3K04

Assignment 1

Documentation for DCM

John Wyeth 400171677

Ethan McAuliffe 400179744

Sawoud Al-los 400188404

Kevin(zhenyu) Zhang 400176017

Andrew Faulkner 400193955

Dr. Bokhari

1 November 2020

**Table of Contents**

[**A brief overview of the DCM**](#_f7p760zjeq) **3**

[**Application of the DCM**](#_n4rup9whsac2) **3**

[**Module interaction Diagram**](#_5jjlet8qk1rr) **3**

[**Likely Requirement Changes**](#_9f88etp54thb) **4**

[**Likely Design Decisions Changes**](#_dy5wvxapwqtm) **5**

[**In Depth Module Explanations**](#_vz1zj16ez52v) **6**

[Part1.py](#_207w4fhy2y9o) 6

[Purpose of the module](#_5izgqjbj3mrj) 6

[Secrets of the module](#_f42homz140qp) 6

[List public function (& vi, vii)](#_z8rypq3teqd) 6

[Black-box behaviour](#_al1v3c2ifj9p) 7

[Global variables](#_6kgdekvab07h) 9

[Private functions](#_4z3zrac7sgbp) 9

[Internal behaviour of each public and private function](#_ufkhe8gg0l92) 9

[Testing video](#_2wi9jdk81j3e) 9

[My.kv](#_gz7nc4psjfoh) 10

[Purpose of the module](#_yabtrpe8ssmv) 10

[Secrets of the module](#_p5ibpermld8r) 10

[An explanation of format and code](#_rnz2vukqw5q2) 10

[Database.py](#_v8h843d2cn4b) 11

[Purpose of the module (& ii)](#_xena5pchx9oz) 11

[Secrets of the module](#_gxwvp3kuopf5) 11

[List public function](#_xvxb7910zi71) 11

[Black-box behaviour](#_2tul7o72y7bk) 12

[Global variables](#_p7luy6eu04un) 13

[Private functions](#_y5ufo22b6kc0) 13

[Internal behaviour of each public and private function](#_vi8nqilhdnh3) 13

[Testing](#_i484e0jq6t7o) 13

[Requirement implementation](#_wyickeweb7qc) 14

[Part2.py](#_x1syxcckscd6) 15

[Purpose of the module](#_jiplx11pq148) 15

[Secrets of the module](#_ynls34xhqcj4) 15

[List public function](#_ebkkhcmc3dg9) 15

[Black-box behaviour](#_gmeimk9dy4zp) 17

[Global variables](#_rcjzbsj6fp96) 17

[Private functions](#_fu423ab5xcn5) 18

[Internal behaviour of each public and private function](#_ppaks5wv8u3h) 18

[Testing Table](#_gea6lhunyotn) 19

[**Highlight of Requirements implementation**](#_vrd7eall53l4) **22**

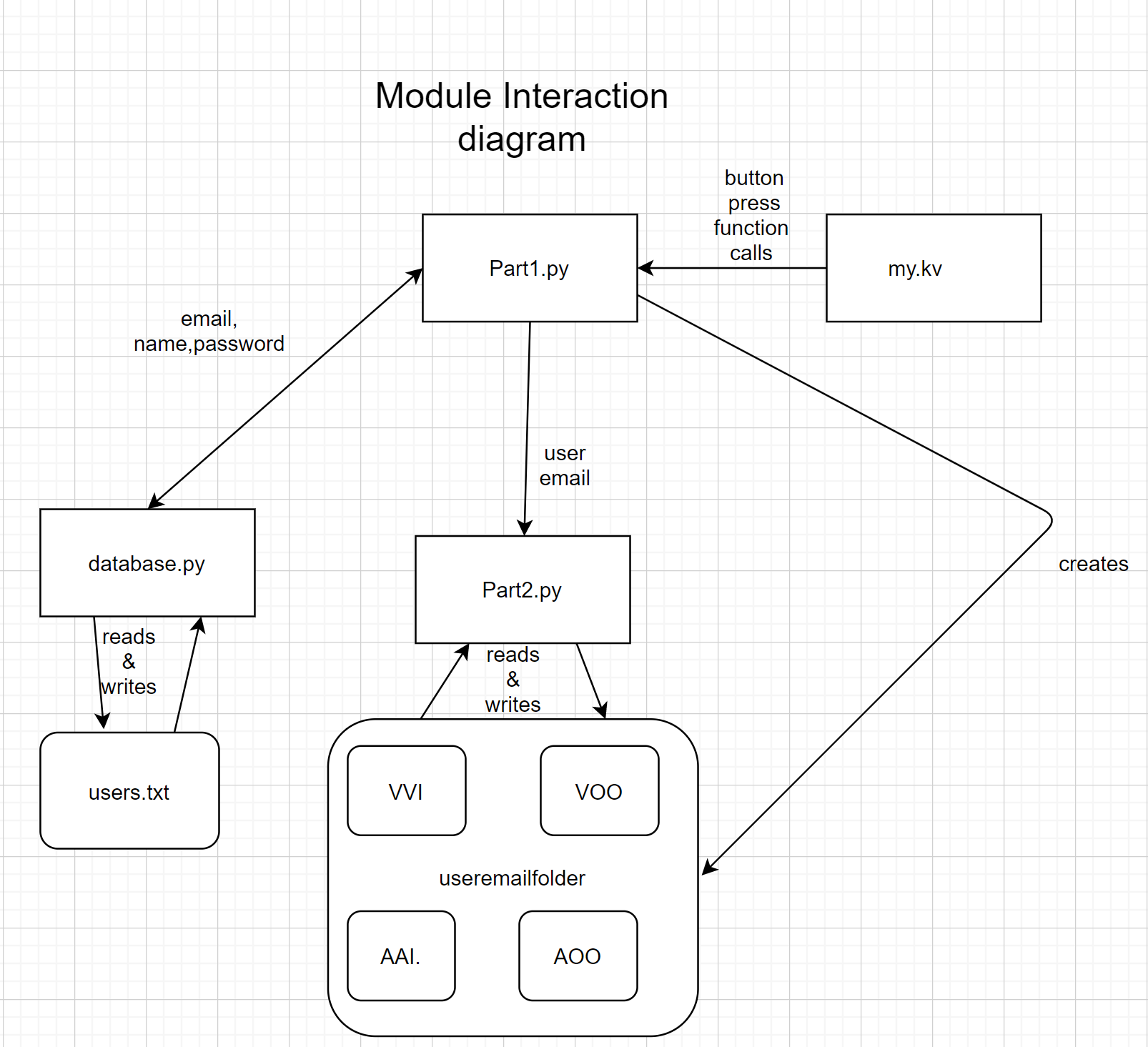
# A brief overview of the DCM

The data collection and monitoring is done using a graphical user interface which has been implemented using Python and it’s Kivy library. The GUI allows the user to store up to 10 accounts. These accounts store variable data for 4 pacemaker modes. Once the user has successfully logged in using their email and password, the user can view and edit the values of these variables. The user can also see if the DCM is connected to the pacemaker or not, this is done using pyserial and checking the specific pacemaker port . The GUI’s code is stored in 4 modules: part1.py, part2.py, database.py, and my.kv. Module part1.py handles creating, verifying and deleting accounts and does this with the help of the database.py module. The database.py reads and writes to a text file which stores all of the accounts login information. The part1.py module also creates each folder for each user which contains all of their pacemaker mode data. The my.kv contains information for the formatting for the screens used to display the information in part1.py. Part.2py handles the screens allowing for the user to view and change inputs relating to the different pacemaker modes.

