Nicholas DeVilliers

Simple and Secure

Locked away by a mathematically unbreakable encryption algorithm, this program will easily and simply store the user’s username and password for any site.

Tkinter Password Manager - Documentation

Non-examinable Assessment 2018

Contents

[Analysis 4](#_Toc512519527)

[The Problem 4](#_Toc512519528)

[Objectives 5](#_Toc512519529)

[Preliminary objectives/Potential features 5](#_Toc512519530)

[Final objectives 5](#_Toc512519531)

[The End User / Target Market Group 5](#_Toc512519532)

[Prototyping 6](#_Toc512519533)

[Third Party Comments 6](#_Toc512519534)

[Further Research and Development 7](#_Toc512519535)

[1 - Password Rules 7](#_Toc512519536)

[2 - “FileManager” Class 7](#_Toc512519537)

[3 - Breaking down code/creating modules 7](#_Toc512519538)

[4 - Updating GUI Styling 8](#_Toc512519539)

[5 - Copyrighting 8](#_Toc512519540)

[6 - Database Design 8](#_Toc512519541)

[Documented Design 9](#_Toc512519542)

[Critical Path 9](#_Toc512519543)

[Logging in 9](#_Toc512519544)

[Encryption/decryption process – pulling records from database 10](#_Toc512519545)

[Displaying the records / Main GUI Initialisation 12](#_Toc512519546)

[Modules used 14](#_Toc512519547)

[Data Structures 15](#_Toc512519548)

[Dictionaries 15](#_Toc512519549)

[Tuples 16](#_Toc512519550)

[Lists 16](#_Toc512519551)

[File structure 16](#_Toc512519552)

[Storage system 16](#_Toc512519553)

[Text files 17](#_Toc512519554)

[INI Files 17](#_Toc512519555)

[DB Files 17](#_Toc512519556)

[CSS Files 18](#_Toc512519557)

[Structure Diagrams 19](#_Toc512519558)

[Data flow diagrams 20](#_Toc512519559)

[Screenshots 26](#_Toc512519560)

[Testing 32](#_Toc512519561)

[Test 1 – Opening windows 32](#_Toc512519562)

[Test 2 – Basic GUI Functionality 32](#_Toc512519563)

[Test 3 – Exporting to Different File Types (Figure 11) 33](#_Toc512519564)

[Test 4 – Password Rules 34](#_Toc512519565)

[Test 5 – Autosave and login timeout 34](#_Toc512519566)

[Resolution of testing 35](#_Toc512519567)

[Evaluation 35](#_Toc512519568)

[Reflection on Third Party Feedback 35](#_Toc512519569)

[Analysis of third party feedback 35](#_Toc512519570)

[Coding style and PEP8 conformity 35](#_Toc512519571)

[Did I meet my objectives? 36](#_Toc512519572)

[What could have been improved? 36](#_Toc512519573)

[Ideas for new, useful features 36](#_Toc512519574)

[References 37](#_Toc512519575)

[Appendix 1](#_Toc512519576)

[Appendix A – Program Changelog 1](#_Toc512519577)

[Appendix B: Skeleton Program 3](#_Toc512519578)

# Analysis

## The Problem

Social media is currently one of the biggest markets on the planet, especially when young people are considered – 81% of millennials check Twitter at least once per day, and most Instagram users are between the ages of 18 and 29 (McLeod, 2017). Most people who use one social media platform (e.g. Snapchat) will also be users of several others. As an example, I personally have over ten different accounts of various websites over the internet.

Figure 1: Website usage statistics 2017 (Chaffey, 2017)

The primary issue lies with security – in theory, to make your online presence as secure as possible, each account should have a different password. In theory, keeping to this practice would have been easy when people didn’t have many social media accounts, but nowadays, with more than 56% of adults using more than one social media platform (McLeod, 2017), this can become very awkward to manage, resulting in the majority of people using the same password for most of their accounts, on various websites, if not all.

## Objectives

The ideal solution to the security issue would be a password manager – a program that would store a user’s usernames and passwords for multiple websites. This would allow a user to keep all their passwords in one place and look them up when they need to log into a website. Ideally, this program should be password-locked, and the records should not be left in plaintext form.

### Preliminary objectives/Potential features

After some research, most password managers available to download generally have some, if not all the following features:

* Import details from browsers
* Import details from competitors (other managers)
* Two-factor authentication (sending an email or text with a code, or using security questions)
* Export data (to a csv file or similar)
* Automatic password capture (automatically adds new passwords for new websites)
* Automatic password replay
* Fills web forms automatically
* Multiple form-filling identities (email or username, etc.)
* Actionable password strength report (tells user how strong their password is)
* Browser menu of logins (menu to show all details)
* Application passwords (fills in passwords for non-web applications)
* Secure sharing (secure sharing of details)
* Digital legacy (usage statistics, logins, etc.)

As this will be a python 3-based program, some of these features won’t be able to be included (e.g. automatic form filling and data importing), but a lot of these could still be included. Ideally, to make it more secure, it shouldn’t have to connect to the internet at all, except for email functionality (two-factor authentication and password resets).

### Final objectives

In order of importance (most-least), my primary objectives were:

1. Ensure all records are encrypted
2. Use a Vernam cypher and hashing
3. Store records in an SQL database
4. Have the user put in a username and password to be allowed access
5. Allow the user to search for specific details
6. Require two-factor authentication to reset passwords

## The End User / Target Market Group

The program is intended to be marketed towards people who use social media extensively. If you only use one or two online services, then having a password manager isn’t really necessary, since you only have a couple of passwords to remember. This program is made for people who use 10 or more services, and these types of people are becoming increasingly common, especially among teenagers and young adults. A program like this could greatly help to improve account security for this demographic, as a lot of people raise complaints about getting their accounts hacked. Once the password for that account has been figured out, if they use the same password for other accounts, then the hacker can easily get into those.

## Prototyping

Taking from my objective list, my program needs the following basic functions:

* Connection and manipulation of SQL databases
* Tkinter GUI
* Some form of encryption
* E-Mail connection for password resets and two-factor authentication

From this, I created a skeleton program with a few classes and empty methods. I’d already made two Vernam cypher encryption modules (see section E) in the past, so I simply refurbished them and imported them. The program has the “ability” to do the following:

* Write to and read from databases
* Add, modify, and delete records
* Encrypt/decrypt records
* Search for records
* Use a login on the front-end to protect data

These functions are just the basic ones, but there are many more in the finished program. The skeleton program gave me a good ground on which to begin developing the full program, as well as an idea of the critical path of algorithm and data flows. The skeleton program contained only the absolute essentials for my program, so all new features would be developed around this.

As I went along, I made note of any other useful features that could be added. These include:

* Ability to create backups of the records
* Exporting records to different file types
* Built-in password strength checker
* Autosave
* Automatic login timeout

During the development of the program, I continuously made copies of the code as I began to change more features (source control). The final version was 0.6 beta (being the 10th iteration of the program). Doing this allowed me to review my progress as I went along, and always meant I could revert to an older version if I broke anything that I couldn’t seem to fix. I also kept the code together as one single program, since it was far easier to work with in development. This did result in my final beta version being over 3000 lines of code in length, which is excessive for a single module, but this would be broken down into separate modules before the final release. Every time I finished with a version and moved onto the next, I made the old one read-only, ensuring that it would be safeguarded from any accidental changes, so I could move back to it if needed.

## Third Party Comments

Once I’d constructed a second prototype of the program with about 80% of the features I’d implemented, I sat down with my neighbour (a software engineer at Ocado), to see which features I could improve, anything that should be added, and how my code could be improved. He said that I should first include password rules for when I’m creating a new user, to ensure the security of the program at a basic level. He also commented that with my classes FileManager and Security, since Security doesn’t really extend the functionality of FileManager as such, I should instead write FileManager so that it takes an argument for the file name and instantiate it as many times as need be for the purposes of the security class. We discussed exporting the data as CSV files and SQL Databases, and we decided that the functions in FileManager could instead be written as abstract methods and have two classes inherit it – TextFile and CSVFile. They would both be forced to contain the methods to write and read from files, but these methods could be overwritten as required. As well as this, my GUI at that stage was using standard Tk widgets, which have the styling of Windows 2000, which made this program look very dated. He suggested to me to use the Ttk library from Tkinter (Ttk = Themed Tkinter), which updates the styling to the Windows version you’re currently using (Windows 10 in my case). Lastly, my code was nearing 3000 lines, and while this is fine in development, I was advised to break down the code into separate modules for the final release. The changelog for each version is provided as an appendix.

## Further Research and Development

Having been given this feedback, I did some further research to see how I could implement these features. I looked at the following:

### 1 - Password Rules

For the password rules, there wasn’t much research to be done. I just had to create a function that would generate lists of all types of ASCII characters that can be typed – lowercase and uppercase letters, digits, and special symbols – and another function to use these lists to check that a password conforms to the rules I set. I used the ASCII generator function to create a password strength checker tool as well, so that the user could also use my program to check the strength of a new password before changing it on a service or creating a new account on something.

### 2 - “FileManager” Class

Rewriting the FileManager class and creating separate classes to inherit from it was a far easier task than what it seemed. It was easy enough to convert its methods to abstract methods, create new subclasses for each file type, and change the usages. This also meant that as I added different file types to the program, I could just create more subclasses of FileManager, e.g. CSVFile, XMLFile, etc. This can be seen in the class Diagram (Figure 4).

### 3 - Breaking down code/creating modules

I started breaking down my code by moving FileManager and all its subclasses into its own module, which was a fairly simple process, since FileManager wasn’t dependent on anything in the main program. DBManager was also like this, so I just had to remove some references in the doctests in the class, and then move it into its own module. However, the rest of my classes (RecordManager, Security, DeveloperOptions, WindowManager, and Windows) were all co-dependent, meaning they needed to remain in the same program; the result of this was that my final program was still over 2000 lines in length. After much debate, I decided that it would be more beneficial to re-write the GUI as a set of classes, so that I could reduce co-dependency between the back-end and the front-end, despite the amount of time that this would take. To my surprise, this took far less time than I had anticipated, and was able to enhance the user experience during the process by adding in a menu bar at the top of the main screen, as well as showing the user all the records as soon as they logged in. These two features made the old main menu redundant and made using the program much quicker and easier.

I also converted my modules for my Vernam cypher into a single module (section E), rather than having two separate ones for encryption and decryption. They both had their own classes in the new module, so that encryption and decryption would be separate and independent processes. I purposefully designed this new module to be expandable, since it was a standalone module before I used it for this program, and I intend for it to be kept that way. Adding in a new cypher system to this module is simply a process of making two new classes per cypher – one for encryption and one for decryption – and filling these classes with the necessary methods needed to run the cypher.

### 4 - Updating GUI Styling

Restyling my GUI was a fairly simple process at first. I imported Ttk into my program and changed all the widgets to use this. This meant that my program looked a lot more up-to-date when running on Windows; you can see these changes between versions 0.3 and 0.4 (Figure 15). However, during development, I switched over to Ubuntu on my laptop, my secondary development device. When running my program on this, it retained the old Windows 2000 styling, despite not even running on any Windows version. I did some research and found a module named “ttkthemes” by RedFantom (see page 15). I used his “radiance” theme, which was a good match for Ubuntu, to alter the looks of the program when running on Linux (Figure 13). I ran the program on Windows and Ubuntu to confirm compatibility with the added changes, and it ran smoothly on both, looking like native software. However, the menu bar I had implemented (mentioned in the previous section) was not yet supported by Ttk, which unfortunately meant that this widget still retains the old Windows 2000 styling. This isn’t noticeable when running the program on Windows 10, but when running on any Linux distro, it sticks out very obviously, especially since the most popular desktop environments for Linux have their menu bars at the top of the screen, not the top of the window. There is nothing that I can do to change this, unfortunately, and I believe that it remains a case of function-over-form.

