handouts/EyeDrawDocumentation.pdf
[2] https://classes.cs.uoregon.edu/22W/cis422/Handouts/CALCDOC_World_Tax_Planner.TXT