# Application of the DCM

This GUI first welcomes the user and displays how many accounts are stored in the database, after clicking “start” the user is brought to the login screen and can enter their email and password and press “login” to login and select which pacemaker mode they want to see/edit. They can also select either “create an account” or “delete an account” to be brought to a separate screen which they can enter a name, email, and password into to delete or create an account. Once a user has created an account they can return to the login screen and login. Once they have chosen one of the four modes they can click “values” to view a list of the parameters. They can also either type characters or numbers into text boxes beside the variable names and then press the corresponding “change” to change the variable stored. At the top of the screen there are buttons to navigate to the other modes and at the bottom right hand side of the screen the user can see if the pacemaker is connected or not.

# Module interaction Diagram



# Likely Requirement Changes

* The limit for how many users may use the device might increase to accommodate more patients
* The number of modes must increases to cover more pacing modes such as DDDR, AOOR, VOOR, VVIR and so on
* The ability to enter characters for mode variables may need to be restricted to only numbers as this information is only numerical, and also it should only be limited to a specific range.
* A graph showing the signal from the pacemaker will have to be implemented to show the user how the device is operating
* Ability to sign out without having the user to close out the program

# Likely Design Decisions Changes

* The decision to separate the code into 2 main modules part 1 and part 2 The first of which handles the keeping track of user sign ups, logins, and deletions. The second handles reading from the created user files and writing data to them by navigating the display.
* The decision not to use a .kv for part2 and have the kv library implemented entirely using Python code. Vice versa the decision to use a .kv file for part one to separate the each screen’s formatting from the functions and information stored by that screen.
* Separation of user’s pacemaker mode data from the user’s signup info. Currently there is a text file (users.txt) which stores the name email and password of all users. There are also folders created one for each user that contain the text files for all the different modes of operation that contain the data of each variable. We may want to change the way users store user’s information in general. Specifically in the future we may want to have all the user’s data from each pacemaker mode stored in one file.
* Also the variables are going to be sent to the pacemaker and not just stored in a file.

# 

# In Depth Module Explanations

## Part1.py

### Purpose of the module

The purpose of part1.py is to greet the user and allow them to create up to 10 accounts, delete an existing account and login to an account given the user has entered the correct email and password. This module also passes the email of the user which has logged in to part2.