### 5 - Copyrighting

I did some research into copyright law and how I should copyright my code. I found some answers on this on Stack Overflow and GitHub – both contained information on how I should go about including copyright statements in my code. The first answer (Shefali, 2010) showed the inclusion of variables at the top of my code: ‘\_\_author\_\_’, ‘\_\_copyright\_\_’, ‘\_\_license\_\_’, and ‘\_\_version\_\_’. I put these in at the top of the main program, and at the top of my cypher module (since this is a standalone module that I made). On the GitHub page was the source code for the setup file of a python module called ‘Virtaal’ (friedelwolff, 2016). I adapted this statement and put it at the top of all my modules, referencing the main program where needed. I also included a full copy of the GNU General Public License version 3.

### 6 - Database Design

Initially, I had a separate database for my keys, passwords, and login records. I was made aware that a database could contain several tables, so I decided to put each table into a single database. This meant that I only needed to have one database connection open at a time, which allowed the program to run faster, and use less memory. It also made storing program data far simpler, and less space-consuming, since there was only one database, which held all the information about each table, as well as metadata. Having three separate databases meant that each database stored metadata and information about the table within, which lead to the program data taking up much more space. Although it only reduced the size of the program by a matter of kilobytes, it was still a necessary and important change, to make the program simpler and faster.

# Documented Design

## Critical Path

The critical path of a program is the set of algorithms that are absolutely required for the program to be launched and run successfully. In an event-driven program like mine, this critical path can be harder to identify, so I have chosen the algorithms that launch the program into its main interface.

### Logging in

The process of logging in is simple, and the process can be seen in Figure 7. On the front end, this is even more simple, as shown in Figure 17 - Logging in. Logically, the process can be broken down into the following steps (going from the normal login window being drawn):

1. Check username
2. Check corresponding password if username is correct
3. If password is correct, re-hash password and close login window
4. Draw main program interface

The whole process is contained by the function log\_in from the WindowManager class in tk\_password\_manager.py. This is a relatively simple function, as shown below:

success, admin = secure.log\_in(password\_entry, un\_entry, win)

if success:

win.destroy()

if self.gui.system == 'Linux':

main = thk.ThemedTk()

main.set\_theme('radiance')

else:

main = Tk()

mainwin = MainWindow(main, admin)

mainwin.create\_table()

main.mainloop()

This function doesn’t do much of the actual work to check the user’s credentials – it just contains the process so that the necessary windows can be destroyed and created (step 3 (partially) and 4). The process handler for logging in is the log\_in function from the Security class in security.py (below).

success = False

admin = False

user = un\_entry.get()

if user == "admin":

valid\_un = True

admin = True

else:

valid\_un = self.check\_username(un\_entry)

if valid\_un:

valid\_pw = self.check\_pass(pass\_entry, admin)

if valid\_pw:

success = True

pass\_text = pass\_entry.get()

bcrypt.hash(pass\_text)

else:

mb.showerror(ERROR\_BOX\_TITLE, "Incorrect password.", parent=win)

else:

mb.showerror(ERROR\_BOX\_TITLE, "Username not recognised.", parent=win)

self.record\_login(user, success)

return success, admin

It is evident in here where step 1 and 2 are located – from line 4 to 8 (inclusive) is where the checking of the username happens – if the username isn’t “admin”, the function for checking usernames checks the given username against that stored in the INI file. If this is correct, it proceeds to check the password – the password checked corresponds to whether or not the user is the admin. If the correct password is given, the password is hashed again and stored in the file, and the Boolean variable success is set to True. As an aside, the login attempt is recorded into the database, regardless of whether or not it was successful. This is a useful security feature, since the user is able to log on and see whether or not someone else has logged into the program, or unsuccessfully attempted to do so.

### Encryption/decryption process – pulling records from database

It should go without saying that encryption is a key aspect of this program’s functionality – it is far more secure to encrypt all of the user’s data as opposed to storing it as plain text. This process is carried out by my cypher module ndv\_cypher.py (Section E)). I had designed this module before I had considered building this program, which greatly helped to speed up the development of the program.

When records are extracted from the database, the following steps happen:

1. Extract records and encryption keys into a list of tuples (where each tuple represents a row, and each place in the tuple corresponds to a column)
2. Run each record/key pair through the decryption algorithm
3. Put decrypted record into a list as a tuple (where each tuple represents a row, and each place in the tuple corresponds to a column)
4. Convert the list into a numbered dictionary beginning at 1 (so that all records correspond to the database indices)

The extraction of the records is fairly simple, and is managed by a simple SQL query in the read\_all\_from\_db function, in dbmanager.py. The function is quite simple, consisting of only 11 lines:

databox = []

keybox = []

all\_records = self.cur.execute("SELECT \* FROM passw\_table")

databox = self.append\_to\_list(databox, all\_records)

all\_keys = self.cur.execute("SELECT \* FROM key\_table")

self.append\_to\_list(keybox, all\_keys)

if all\_keys is None:

keybox = []

else:

keybox = self.append\_to\_list(keybox, all\_keys)

return databox, keybox

The append\_to\_list function is what converts the rows into tuples and puts it all into a list. This makes the data a lot easier to work with. It does this for the keys and for the records so that they can be run through together and sequentially decrypt the records. The data from this function is then returned so that it can be used in the db\_decryption function from recordmanager.py. This function works by processing the data and deciding what action to take, using the variable go\_ahead. The value meanings are as follows:

* 0 = Data missing from passw\_table
* 1 = Decrypt the records as usual
* 2 = Data missing from key\_table

The code for this function is as follows:

records = []

go\_ahead, databox, keybox = self.db\_manager.check\_db()

if go\_ahead == 1:

if len(databox) == len(keybox):

for i in range(len(databox)):

decr\_list = []

data\_record = databox[i]

key\_record = keybox[i]

for x in range(1, 4):

decrypted = ndv\_cypher.VernamDecrypt.decrypt(str(data\_record[x]), key\_record[x])

decr\_list.append(decrypted)

decr\_tuple = (decr\_list[0], decr\_list[1], decr\_list[2])

records.append(decr\_tuple)

elif len(databox) > len(keybox):

mb.showerror("Error: More data than keys")

elif len(databox) < len(keybox):

mb.showerror("Error: More keys than data")

elif go\_ahead == 0:

result = mb.askquestion(ERROR\_BOX\_TITLE,

"Error: passw\_table is empty, but key\_table "

"contains records. Clear 'key\_table'?",

icon="warning")

if result == "yes":

self.db\_manager.clear\_db("key\_table")

mb.showinfo(INFO\_BOX\_TITLE, "'key\_table' cleared.")

else:

mb.showinfo(INFO\_BOX\_TITLE, "'key\_table' not cleared. Program may not function correctly.",

icon="warning")

else:

records = databox

return records

In lines 3, 19, and 28, you can see the different conditions for go\_ahead – focusing on the first condition (go\_ahead == 1), you can see a final check condition that ensures that there are an equal number of keys and records. Beyond this, each record is processed individually and decrypted by the cypher algorithm.

The way a Vernam cypher works by shifting each letter a random amount of places up/down in the alphabet, e.g. “Hello” could be shifted by the key “3 -2 5 6 4”, and it would become “Kcqrs” - each letter corresponds to a number in the key, and that letter is shifted by that amount. In my algorithm, I’ve used ASCII codes as opposed to just the alphabet, so each letter could become a symbol or a numeric character as well. To decrypt the values, it inverts the sign of each key value, and shifts the corresponding characters back by that amount (e.g “2 56 -8” becomes “-2 -56 8”). The key and record tables are kept in the same order, so that each key/value pair always corresponds by the table’s primary key.

Python comes with a build in function ord that changes any given character into an ASCII value, which made it a lot easier to use ASCII to make the shifts. During the decryption process, the records are converted from plain text into ASCII so that the shifts can be performed, and then converted back into plain text so that they can be read by the user.

The decryption algorithm is as follows:

cipher\_list = cls.\_user\_input(cipher)

plain\_list = []

key = str(keytext).split()

for x in range(0, len(key)):

new\_val = chr(int(cipher\_list[x]) - int(key[x]))

plain\_list.append(new\_val)

plain\_text = "".join(plain\_list)

cipher\_list.clear()

plain\_list.clear()

return plain\_text

The function \_user\_input is a private method that converts the input into a list of ASCII values – the ‘input’ is each column in the record. The algorithm then splits the key into its individual values, so that each one can be used to shift back the value (line 3). Then in line 5 you can see the key value being taken away from the ASCII value, and the result being converted back into a char by python’s chr method. Each char is then appended to a list, which is converted back into a string, and then return. After that it’s dealt with by the previously discussed method, db\_decryption. The final step of this process is converting the list into a numbered dictionary (so that the numbers align with the primary keys in the database). This is done by the create\_dict method:

records = self.db\_decryption()

record\_dict = dict(enumerate(records, start=1))

return record\_dict

This is a very simple method, centred around Python’s built-in “enumerate” method. You can see the start parameter is set to 1 so that the first key in the dictionary is 1. This aligns the keys in the dictionary to the primary keys in the database, which keeps all of the data in order.

### Displaying the records / Main GUI Initialisation

Displaying all of the records in the main window proved to be tricky. The best way to do it was to use recursion to print each record in the correct place in the window so that it effectively displayed a table. The code to do this came from the main program is in the MainWindow class in tk\_password\_manager.py, through the method create\_table. Since the source code for this function is 63 lines long, I’ll break it down into sections. The first section retrieves records from the dictionary, according to the search conditions (default is to pull every record from the dictionary).

self.create\_frame()

records\_shown = []

box, indices, searched = manage\_records.search\_dict(self.record\_dict,

self.searching\_text,

self.conditions)

ttk.Label(self.frame, text="Fields searched:").grid(row=0, column=0, sticky=E)

if len(searched) == 1:

searched\_str = searched[0]

elif len(searched) == 2:

searched\_str = "%s, %s" % (searched[0], searched[1])

elif len(searched) == 3:

searched\_str = "All"

else:

searched\_str = "None"

ttk.Label(self.frame, text=searched\_str).grid(row=0, column=1, sticky=W)

As you can see, the function does the following things in this stage:

1. Draws the frame that contains the table
2. Retrieves records from the dictionary as a list
3. Indicates which fields have been searched
4. Shows the search terms, if there are any

After this, the function begins setting up the table:

if len(box) > 0:

ttk.Label(self.frame, text='Showing records').grid(row=0, column=2,

sticky=E)

if len(box) >= self.upper\_bound:

highest = self.upper\_bound

else:

highest = len(box)

ttk.Label(self.frame, text='%d - %d of %d' % (self.lower\_bound + 1, highest,

len(box))).grid(row=0, column=3, sticky=W)

ttk.Label(self.frame, text="Site", font=self.HEADER).grid(row=1, column=1)

ttk.Label(self.frame, text="Username", font=self.HEADER).grid(row=1, column=2)

ttk.Label(self.frame, text="Password", font=self.HEADER).grid(row=1, column=3)

This section draws the table headers if there are any records to display. It also says which records are showing out of all the records returned in the previous stage, e.g. it could be say “Showing records 1 – 10 of 15” if the page limit was set to ten and there are 15 records in total. The next section begins drawing the records; it will either display all of the records, if it does not exceed the user-set page limit, or it will show as many records as will fit on the page:

if len(box) <= self.max\_records\_shown:

for i in range(len(box)):

try:

record = box[i]

except IndexError:

break

index = indices[i]

for x in range(1, 4):

if x % 3 == 0:

wm.edit\_button\_func(self, i, box, index)

ttk.Label(self.frame, text=record[x - 1]).grid(row=i + 2, column=x,

padx=self.DEFAULT\_PAD, sticky=W)

else:

for i in range(self.lower\_bound, self.upper\_bound):

try:

record = box[i]

except IndexError:

break

records\_shown.append(record)

index = indices[i]

for x in range(1, 4):

if x % 3 == 0:

wm.edit\_button\_func(self, i, box, index)

ttk.Label(self.frame, text=record[x - 1]).grid(row=i + 2, column=x, padx=self.DEFAULT\_PAD, sticky=W)

The first part of the if statement is for when the number of records retrieved is less than or equal to the maximum number of records shown at a time. It iterates through the list using a for loop with the variable i, to pick each record. A small try/except block is used to catch any index errors, in which case the loop is broken. Following this, another for loop is used with the variable x to iterate through each field in a record. It also draws a button to edit the record, placed before the first column. Each part of the record is then displayed as a Label, using its index and field to choose where it will be located. Tkinter’s grid method makes this very easy, as the frame can be used as a grid. The variable i defines the row while x defines the column.

The else part is used to draw the records when the number of records retrieved exceeds the maximum number of records shown at a time. It runs similarly to the first for loop in the first part of the if statement, except that it uses self.lower\_bound and self.upper\_bound to iterate through the loop as opposed to the length of the list of records. After this, the rest of this section is identical to the previous, except for the following code added to the end:

if self.upper\_bound < len(box):

ttk.Button(self.frame, text='Next',command=self.next\_page

).grid(row=self.upper\_bound + 2, column=3)

if self.page != 1:

ttk.Button(self.frame, text='Back',command=self.previous\_page

).grid(row=self.upper\_bound + 2, column=0)

if self.max\_records\_shown < len(box):

ttk.Label(self.frame, text='Page').grid(row=self.upper\_bound + 2, column=1,

sticky=E)

self.calculate\_page\_numbers(box)

ttk.Label(self.frame,

text=str("%d/%d" % (self.page, self.no\_of\_pages))

).grid(row=self.upper\_bound + 2, column=2, sticky=W)

This piece of code puts in controls for moving forwards and backwards between pages, based on which page the user is currently on and if there are records still beyond the upper and lower bounds. The number of pages are calculated by dividing the length of retrieved records by the maximum number of records shown, rounded down to the next whole number. This number will be increased by one if the remainder of this division is more than zero.

The final part of this method is a condition that tells the user that there are no records found if no records are retrieved from the dictionary. This can happen if the search terms don’t return any results, or if the database is empty:

else:

self.master.geometry("%dx75" % self.DEFAULT\_WIDTH)

ttk.Label(self.frame, text='No records found.').grid(row=1)

### Modules used

Often, in the development of a higher-level program, the standard modules that are shipped with Python will not satisfy a developer’s requirements, and will need more specialist modules to be able to do what they are trying to do. I have used the following four non-standard modules in my program:

#### Passlib and Bcrypt

When I was designing the security for my program, I looked into different ways of storing passwords. I could encrypt the password, but if the key was taken then someone could decrypt the password and steal the data. Storing as plaintext has only the advantage that the user could be told their password if they forgot it; however, this is a very poor way of storing a password. I then came across the technique of hashing, which generates a string of characters based on an input. Hashing is unlike encryption, in that each input will always give a unique hash, and a hash cannot be un-hashed. I decided that this was the most secure way of storing a password, since the most secure hashes available have not had anyone give two inputs that create identical hashes. It also meant that the user’s password is never known by the program or the developer.

I found two hashing libraries available for python – Hashlib and Passlib. Hashlib is designed for generating hashes used in checksums, whereas passlib was specifically designed for secure password hashing. Passlib has a large variety of hashes available, some of which are more secure than others.When looking into the hashes available, I narrowed them down to the two most secure – “bcrypt” and “SHA256”. I decided on bcrypt since SHA1 was recently cracked by a team at Google, so it was unlikely that other SHA algorithms would be safe for much longer.

#### Ttk themes

I decided on using this library later on in the development when I was implementing Linux support. As mentioned in the Further Research section (page 8), the styling of the program in Ubuntu retained the look of Windows 2000. I found out that this is because Tkinter is simply a python interface for the Tcl interpreter, which is Windows’ way of programming a window. As this is not the system used in any Linux distributions, it was ported over so that it could still be used, hence why it retains Windows styling. Despite Tkinter coming with a Themed Tkinter module, this only fixes the theming to make it look more up to date in newer versions of Windows. In order to make it look more at home in Ubuntu, I had to use Ttk themes by RedFantom; the theme ‘Radiance’ in the package is a perfect match for making the program look nicer in Ubuntu, so this is the one I decided to go with. Although this module has no real effect in the program’s functionality, it helps it look nicer, and a good-looking user interface is an important aspect of any consumer program.

#### Dicttoxml

Building in different data types to export to was a mostly easy process, since a lot of the necessary modules are shipped with python (e.g. CSV, JSON, etc). However, exporting to XML proved to be more of a challenge. Fortunately, I came across a module called “dicttoxml” when I was searching for ways to turn a dictionary into an XML file. However, there is a bug in this module where it will not work if you have integer dictionary keys, like I had. To resolve this, I simply used a dictionary comprehension to convert all of the keys to strings:

content = {str(k): v for k, v in content.items()}

where “content” is the original dictionary containing all of my records. This comprehension iterates through the original dictionary and creates a new one, using the same keys except converting them into strings (str(k)). The module also comes with a function ‘toprettyxml’ which tidies up the XML file with indents and different lines, so that it is more readable (see Figure 21).

## Data Structures

In any high-level program, a variety of data structures are needed for different purposes. For my program, there are three main data structures that I have used for containing my records during runtime – dictionaries, tuples, and lists.

### Dictionaries

A dictionary is the primary way of storing records in my program. I chose this because it allows me to assign a key to each piece of data – this way, I can make sure each key corresponds to the primary key in my database. This makes it much easier to keep track of each record and make sure the encryption keys don’t get mixed up and accidentally corrupt the data. Dictionaries are a direct-access structure, so pulling specific records is easier, which is useful in my case where a user may want to search for a password to a specific service. I use Python’s built-in method enumerate to create my dictionary (see “Encryption/decryption process – pulling records from database” for more information on this algorithm).

### Tuples

Tuples make up each individual record in my dictionary – since each record will always have a fixed length of 3, tuples are much better suited to storing each record. This means that they will always take a fixed amount of memory, and are much faster to create and manipulate. If I were to use a list, these would have to be dynamically created, which takes longer to do and uses more memory, and would reduce the efficiency of my program over all.

### Lists

Lists do have their advantages, however, when I am displaying my records. This is because for any individual search, different numbers of results will be returned, thus it makes a lot more sense for them to be dynamically created. As well as this, when new records are added, they are added first to the dictionary, and then the window is refreshed to reflect these additions – this of course means that the list used to contain the records when they are displayed will increase in length. Tuples cannot increase or decrease in length, and thus would not allow me to properly display my records to the user. Lists are also faster to use than dictionaries when iterating through them, which means less time is taken when drawing the window that displays my records.

## File structure

My program makes use of several different file types, and so do many high level programs. The main file time used is of course a Python (.py) file, since that is where the program source code is stored. It also uses Database (.db) files to store all of the user’s records, Initialization/configuration (.ini) files to store settings, Plaintext (.txt) files to store the email template and copyright information, and a Cascading Style Sheet (CSS) file to store styling information for the HTML export process. Each of these files is discussed below.

### Storage system

First and foremost, it is important to understand the structure of the program’s file-system. It is fairly simple, as shown in the following diagram.

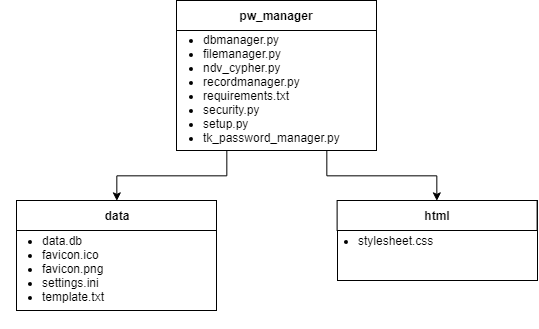


Figure - Program file-system.

The primary folder in the diagram is pw\_manager, from which two other folders stem: “data” (which contains some necessary backend data) and “html” (which contains the stylesheet, and acts as a temporary folder to zip when HTML files are exported).

### Text files

Text files have three usages in my program:

* Program requirements
* Password reset email template
* Copyright window text

Storing this information in text files makes it a lot easier to format, as opposed to storing them as constants in my program. They can be retrieved easily through my TextFile class and used in the program as necessary.

### INI Files

INI files are sequential access, and work similarly to dictionaries, except with a third layer. The file has headers for different sections, which then have key/value pairs below them. Below is a dump from one of the sections in my file “settings.ini”:

[Preferences]

autosave time = 20

timeout active =

timeout = 60

records displayed = 10

logins displayed = 20

This section contains the user preferences for the program. One disadvantage to INI files is that each value is always stored as a string, so when the program reads it into runtime I have to put a step in to convert it back into an integer or a Boolean. Note that the timeout active key is left blank to indicate a False Boolean value. Aside from this one disadvantage, INI files are a very convenient and easy-to-use method of storing settings for a program.

### DB Files

Database (.db) are a standard method of storing database information. They are used widely, and so I decided that this would make the most sense for storing all of my data. The database (data.db) contains three tables – pass\_table (contains the encrypted passwords), key\_table (contains the encryption keys), and log\_table (contains a log of attempted logins). pass\_table and key\_table each contain the same three fields – “Site”, “Username”, and “Password” (these each have “key” on the end in key\_table). log\_table also has three fields – “Date”, “User”, and “Success”. See 6 - Database Design for more information on the development of the database.

### CSS Files

Cascading Style Sheet (CSS) files are the standard way of storing styling information for HTML files. This typically contains information on the colours, margins, fonts, and other styling information. I use this folder to export HTML files into, after which I zip the folder to a location of the user’s choice. I zip it so that the stylesheet stays with the HTML file and the styling isn’t lost. My CSS file contains the following information:

body

{

font-family: arial;

background-color: #FFFFFF;

}

h1

{

font-size:40px;

color:black;

padding-left:10px;

}

table

{

background-color:white;

}

td

{

background-color:lightgray;

}

This results in my HTML table looking like this:

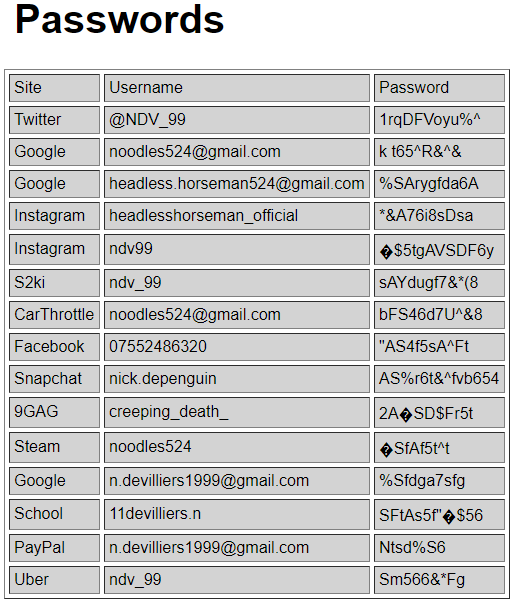


Figure - HTML Table

## Structure Diagrams

The following diagrams show the structure of the program code, and the GUI front end.