### Secrets of the module

This module has no secrets.

### List public function (& vi, vii)

* Class CreateAccountWindow(screen): has 3 variables namee,email, and password which are inputs from the my.kv file(handles gui’s textbox inputs). Inherits from the kv’s screen library which allows it to communicate with the .kv and be displayed to the user.
* submit(self): the function that is called when the user submits the information they have entered in the text boxes to create a new account. This function checks to make sure the user entered valid information if they haven’t, the function calls invalidForm(). If they have entered valid information but there are more than 10 users stored calls ToManyAccountsForm(). Otherwise it adds the user’s name email and password to the database. Then it tries to make a folder with the same name as the input email and creates 4 textfiles which correspond to the different pacemaker modes. Calls reset(), then sends the user back to the login screen.
* login(self): Sends the user back to the login screen.
* reset(self): Clears the texts boxes
* Class WelcomeWindow(screen): has 1 variable num which stores the number to be displayed using the my.kv module
* startbutton(self): sends the user to the login screen (sm.current=login)
* on\_enter(self, \*args): function that runs upon entering that screen sets the private num variable to be equal to number of users in the database so it can be displayed to screen using the my.kv module
* Class DeleteAccountWindow(screen): has 2 variables email and password which are inputs from the my.kv file.
* login(self): Sends the user back to the login screen. (sm.current=login)
* reset(self): Clears the texts boxes
* submit(self): uses db.validate to check if the input email and password are in the database. If db.validate finds a user then the user is deleted from the database using db.delete\_user. The folder containing all the user’s pacemaker mode variables is also deleted.
* Class LoginWindow(screen):has 2 variables email and password which are inputs from the my.kv file.
* loginBtn(self): uses db.validate to check if the input email and password are in the database. If db.validate finds a user then the App is closed and the main() from part2 is passed the email entered.
* createBtn(self): sends user to the CreateAccount screen and calls reset (sm.current= create)
* deleteBtn(self): sends user to the DeleteAccount screen and calls reset (sm.current= delete)
* reset(self):Clears the texts boxes
* Class WindowManager(ScreenManager): an object from the kv library which manages the different screens.
* invalidLogin(): displays a pop up saying ‘invalid login’
* invalidForm(): displays a pop up saying ‘invalid info, Email may already have been used’
* ToManyAccountsForm(): displays a pop up saying ‘there are already 10 users stored’
* Class MyMainApp(App): returns the screen manager

### Black-box behaviour

Black box behaviour of functions with more than 1 possible input

|  |  |
| --- | --- |
| function | Class CreateAccountWindow  submit(self) |
| input: | Output: |
| Fields are empty | invalidForm(), sm.current=create |
| Improper email | invalidForm(), sm.current=create |
| Repeated email | invalidForm(), sm.current=create |
| Valid password & email  but db.num\_users() > 10 | ToManyAccountsForm(), sm.current=create |
| Valid password & email &  db.num\_users() < 10 | sm.current=login, self.reset(), folder named “email” created |

|  |  |
| --- | --- |
| function | Class DeleteAccountWindow  submit(self) |
| input: | Output: |
| Invalid password & email | invalidLogin(), sm.current=delete |
| Valid password & email | db.delete\_user(self.email.text)  shutil.rmtree(os.getcwd()+'\\'+self.email.text)  self.reset(), sm.current=delete  (deletes user from database and deletes folder and resets field) |

|  |  |
| --- | --- |
| function | Class LoginWindow  submit(self) |
| input: | Output: |
| Invalid password & email | invalidLogin(), sm.current=login |
| Valid password & email | App.get\_running\_app().stop() (stops app)  main(self.email.text) (runs main program from part2.py) |

### Global variables

Global variables for part1.py

* kv, used to store the value obtain from calling Builder.load\_file("my.kv") which builds the GUI using the .kv file
* Sm, stores an instance of the window manager class. Note the value of sm.current is the screen that the GUI will display
* Db, store an instance of the database class with input “users.txt”
* Screens, an array full of the different screen objects which are added to the screen manager using the kv library’s add\_widget function

### Private functions

See pt. iii

### Internal behaviour of each public and private function

See pt. iii

### Testing video

link to testing video:  
<https://drive.google.com/file/d/1SWIL2nVpoErxbBKJ42qu9MnQyJHpNWJT/view?usp=sharing>

## My.kv

### Purpose of the module

The purpose of this module is to format the windows created in part one. This module defines where display text, input text boxes and buttons go along with their sizes. This module also links variables input from the textbox to variables written in the part1.py as well as functions implemented in the pushbuttons.