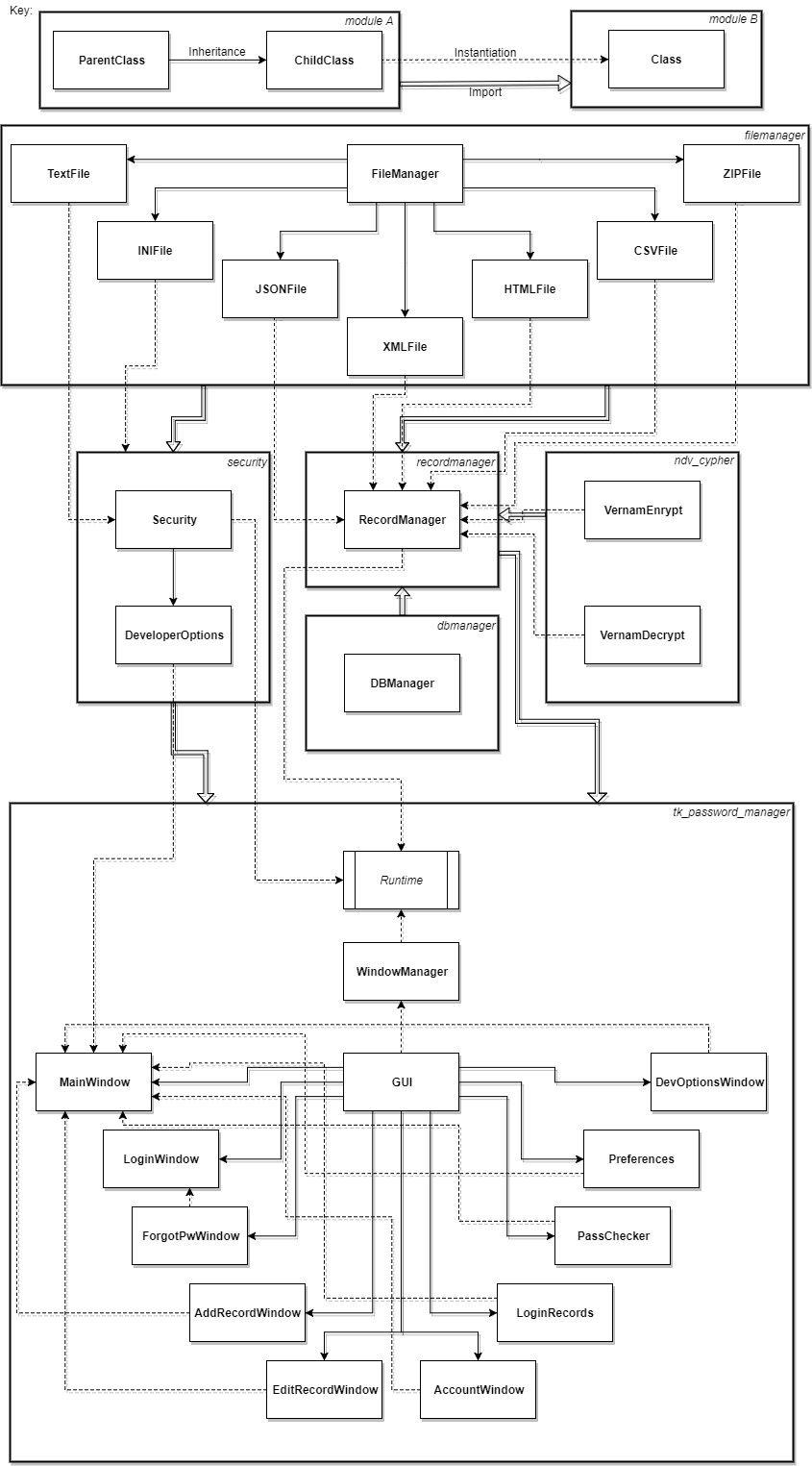


Figure 4 - Class Diagram – Hierarchy of classes

This is a layout of how all the classes and modules interact with each other. The arrows show a flow of data from one class to another, rather than pointing toward parent classes. The diagram also represents the class hierarchy (i.e. which classes are most/least important)

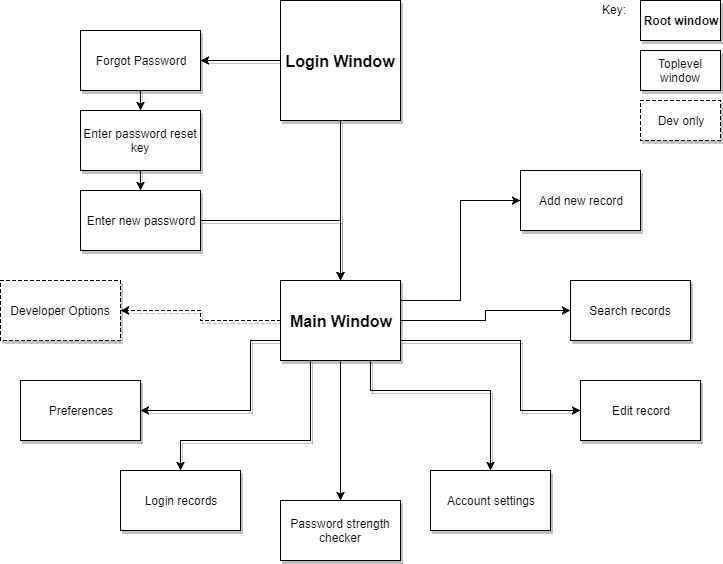


Figure 5 – Windows

The path of accessing all window in the interface. Except for the ‘search records’ window, every window is its own class. For the process of resetting a password, the three windows involved simply change the content of the window and make use of the same window.

## Data flow diagrams

These diagrams demonstrate the flow of data within the program. It should be noted that these diagrams do not represent the highest level of complexity of the operations described, in favour of readability and simplicity. For example, arguments passed to functions for the sake of lifting windows after a message box has been closed have been redacted, as well as some paths showing what happens when an operation is cancelled.

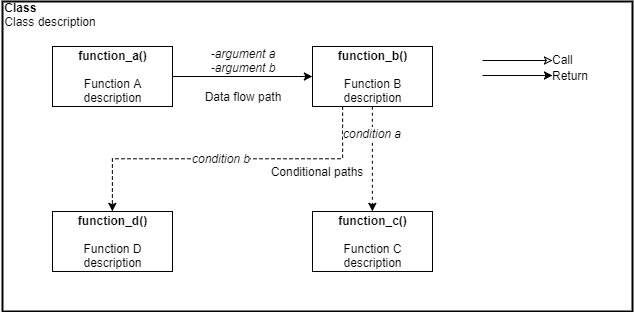


Figure 6 - Key for data flow diagrams

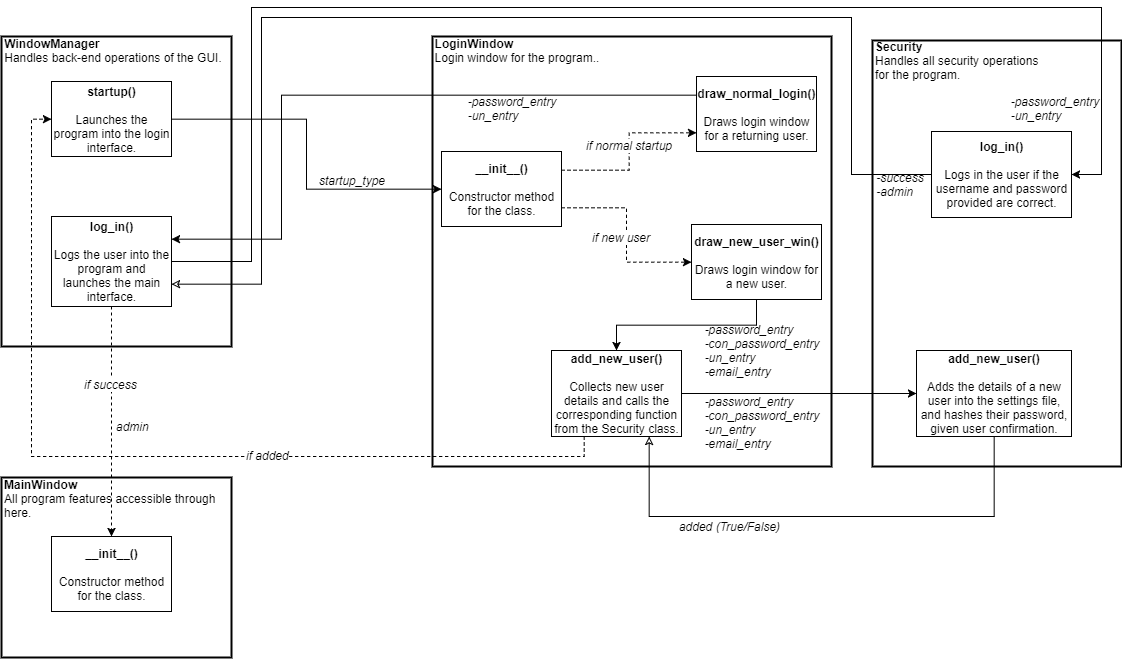


Figure 7 - Logging in

The process handles logins for new users, the main user, and the admin account.

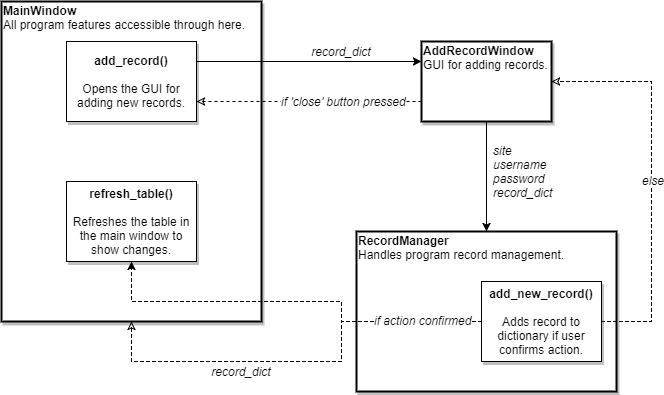


Figure 8 - Adding a new record

In each addition, the argument ‘record\_dict’ is an instance attribute of MainWindow, so every function performed is modifying this attribute in a specific instance, so the class doesn’t need to be recalled every time a record is added.

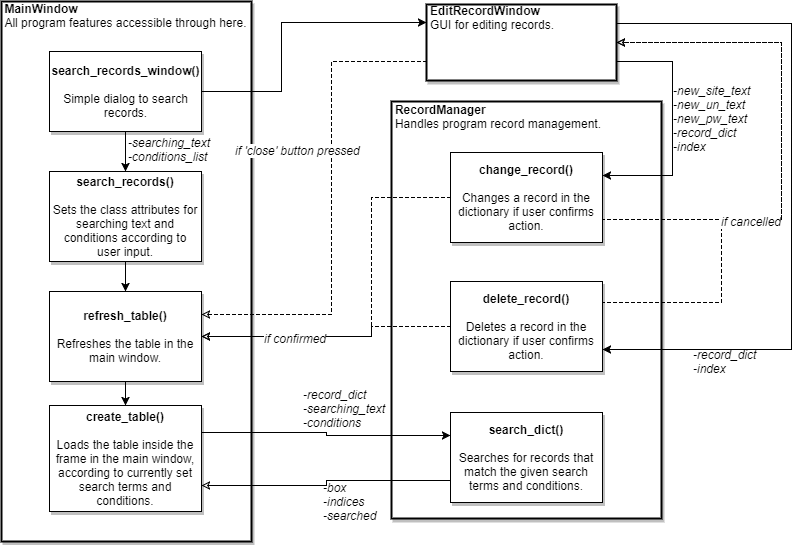


Figure 9 - Editing, and deleting records

The MainWindow class remembers the last set search terms and conditions and refreshes accordingly until changed. Upon opening, these will default to show all records.

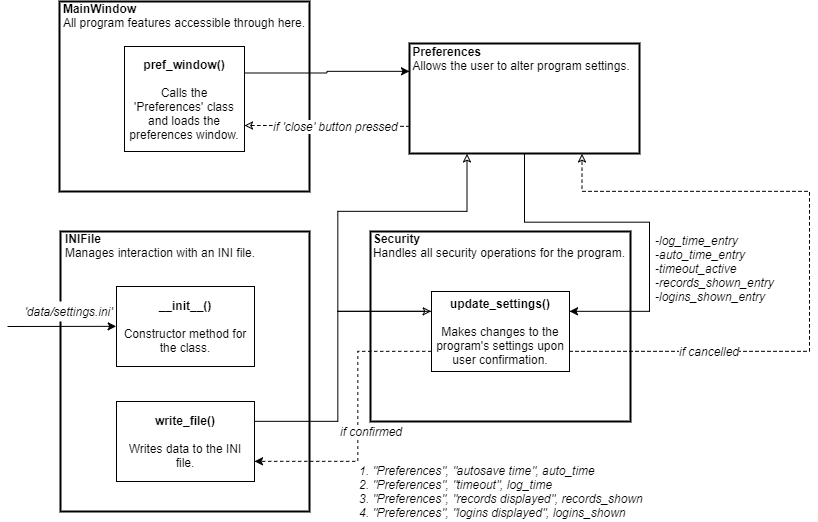


Figure 10 – Altering preferences

For the changes to be applied, the program must be restarted. Each preference is written to the INI file individually (hence the numbering of the arguments, where each numbered item is the next call).

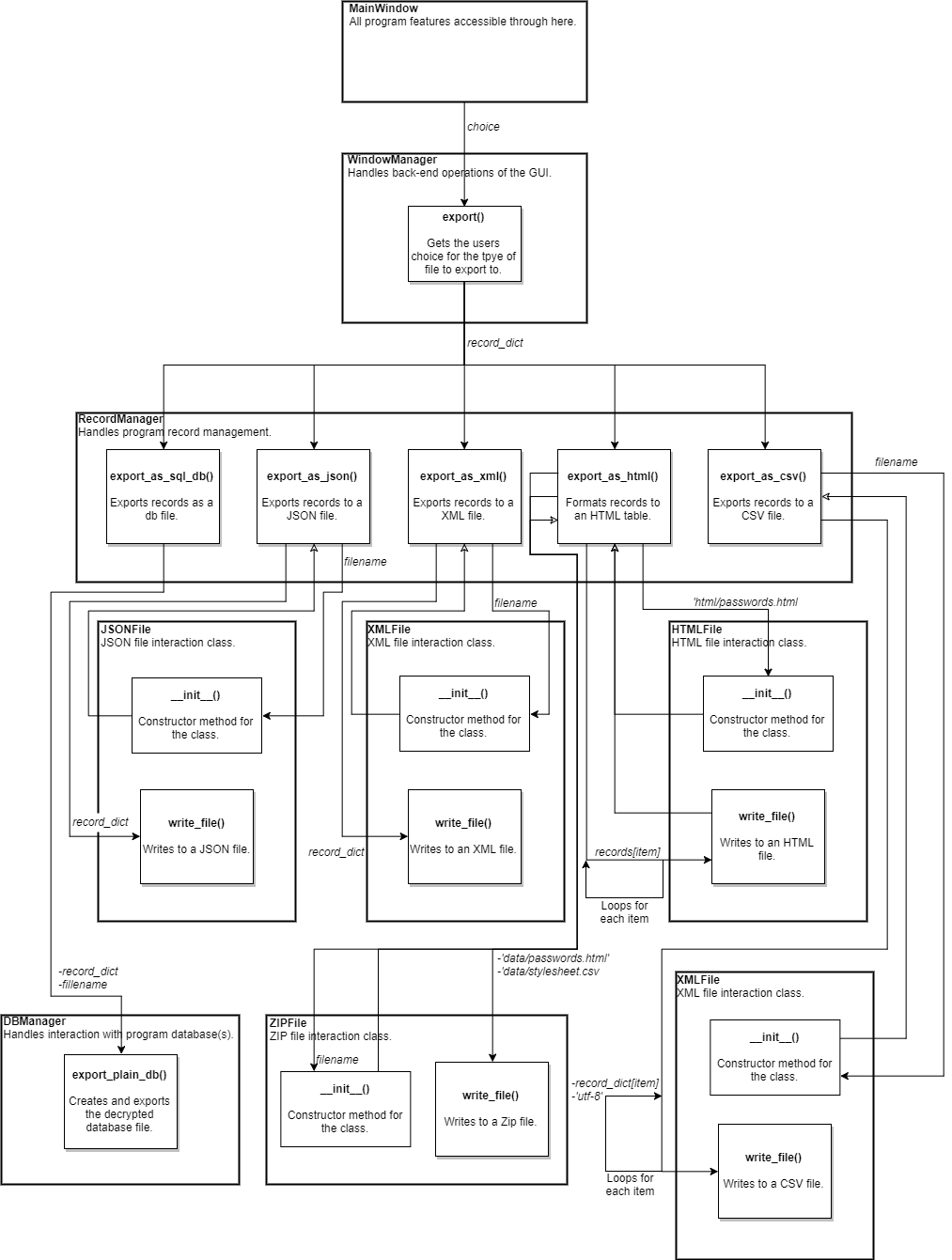


Figure 11 - Exporting the records to different filetypes

All exporting is handled by the WindowManager class, to help eliminate co-dependency between higher and lower level classes. Operation cancellation has not been included in this diagram, for the sake of simplicity and readability, although it is in the program.

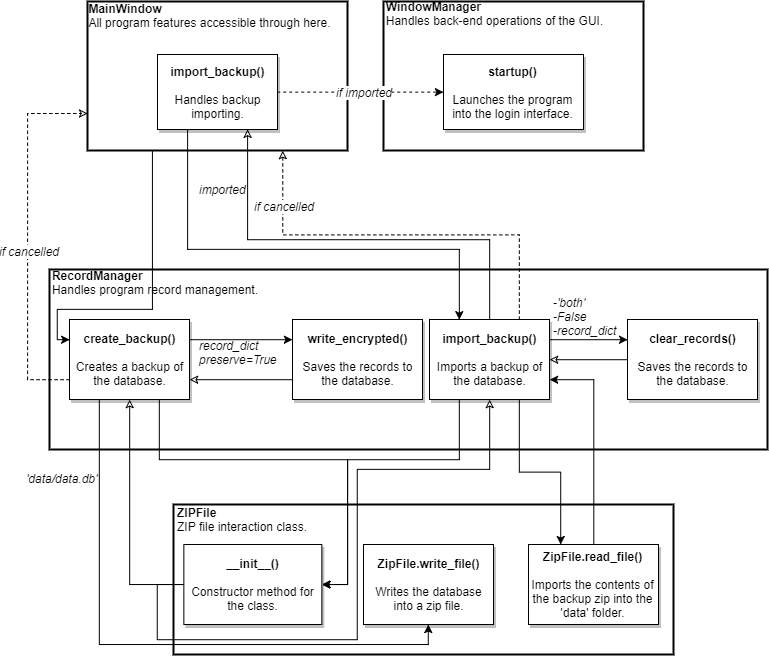


Figure 12 - Creating backups of the data

Importing a backup is handled slightly differently to exporting. The main window waits for the importing process to finish, which returns a Boolean variable ‘imported’. If this is true, the program restarts. On the other hand, the creation of backup doesn’t require the program to be reloaded, so its operation is far simpler.