### Secrets of the module

I would say the secret of this module is connecting the variables and functions to their corresponding counterparts in part1.py

### An explanation of format and code

The my.kv module is not a module written in python it is written in the kv language which is handled by the kv library. The kv file has an object created for each screen object created in part1.py. These screen objects must have the same name as the class counterpart created in part1.py. Inside these screen kv screen objects the name of the screen must be defined again and variables input from text boxes must be mapped to variables in the main function(part1.py). After this is done you can define the layout using creating a FloatLayout object inside and inside this object you can create more objects such as labels and buttons and define their position and sizes by giving x,y values. Button objects have an on release property which allows you to call a function from python code given you type root.functionname(). Given this information and the information above from part1.py you should be able to understand the code in the my.kv file. To summarize the my.kv works with part1.py by passing it variable information from the user and formatting the actual graphical interface the user sees.

## Database.py

### Purpose of the module (& ii)

The purpose of this module is to create a way to store all the users login information(password, name, email.) in which it is able to be accessed again later even if the program closes later. To do this the database stores the user’s information in a database which saves it’s contents to a text file and loads the dictionary with the contents of that file upon instantiating the database class with the given filename. This entire module is hidden from the user and can only be seen from the text file written that stores all of the data written.

### Secrets of the module

See pt i

### List public function

* Class database:
* \_\_init\_\_(self, filename): stores the name of the file which will be created to store users info. Calls the load function read from the file and creates a dictionary containing all the data from previous users.
* load(self): creates a text file with name filename if there is already a file with this name reads from the data from it and stores it in a dictionary called users
* get\_user(self,email): returns users email if it’s in the database’s dictionary else returns -1
* add\_user(self,email,password,name): adds a user’s info to the dictionary if the email input isn’t already in the dictionary
* delete\_user(self,email): removes the user with the input email from the dictionary and updates the txt file containing the users
* num\_users(self): returns number of users in the dictionary
* validate(self,email, password): confirms the input email and password correspond and are in the database if so returns True otherwise returns false.
* save(self): writes all of the users informations to a text file
* get\_date(): static method to get the data so we know the date the user made the account

### Black-box behaviour

|  |  |
| --- | --- |
| Function: | Database’s \_\_init\_\_(self,filename) & load(self) |
| input | output |
| users.txt | Dictionary with users.txt info |

|  |  |
| --- | --- |
| Function: | get\_user(self,email) |
| input | output |
| faulknea@mcmaster.ca (email in database) | Information related to that dictionary key  Ie |
| faulknea@mcmaster.ca (email not in database) | -1 |

|  |  |
| --- | --- |
| Function: | add\_user(self,email, password, name) |
| input | output |
| faulknea@mcmaster.ca , 123, Andrew (email in database) | 1 (and user’s info is added to dictionary) |
| faulknea@mcmaster.ca (email not in database) | -1 |

|  |  |
| --- | --- |
| Function: | num\_user(self) |
| input | output |
| (10 users stored) | 10 |

|  |  |
| --- | --- |
| Function: | validate(self,email,password) |
| input | output |
| [faulknea@mcmaster.ca](mailto:faulknea@mcmaster.ca),123(correct email and password) | True |
| faulknea@mcmaster.ca,123 (email not in database) | False |
| faulknea@mcmaster.ca,124 (correct email, wrong password) | False |

|  |  |
| --- | --- |
| Function: | save(self) |
| input | output |
|  | Updated txt document |

### Global variables

Global variables for Database class

* Self.filename, name of the file which holds the user’s login information
* Self.users, a dictionary loaded with users login information
* Self.file, holds the open version of self.filename which in writing or reading mode

### Private functions

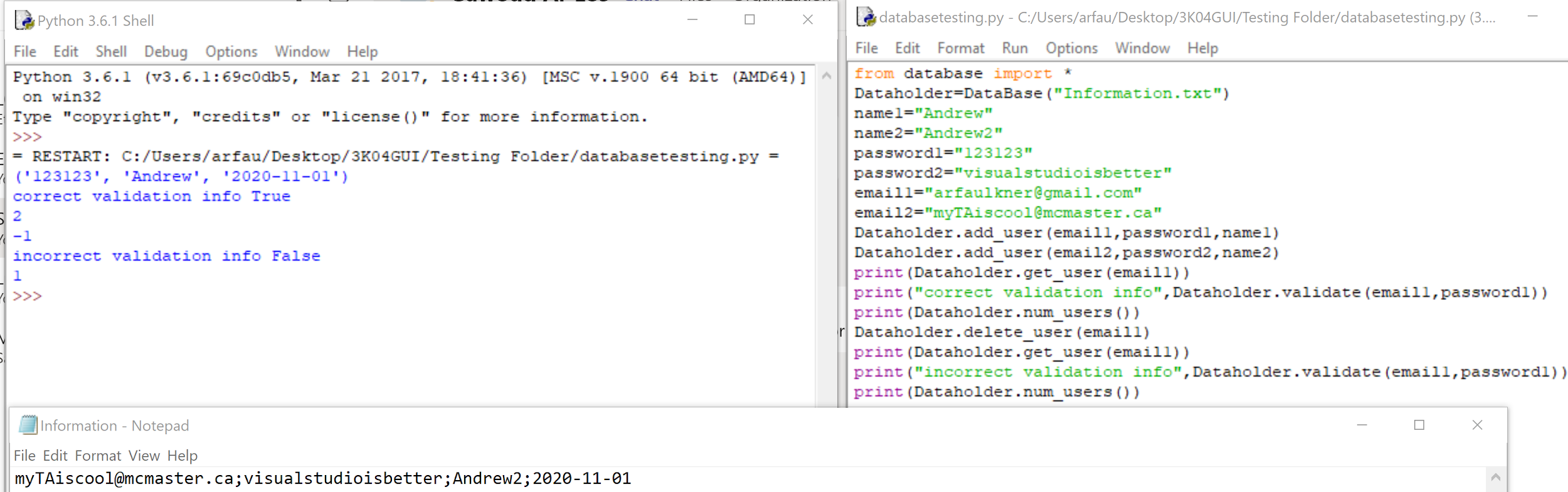
No private functions.