## Screenshots

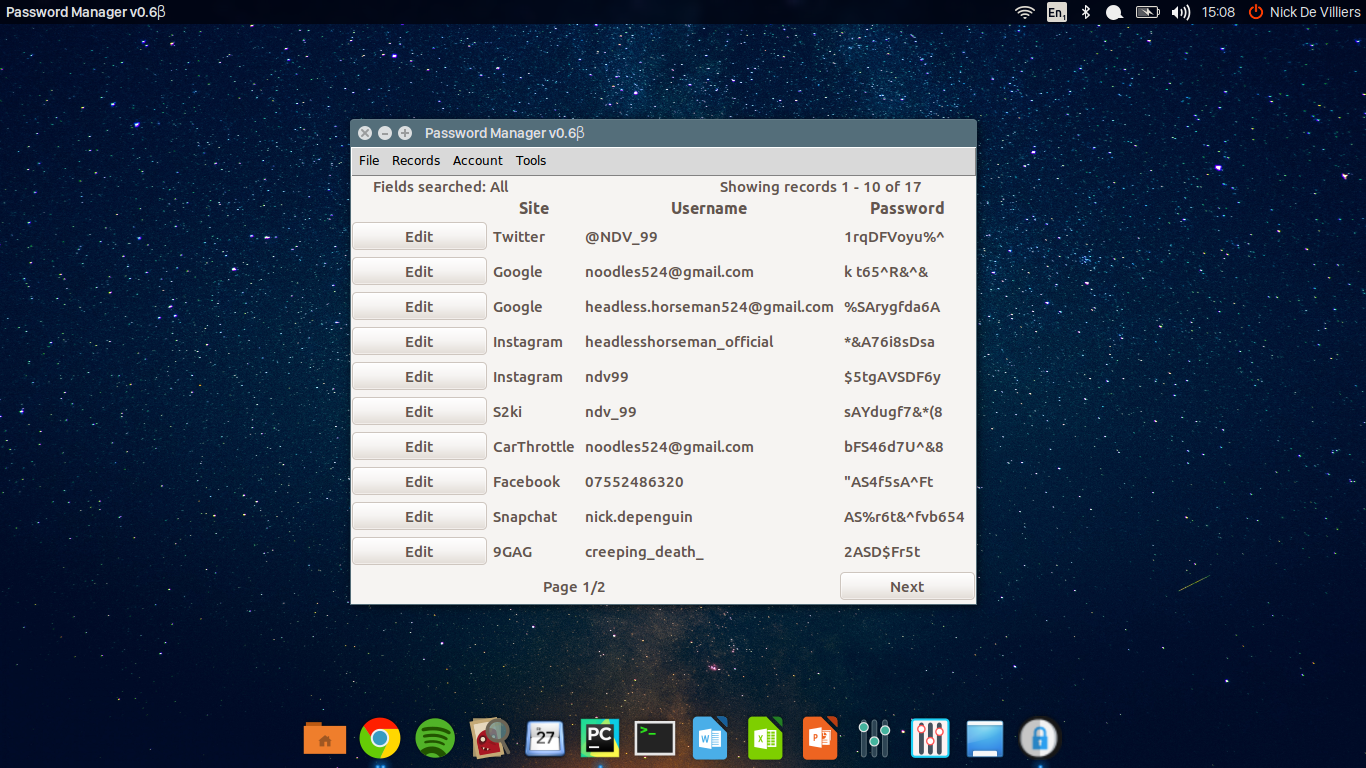


Figure 13 - Program running in Ubuntu

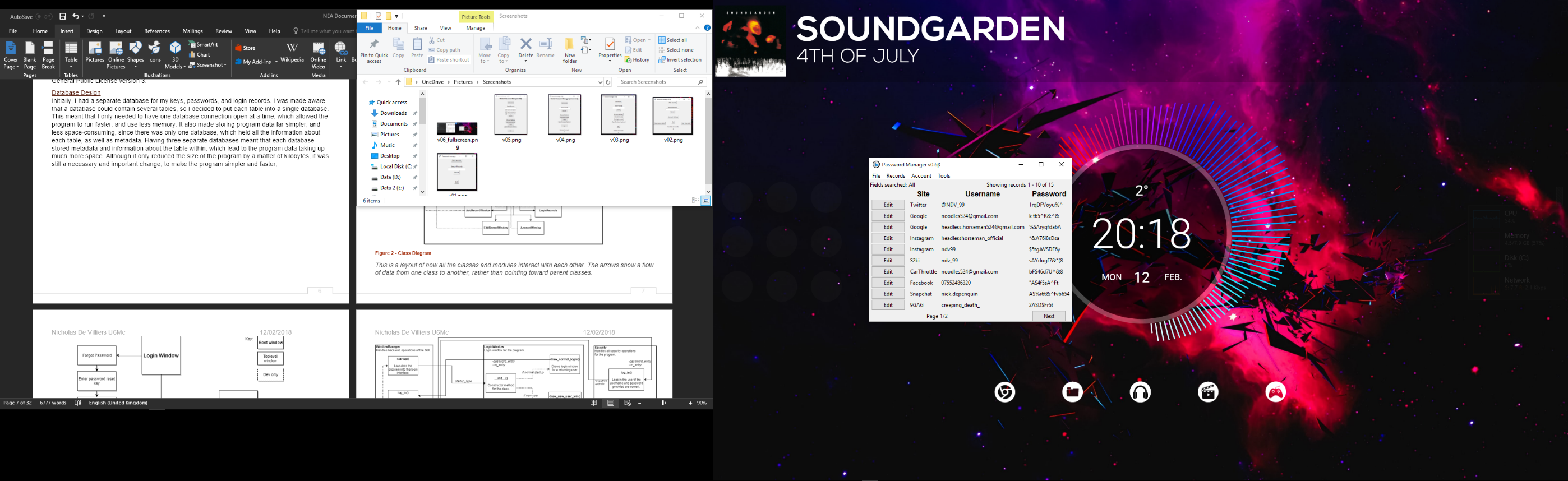


Figure 14 - Program running in Windows 10



Figure - Program evolution over versions

Each window pictured is the main menu screen from each version. This diagram shows how far the program has come during its development.

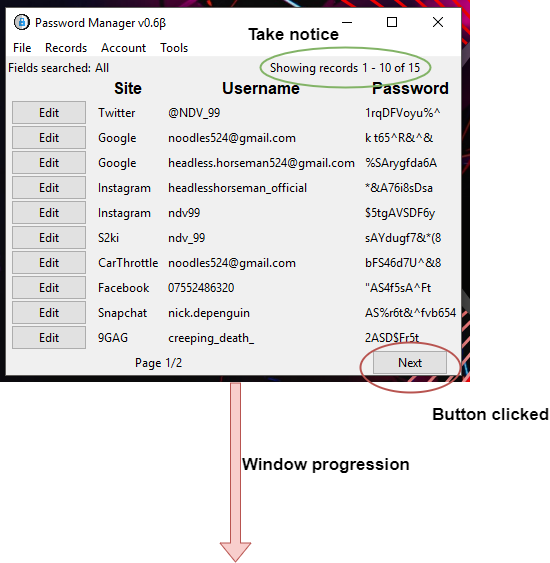


Figure - Key for screenshot diagrams

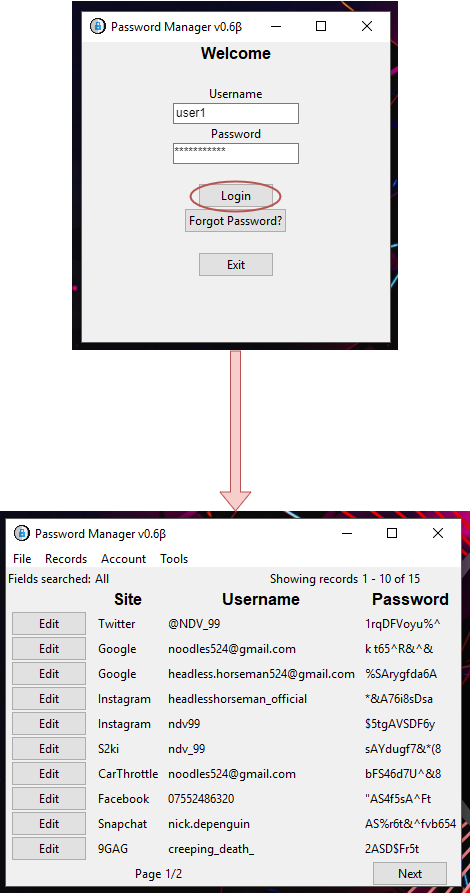


Figure 17 - Logging in

A simple process. If the user’s credentials are incorrect, an error message displays, and the user does not progress to the main menu.

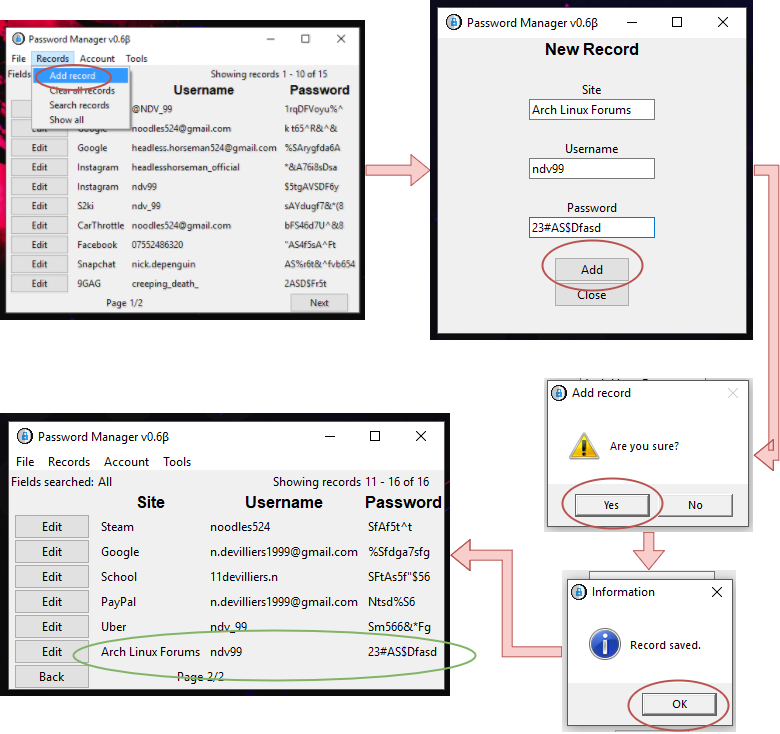


Figure 18 - Adding a record

Note: The ‘New Record’ window will stay open until closed manually, so the user can add multiple records in less time. The main window will be updated once that window is closed, and the new records will appear after all the pre-existing records.

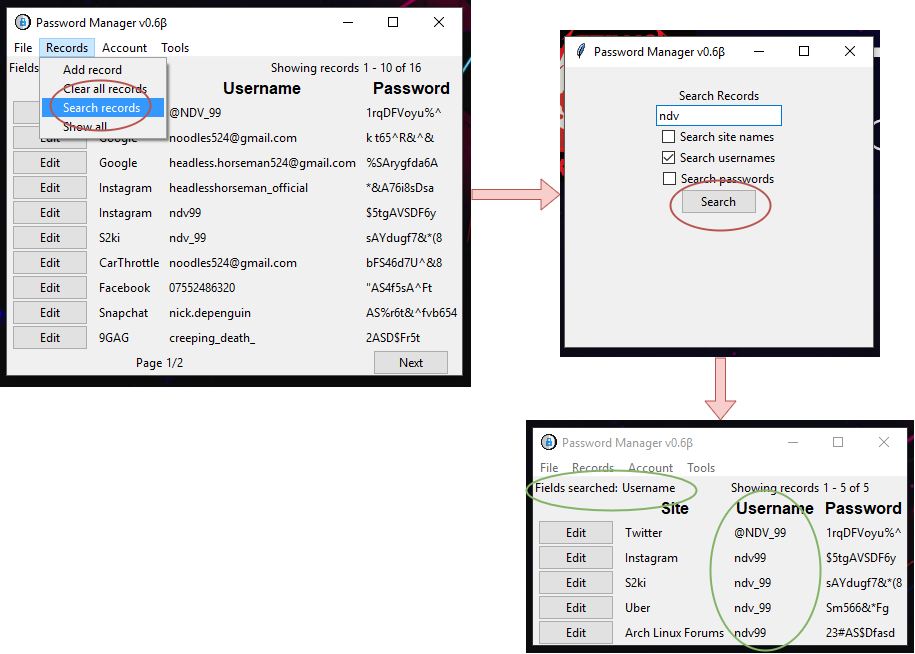


Figure 19 - Searching for records

The program allows you to narrow down your search to specific fields, and will show you which fields are being searched when the results are returned.

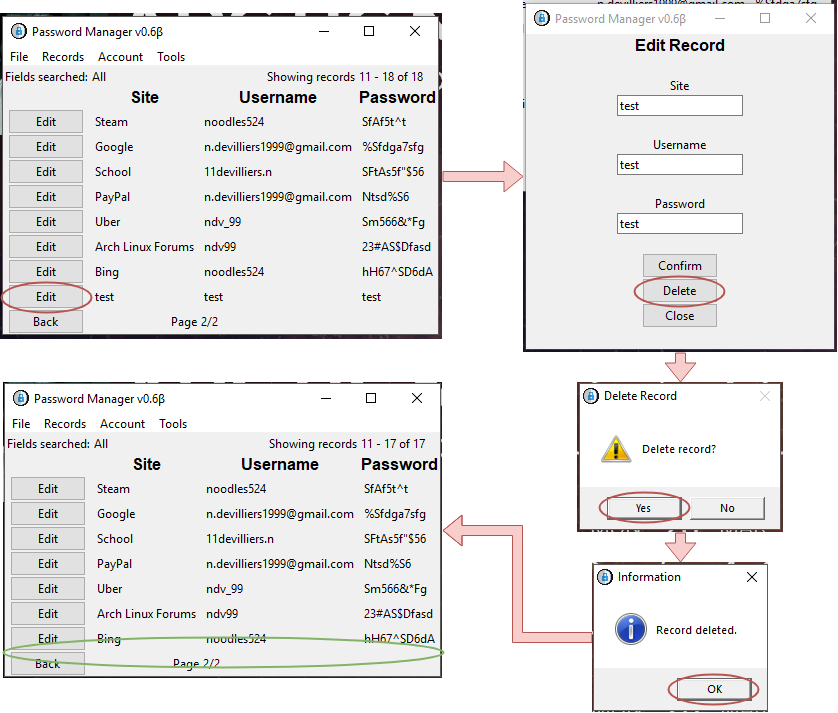


Figure 20 - Deleting a record

When a record is deleted, the table is refreshed immediately to reflect the changes.