### Internal behaviour of each public and private function

Module is simple to be understood from the explanations above. Further information can be added here when needed

### Testing

Some test cases



### Requirement implementation

## Part2.py

### Purpose of the module

The purpose of this module is to mainly permit the user to change the values of the various modes in the pacemaker. It additionally allows the user to view what values have already been sent to the device and it also shows whether the device is connected or not connected

### Secrets of the module

The list of functions that do not directly contribute in visualization are the following:

* pressed
* connection
* main

And the variables that do not affect the visualization directly are:

* user
* connect
* mode
* var
* File\_loc

### List public function

The used modules are:

* Connect(): this module is used to determine if the pacemaker is connected, it takes no inputs.
* class MyGrid(GridLayout):
* AddButtons(self,x): this module is used to determine which buttons for navigation should be displayed at each screen
* Popup(self,var,temp): is used to show the the value of the parameters at a specific function
* def \_\_init\_\_(self,\*\*kwargs): is used mainly to define the grid structure and display the startup page
* AOO(self,temp): is used to display the parameters for the AOO mode, as well as allowing the user to edit them, also on the lower right corner it shows whether the pacemaker is connected or not.
* VOO(self,temp): is used to display the parameters for the VOO mode, as well as allowing the user to edit them, also on the lower right corner it shows whether the pacemaker is connected or not.
* AAI(self,temp): is used to display the parameters for the AAI mode, as well as allowing the user to edit them, also on the lower right corner it shows whether the pacemaker is connected or not.
* VVI(self,temp): is used to display the parameters for the VVI mode, as well as allowing the user to edit them, also on the lower right corner it shows whether the pacemaker is connected or not.
* pressed(self,instance,mode,var,temp3): is used when one of the change buttons have been pressed, the parameters that it takes are, what mode did the user want to make the change in, what was the parameter the user changed, and what was the value the user wanted to change to. The function opens the folder by using the following path (user\_email\mode\_of\_operation.txt), the function finds the location of the variable that needs to be changed and changes, then closes the file to prevent further changes.
* main(info): is used to run the program while also setting the name of the user variable.

### 

### Black-box behaviour

|  |  |  |
| --- | --- | --- |
| Function | Input | Output |
| AddButtons | Takes self and an integer values | Displays the needed navigation buttons |
| connect | Takes nothing | Returns a 1 or a 0 |
| Popup | Takes the name of the modes | Displays the relevant values |
| AOO | Takes No input | Displays the relevant parameters along with the text field and buttons to alter them |
| VOO | Takes No input | Displays the relevant parameters along with the text field and buttons to alter them |
| AAI | Takes No input | Displays the relevant parameters along with the text field and buttons to alter them |
| VVI | Takes No input | Displays the relevant parameters along with the text field and buttons to alter them |
| pressed | Takes the mode name, the parameter name and the value | Writes the information into the specific file |
| main | Take the name of the user | Runs the program (calls MyGrid.\_\_init\_\_()) |

### Global variables

The variables that are in the scope of all functions are:

* Connection, which is used to determine if the pacemaker is connected, 0 if its not, 1 if it is.
* user, used to store the user email and it is used in other functions.

### Private functions

There are no private functions in this module, all the functions can be accessed from all locations in the code

### Internal behaviour of each public and private function

* AddButtons(self,x)

|  |  |  |
| --- | --- | --- |
| Main Condition | Secondary Condition | Action |
|  | x = 1 | * VOO, AAI, VVI buttons displayed |
|  | x = 2 | * AOO, AAI, VVI buttons displayed |
| AddButtons = True | x = 3 | * AOO,VOO, VVI buttons displayed |
|  | x = 4 | * AOO,VOO, AAI, buttons displayed |

* + Takes an integer value from each of the pacing functions.
  + If statements determine which navigation buttons need to be displayed.
* Popup(self,var,temp)
  + Takes a string value from the function, the string represents the value represents the requested mode
  + It finds the specific file that the user wanted to access
  + Then displays the parameters using the kivy popup functions
* def \_\_init\_\_(self,\*\*kwargs)
  + Takes the \*\*kwargs argument to figure out how to display the the defined layout
* AOO(self,temp), VOO(self,temp), AAI(self,temp), VVI(self,temp)
  + For each of the parameters the function displays the following
  + The Parameter name | Textbox to change the name | a submit button
  + This above occurs for each parameter in its respective function
  + Then at the last level the following is shown
  + “Show [mode] values” | Button to show parameters | “Connected”/ “Not Connected”
* pressed(self,instance,mode,var,temp3)
  + This is by far the most important function in the program, since it changes the values and it works as follows
  + First the location for the file is stored in an variable (user\_email\mode\_of\_operation.txt)
  + Then the file gets open in both read and write

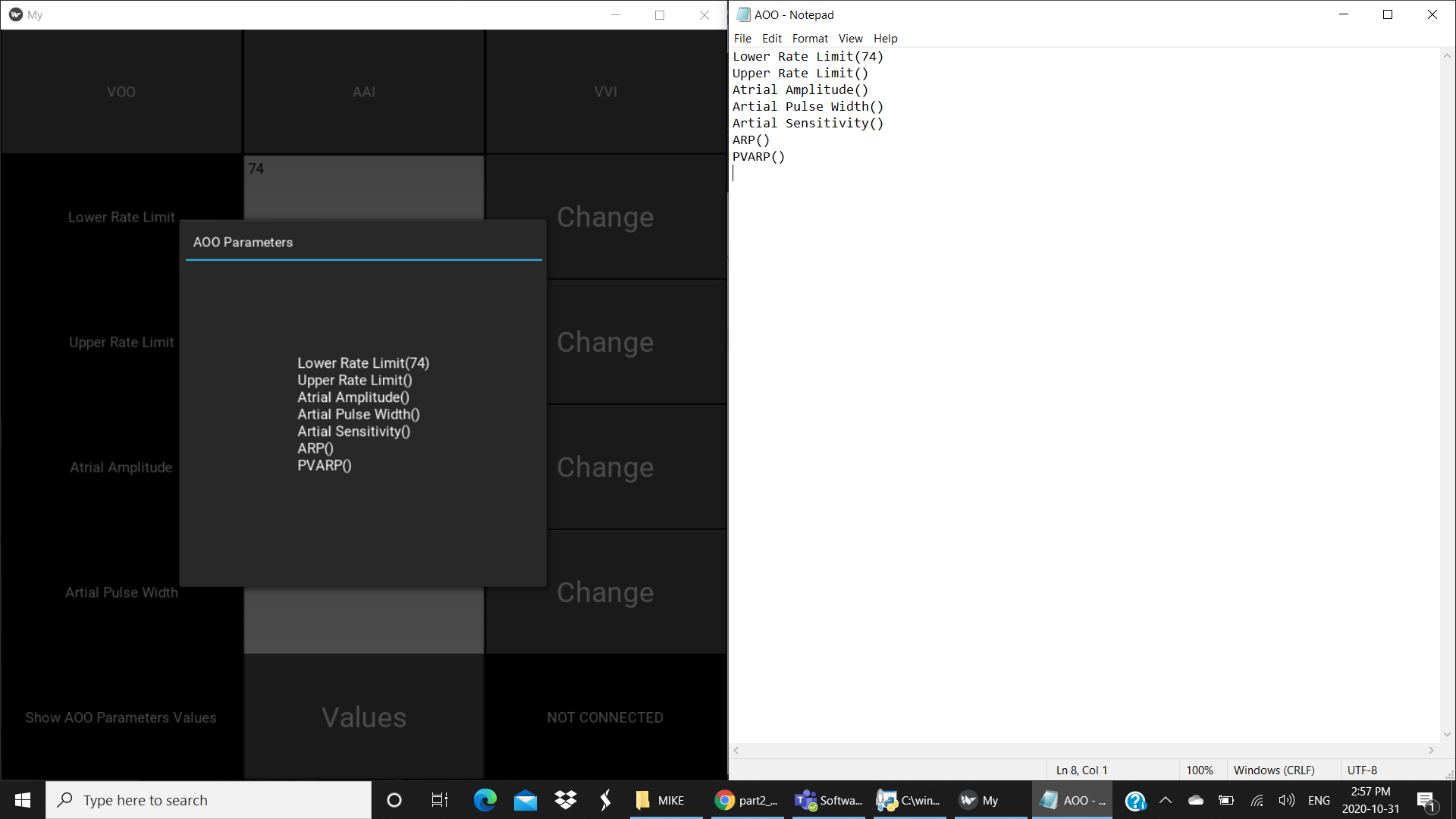
|  |  |  |
| --- | --- | --- |
| Main Condition | Secondary Condition | Action |
| Button\_press = True | var = parameter name | * Value gets inserted between parentheses |
|  | var != parameter name | * For loop gets incremented |

* + The content gets written into the file
  + Finally the file gets closed
* main(info)
  + Takes the email of the user and stores into the global “user” variable
  + Then runs the MyGrid class.