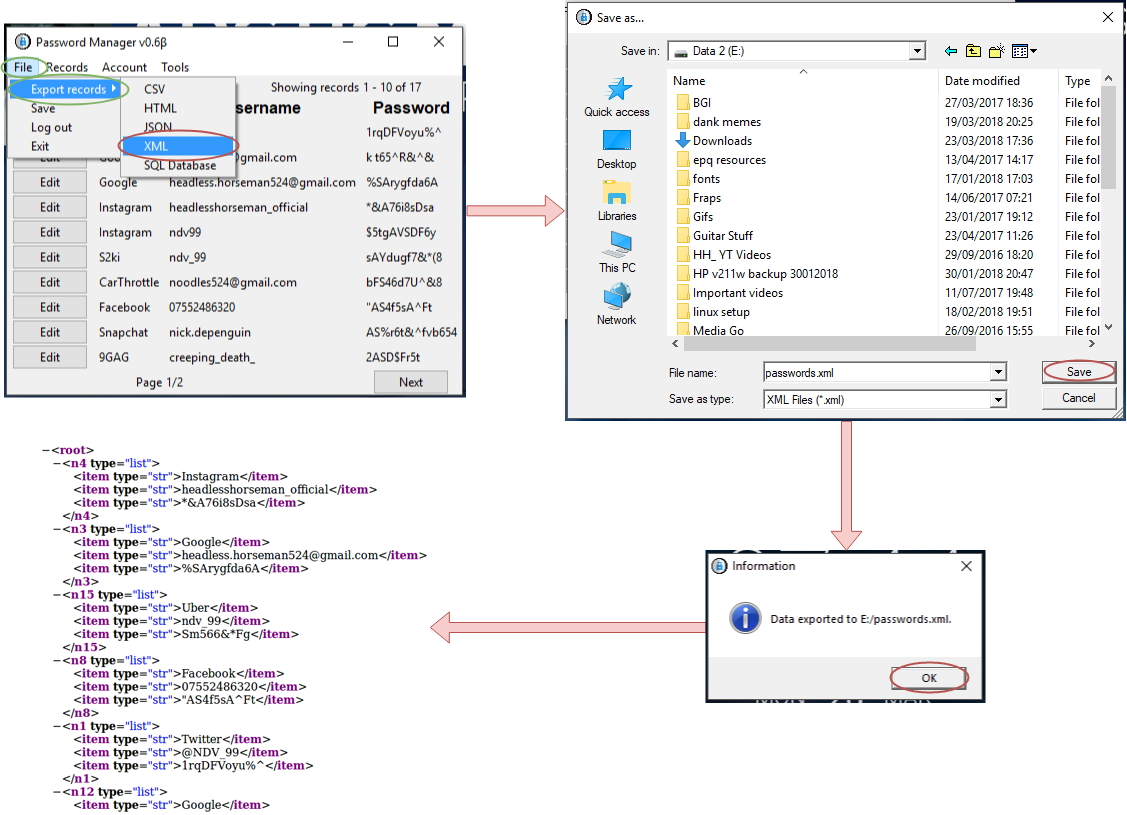


Figure 21 - Exporting a record (XML)

Records can be exported to five different file types; an XML file is used in this example.

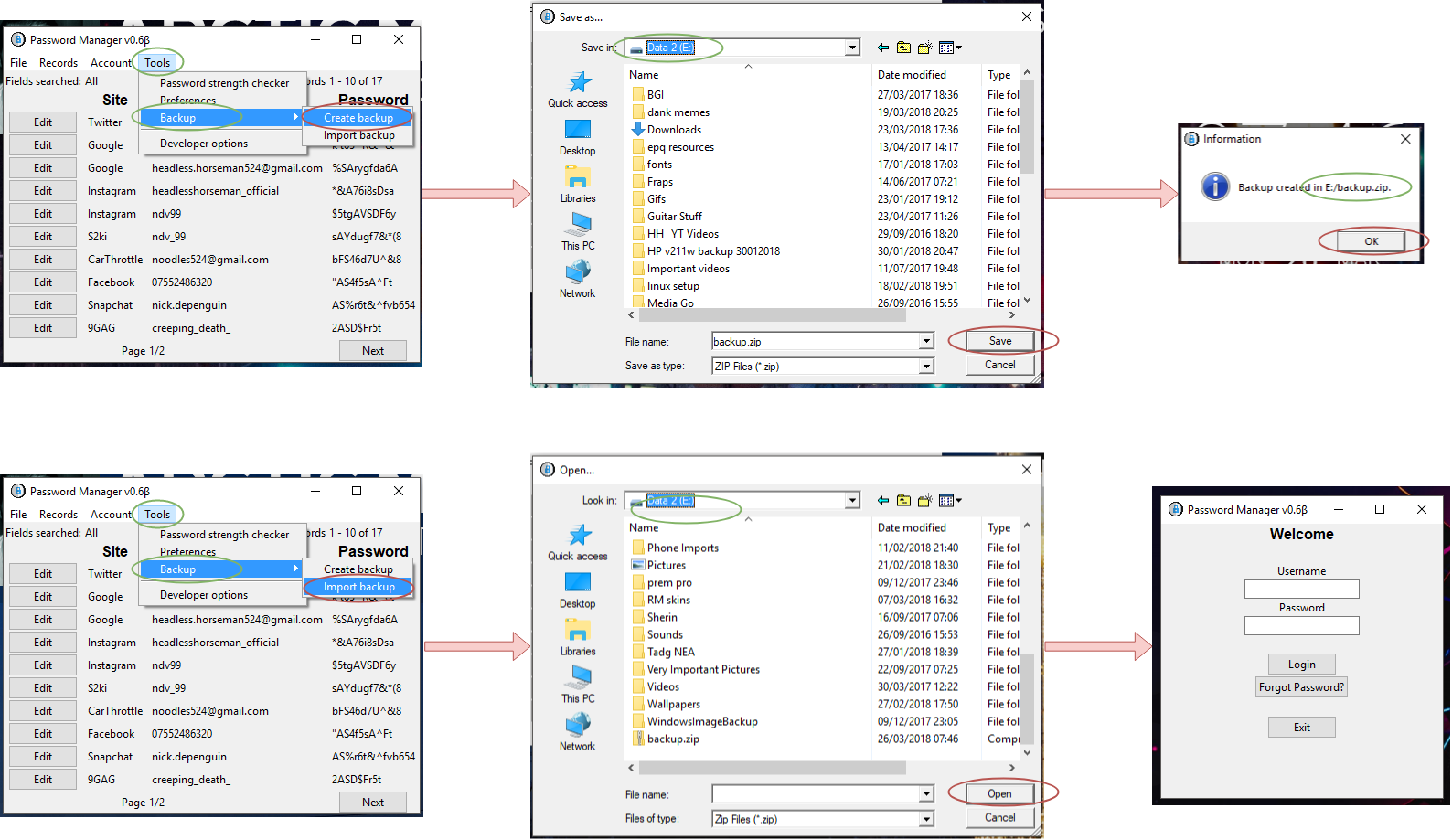


Figure 22 - Creating/Importing a backup

Backups of records are stored in ZIP files, and can be imported easily. The program is restarted when a backup is imported.

# Testing

Due to the extensive features of the program, there was a lot of testing to be done. This includes testing the database operations, the security of the encryption, and testing all the buttons in each window, ensuring that they

## Test 1 – Opening windows

The first form of testing was to ensure that all the windows opened, and in the correct way. The only way to do this was to manually run through the program. Using “Figure 5 – Windows” as a reference, I began at the login window and ran through each function of the program to open all the various windows. The Main Menu window had to be loaded in by three different methods:

* Through the standard user login
* Through the developer login
* By resetting the password.

The window loaded correctly under all three conditions. For the rest of the windows, only the developer login was used, so I would have full access to the program. Each window was tested and opened correctly, bar the Developer Options window – when I was using standard Tkinter, the entry boxes for the email settings and user details would automatically fill themselves with the current settings, but when I switched over to Themed Tkinter (so that the styling would look more up to date), the entry boxes no longer filled themselves. This was resolved by adding a “mainloop” function to the window. Following that change, all windows opened correctly, displaying the relevant buttons, entry boxes, and text labels.

## Test 2 – Basic GUI Functionality

Knowing now that all the windows opened correctly from the right buttons, the next thing to test was the functionality of the program, and that every button has a working function. Part of this test was already completed during the process of logging in through all three methods and opening different windows in the program. The rest of this test involved testing all of the front-end record operations (add (Figure 18), search and edit (Figure 19), delete (Figure 20), and clear), as well as testing the different tools available, including those in the Developer Options section. I tested them by running through each section of the menu bar, as follows:

1. File
   1. Export as:
      1. CSV
      2. HTML
      3. JSON
      4. XML
      5. Database
   2. Save
   3. Log out
   4. Exit
2. Records
   1. Add records
   2. Clear all records
   3. Search records (using different terms and conditions)
   4. Show all records (after searching for specific ones)
3. Account
   1. Edit details
   2. Clear details
   3. View login records
4. Tools
   1. Use the password strength checker (done in test 4)
   2. Change preferences
   3. Backups
      1. Create a backup
      2. Import a backup (after making changes)

Of course, some program features aren’t accessed through the menu bar, but rather through the main window and the buttons on it. To test these features, I had to:

1. Edit a record and ensure the window updated to show my changes
2. Delete a record and ensure the window updated to show my changes
3. Move between pages of records (on both the main page and login records window)
4. Ensure that the record counters were accurate

These tests went mostly as expected. However, there were some errors that required fixing. The first of these was page navigation between records. I had implemented this system by introducing instance variables into the MainWindow class that tracked the page number, the records to display on each page, and upper and lower bounds. These bounds told the program which index is the highest/lowest that should be on display. Clicking the next/previous buttons should shift these bound up and down. Moving forwards pages appeared to work fine but moving backwards consistently gave me errors or didn’t work. I eventually traced this down to my algorithm for shifting down the lower and upper bounds and made amends to this. The upper bound was being set lower than the lower bound, so the program would get confused and not display any records, since the range was mathematically incorrect.

When attempting to export the (decrypted) database to a CSV file, I noticed that there was a blank line in between each record. There didn’t seem to be a solution to this, but it was not a major issue. As well as this, there was an error where the entire database would be cleared when trying to create a backup, which isn’t ideal. I realised that this was happening because the record dictionary was being cleared every time the database was saved and exported (a similar error was happening when I first implemented autosave), and then when the program was shut down, the empty dictionary would overwrite the database. I fixed this by adding in a default argument called “preserve” to the functions that handled encrypting and writing the records back into the database, settings its default value to False, and setting it as True when calling it from the autosave or exporting functions.

## Test 3 – Exporting to Different File Types (Figure 11)

This test was relatively simple to conduct, and just involved testing the different methods of exporting (Figure 11). However, there was an error when exporting to an XML file – this turned out to be an error within the module that I was using for this process, ‘dicttoxml’. The module was encountering errors if the dictionary keys were integers (which they were in my case), and there were two ways of overcoming this – having the dictionary keys be made as strings at the point of creation, which would involve major modification to my program since a lot of its functionality relies on the keys being integers. Alternatively, I could use a dictionary comprehension to convert the integers to keys *only for the XML export process.* This would be a lot more time-effective, and a lot simpler. This worked around that bug efficiently and meant I could easily export to an XML file (Figure 21). I implemented this fix, and it worked as expected. One other error I was experiencing with the XML exporting was that some of the text in my passwords was being picked up as XML tags in Notepad++. I decided that it would be best to leave this as is, since I do not want to be restricting what a user can and can’t have in their passwords for various services.

## Test 4 – Password Rules

To test this, I started by attempting to create a new account, and tried various unacceptably weak passwords, including “password”, “letmein”, “password123”, and others, ensuring that my rules work. The rules I put in place for a very strong password are:

* At least 8 characters in length
* One or more lower case and uppercase letters
* At least one numeric character
* One or more symbols

I tested this in the account settings window too, as well as in the password strength checker. Each of these tests passed as expected and returned the correct results. I ran this test in the strength checker tool as well, and it returned the appropriate results, ranging from ‘Very Weak’ to ‘Very Strong’, based on my changes to length and character type inclusion.

## Test 5 – Autosave and login timeout

Running tests on these features was theoretically a simple process. Testing the login timeout would require two different tests, however. The first test is as follows:

1. Log in
2. Wait for the timeout period to end (60 seconds)
3. Ignore the popup message
4. Ensure the program closes after 20 seconds

And the second test:

1. Log in
2. Wait for the timeout period to end (60 seconds)
3. Respond to the popup message to show that I am still on the program
4. Wait for another timeout period (60 seconds)
5. Ensure that a popup message is shown again
6. Ignore the popup message
7. Ensure the program closes after 20 seconds

Testing autosave was simple:

1. Open the program
2. Make a change to the records
3. Wait for the autosave interval to pass (20 seconds in my test)
4. Force close the program
5. Open it again and see that my changes were in place

The first test was easily passed, with the program logging out 20 seconds after the popup message was displayed (I hardcoded this time interval for security purposes). The second test, however, unearthed an error in the process. If I replied ‘Yes’ to the message, signalling that I was still active, the timeout process would not run for the remainder of the session. The fix for this was surprisingly simple – I just had to make sure that in the timeout function, the call to itself would happen regardless of the ‘if’ statement for the popup message. Upon fixing this, the timeout worked exactly as intended.

An occasional error that was happening would be that the program would wipe all the data from the record dictionary – eventually, I tracked this down to (probably have been) a collision error between the autosave process and the close process, as they were both trying to access the records simultaneously. I worked around this by adding a process lock to the autosave function, and a new variable ‘self.stopped = False’ to the Window Manager constructor method. This was set to true as soon as the close function was called, and the autosave process cancelled, before saving the records for the last time.

## Resolution of testing

Running these tests help to expose some key errors in my code, and bring them to my attention, so that I could fix them for the final version. Had I not run these tests, I would have little idea that these issues existed in my program, which would have been catastrophic, since these were all key features of the program. The autosave testing was arguably the most important, since it had the potential to corrupt the user’s data if it did not function correctly, and my testing exposed errors that lead to data corruption.

# Evaluation

## Reflection on Third Party Feedback

Looking back on what I’d discussed with my neighbour and teacher, I followed their advice very closely, and their experience helped to push my program even further into becoming a fully-fledged and feature-packed password manager. Most of the advice I was given, I followed, including the addition of more types of exports, breaking down the code into separate modules, and reworking my class inheritances. However, there were some things I chose not to do, as it would not have been as practical to follow. For example, one suggestion was to add validation for site names to check whether they existed. I decided against this, since the user may want to store passwords for offline services, or even for websites that they are developing that haven’t gone live yet.

## Analysis of third party feedback

Personally, I found that with third-party feedback, you should analyse suggestions and comments before making changes to your program based upon them. While some suggestions may be perfectly good, you should always think back to your initial vision of the final product, and whether some suggested features would complement the program or over-complicate it. In many situations, it is worth leaving some features out if you’re looking to make your program more user-friendly and intuitive to use, which is what I was aiming for. For a more advanced/technical program, it may have been useful to implement more features, however these programs often serve to a smaller market group and are more specialist/enthusiast oriented. However, when it comes to the feedback on actual code style and writing, it is paramount that advice on this is followed, especially when given by someone with many years more experience than you. This was absolutely the case for me, as I knew I needed to keep my code as close as possible to industry standards.

## Coding style and PEP8 conformity

For the duration of the development of this software, my IDE (Integrated Development Environment) of choice was JetBrains’ “PyCharm Community Edition”. This IDE works on both Windows and Linux, which was important to me since I ran both systems and developed on both. PyCharm helped me to ensure that my code conformed to PEP8 standards as I coded in more classes and functions, which is highly important when coding a program like mine. Another useful feature was the ability to collapse blocks of code (classes, functions, loops, and statements). This meant that I could see everything that was going on in my program much more easily, and I could adjust my focus to a particular section of code with ease.

Prior to using PyCharm, I was using python’s built-in IDE, “IDLE”. This environment was far too basic – it didn’t show line numbers in the left margin, there was very limited ability to find and replace items in the code, and the colour-coding was very poor. I switched over to PyCharm in version 0.2 beta of my program, and it had highlighted just how many PEP8 errors I had made when coding. To give an idea of how bad the issue was: version 0.1 beta was 383 lines in length and contained 177 PEP8 errors – almost one error every two lines.

## Did I meet my objectives?

To recap, my primary objectives were:

1. Ensure all records are encrypted (with a Vernam cypher and/or hashing)
2. Store records in an SQL database
3. Have the user put in a username and password to be allowed access
4. Allow the user to search for specific details
5. Require two-factor authentication to reset passwords

These objectives were all met in the final program, keeping it close to my original vision of what the program would look like and how it would function. I also stated some potential features that could be added, but weren’t essential to the core functionality:

* Ability to create backups of the records
* Exporting records to different file types
* Built-in password strength checker
* Autosave
* Automatic login timeout

Every one of these features made it into the final program, and more features on top of that, including Linux compatibility, more aesthetically pleasing GUI styling, a custom icon, and a developer mode.

Considering that all the primary and secondary objectives were met, I was very pleased with the outcome of the program. It kept true to the original vision of my program, while expanding on it and improving it along the way and increasing the security of the program.

## What could have been improved?

Initially, my program was coded without the use of Object Oriented Programming. This meant that a considerable amount of time was spent adding in classes and sorting out methods between them. If I had done it using OOP from the very start, I could’ve significantly reduced the development time, and my code may also have been closer to industry standards.

As well as this, I wrote the windows in my GUI as separate functions, as opposed to classes. Only when I was near the end of the development of my program did I realise that I should’ve written the GUI as a series of classes – this would’ve made it a lot easier to work with when programming it and adding features. At one stage they were kept as class methods within a class containing all the functions to create them – I realised this was unnecessary and converted them all to instance methods. At this stage, I was already looking at 12 different windows, so this wasn’t a quick and easy task.

Lastly, I would’ve written my code so that the classes were more independent from each other. This would’ve allowed me to break down the final product into more modules and reduce the number of lines of code in the main program – two thousand lines is more than excessive.

## Ideas for new, useful features

Given more time, I would’ve liked to figure out how to keep the program running in the tray when in windows, so that it could be easily accessed if the user wanted. I also would’ve liked to have implemented a way of storing hyperlinks but hiding them, so that you could click on a site name in the search results, and it would take you to whichever URL you linked that record to.

Some other ideas that would be nice to include would be:

* Copying username/password to clipboard by just clicking it
* Scrolling on the search window and login record window
* Custom, more intuitive GUI
* Port to Android App

Given more time, I reckon that at least a few of these features could feasibly be implemented. However, considering the development time and restrictions around that, I feel that this program is already fully-fledged and packed with features, and is more advanced than the time spent developing it suggests.

# References

Chaffey, D. (2017, April 27). *Global Social Media Statistics 2017*. Retrieved from Smart Insights: www.smartinsights.com/social-media-marketing/social-media-strategy/new-global-social-media-research/

friedelwolff. (2016, February 27). *Virtaal*. Retrieved from GitHub: https://github.com/translate/virtaal/blob/master/setup.py

McLeod, B. (2017, April 25). *100+ Social media statistics you'll want to see*. Retrieved from Blue Corona: https://www.bluecorona.com/blog/social-media-statistics-2017

Shefali. (2010, January 12). *Writing copyright information in python code*. Retrieved from Stack Overflow: https://stackoverflow.com/questions/2048874/writing-copyright-information-in-python-code

# Appendix

## Appendix A – Program Changelog

* 1. beta to 0.2 beta:
* Converted back-end functionality to object-oriented
* Introduced login on the front end
* Ability to edit account settings
* Records are written to a dictionary, which is written into the database at the end of runtime
* Implemented password reset by email
* Added shutdown protocol
  1. beta to 0.3 beta:
* Created ‘Windows’ class, stored all windows as class methods
* Added admin mode with developer options window
* Added login attempt logging (and window to display attempts)
* Visual improvements
  1. beta to 0.4 beta:
* Added new ‘tools’ window
* Added exporting as CSV and database
* Improved class inheritance structure
* Wrote FileManager as class with abstract methods, create classes for different file types from that
* Added password strength checker and password rules
* Added version to window title
* Visual improvements (using Themed Tk)
  1. beta to 0.4.1 beta:
* Added global constants to top of program for message boxes
* Added “feature not available function” to show message box – for development use
* Condensed standard user login and admin login into a single form, added separate username for admin
* Made preparations to implement backup facility
  + 1. beta to 0.4.2 beta:
* Implemented autosaving
* Visual improvements
* App settings are now stored in an INI as opposed to loose TXT files
* Fully implemented backup creation and importing
* Redesigned and extended functionality of ‘Tools’ window
* Implemented login timeout
  + 1. beta to 0.5 beta:
* Merged Vernam modules into one cypher module
* Added initial paths and filenames to import/export
* Converted all methods in Windows class to standard methods
* Made settings globally accessible
* Folder organisation
* Selection and indication of searched fields
* Export records directly from search window
* Exporting changed to dropdown menu as opposed to buttons
* Added facility to export to JSON
* Made preparations for exporting as XML, and HTML
  1. beta to 0.5.1 beta:
* Implemented exporting to XML files and HTML Files
* Fixed autosaving
* Updated all docstrings
* Slight visual changes
* Merged all 3 databases into one
* Added icon
* Moved DBManager and FileManager (and its subclasses) into separate modules
  + 1. beta to 0.5.2 beta:
* Added Linux/Ubuntu compatibility
* Fixed bug with XML exporting
* Updated and added docstrings
  + 1. beta to 0.6 beta / 1.0:
* New main screen (shows records straight away)
* Ability to go between pages (main records and login records)
* Ability to alter number of records shown per page
* Added page number and records shown indicator
* Added menu bar to main window
* No more tools window (all tools available as menu bar options)
* Added preferences window
* Enhanced record adding/editing (changes are displayed immediately)
* Ability to log out without closing program
* Option to manually save changes
* Prompt to save changes if autosave hasn't saved already when exiting
* User given ability to clear all app data
* All Tkinter windows converted to classes
* Improved PEP8 conformity
* Added ‘About’ window

## Appendix B: Skeleton Program

class RecordManager(DBManager):

def \_\_init\_\_(self):

super().\_\_init\_\_()

def decrypt\_records(self):

pass

def encrypt\_records(self):

pass

def add\_record(self, site, username, password):

pass

def change\_record(self, site, username, password, index):

pass

def delete\_record(self, index):

pass

def search(self, search\_text):

pass

class Windows(Security, RecordManager)

def \_\_init\_\_(self):

super().\_\_init\_\_()

def login\_window(self):

pass

def new\_user\_window(self):

pass

def menu\_window(self):

pass

def edit\_record\_window(self):

pass

def edit\_account\_window(self , index, site,

un pw):

pass

def search\_records\_window(self):

pass

def new\_record\_window(self):

pass

from tkinter import \*

import sqlite3

import vernam\_decrypt

import vernam\_encrypt

import email

import smtplib

class DBManager:

def \_\_init\_\_(self):

pass

def write\_to\_main(self, site, username, password):

pass

def read\_database(self):

pass

def change(self, site, username, password, index):

pass

def delete(self, index):

pass

class FileManager:

def \_\_int\_\_(self):

pass

def write\_username(self, username):

pass

def read\_username(self):

pass

def read\_password(self):

pass

def write\_password(self, password):

pass

class Security(FileManager):

def \_\_init\_\_(self):

super().\_\_init\_\_()

def log\_in(self, username, password):

pass

def check\_username(self, username):

pass

def check\_password(self, password):

pass

def create\_user(self, username, password, address):

pass

def edit\_user(self, username, password, address):

pass

def forgot\_password(self, username):

pass