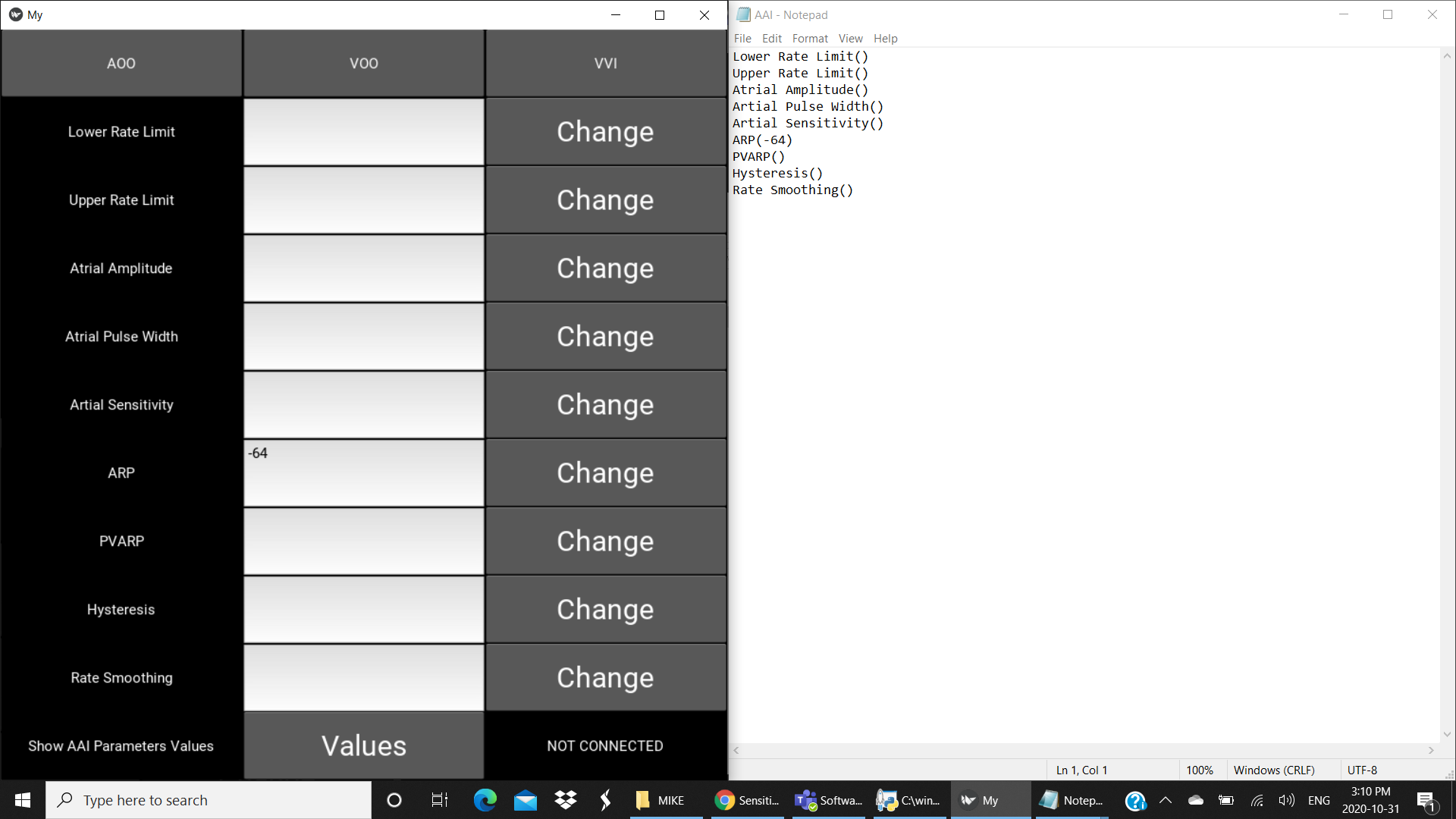
### Testing Table

Here are a few test case scenarios that were run:

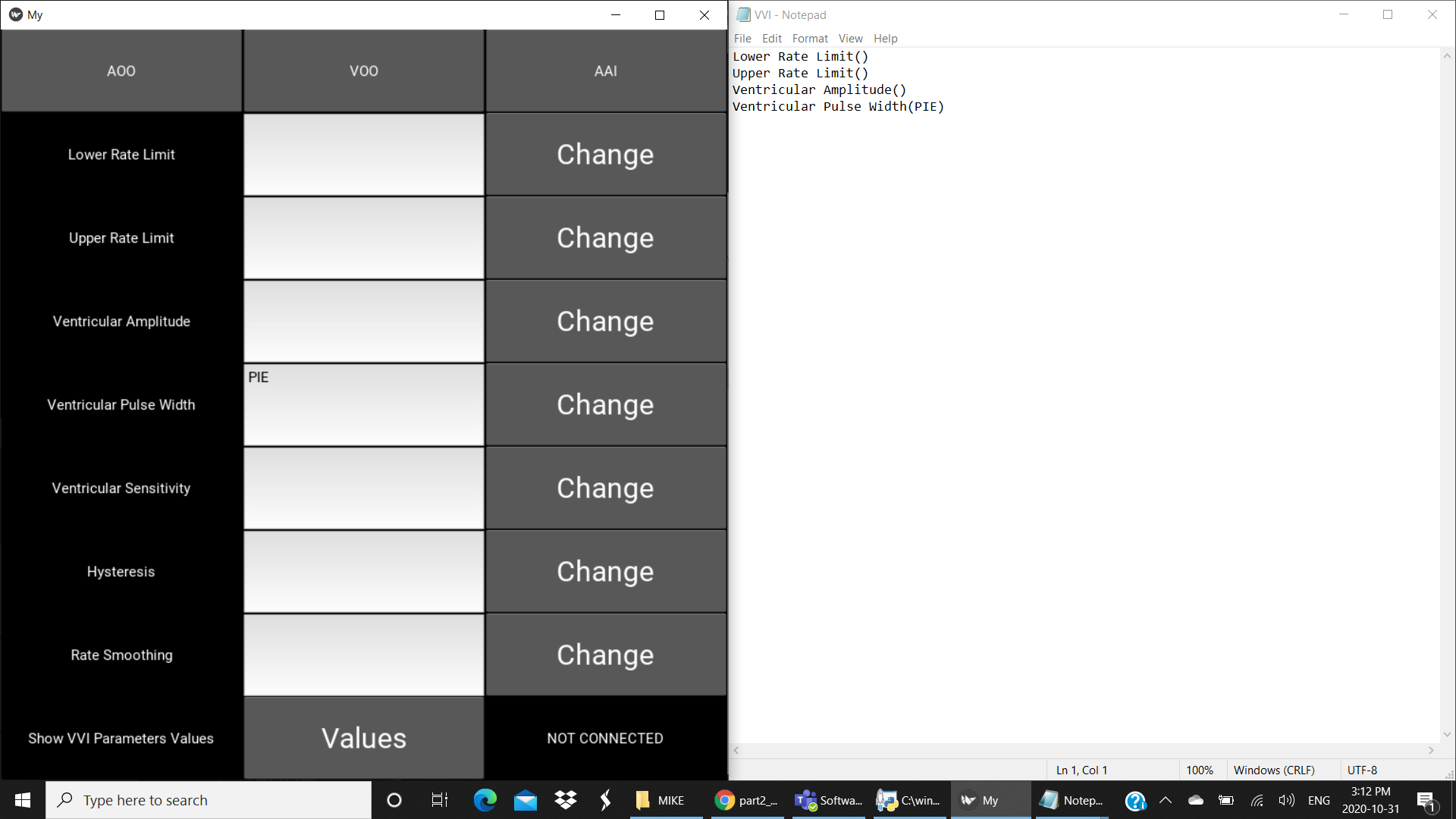
* For the lower rate limit in the AOO mode a value of 74 was inputted:



* For the ARP in the AAI mode a value of -64 was inputted:



* For the Ventricular Pulse width in the VII mode a value of PIE was inputted:



So as the testing shows the program just takes the inputted value and stores it directly into the designated file without filtering, for the future assignments this is going to have to change since there are certain limits on what each value should be.

# Highlight of Requirements implementation

|  |  |
| --- | --- |
| Requirement | Implementation |
| Develop an interface that includes a welcome screen, including the ability to register a new user (name and password),  and to login as an existing user. A maximum of 10 users should be allowed to be stored locally | Database.py  add\_user(self, email, password, name):  validate(self, email, password):  db.num\_users()  Part1.py  class CreateAccountWindow(Screen):  def submit(self):  if db.num\_users()<10  class WelcomeWindow(Screen):: |
| The user interface shall be capable of utilizing and managing windows for  display of text and graphics. | Part1.py  screenmanager() |
| The user interface shall be capable of processing user positioning and input  buttons. | My.kv  root.startbutton() |
| The user interface shall be capable of displaying all programmable param-  eters for review and modification. | Part2.py  AOO(self,temp):  AII(self,temp):  VOO(self,temp):  VVI(self,temp): |
| The user interface shall be capable of visually indicating when the DCM  and the device are communicating | Part2.py  connection():  if(connection == 1): self.add\_widget(Label(text="CONNECTED"))  else:  self.add\_widget(Label(text="NOT CONNECTED")) |