Project: MyLife Writeup
Colin Poindexter and Guwanjith Tennekoon

Our project essentially uses just one class, *MyLife*. This ensures that the code is cleaner and more concise.
The constructor for this class only takes in the parameter, *self*, after which it continues to ask the user for inputs to verify whether the user is new to the system or not, using an if statement.
If the user is not new, then the previously created(when the user first used the system) file *myLifePrefs*, is opened in reading mode to access data without modifying anything. Then, instance variables are created for each user input field into the file. Then the method *welcome* is called, with the user's name input as a parameter, using the accessor function *getName*.
If the user is found to be new, then a new file is created, *myLifePrefs,* to be written into.
Once the file is opened, the user is asked to input their name, favorite Team, favorite Artist, favorite Twitter personality, and current location. All this info is then written into *myLifePrefs* on individual lines. (name on one line, Favorite artist on the next line, Favorite team on the following line,etc.)
After the file is written to, the user is notified that the profile has been created. *myLifePrefs* is then closed.
myLifePrefs is then re-opened, but this time it is opened in reading mode, which enables access to the data within the file, without being able to edit anything within the file.
Then, the function goes through the file, r and makes instance variables out of the user- defined name, *self.name*,favorite artist, *self.favArtist*, etc.
Once all the instance variables have been created, a new file,*myLifeLearning*, is created.
Then, the function calls the method,*welcome* ,which displays a greeting that uses the user's name (which is input as a parameter using the accessor method *getName*). Then, the method *weatherDash* is called, with the user's previously entered zipcode, passed in as a parameter. Then, the method *getNews* is called, and the current news stories are printed legibly. Then, the method *getTweets* is called, with the user's favorite twitter personality passed in as a parameter, and the tweets are printed out legibly. Then, the method *getConcertsDash* is called with the user's location passed into it as a parameter. Finally, The method then calls the *menuChoice* method.Then, the method looks for the tag "location" in the xml code. Once the tag is found, the value associated with the attribute "city" is stored as the value for the key "City" in *currentWeather*. Similarly, the value associated with the attribute 'region' is stored as the value for the key "State" in *currentWeather*.The *weatherDash* method takes in a location (a zipcode) as a parameter. This ocation is then used to modify the weather API url, in order to get the weather info about the user-defined location.
Once the xml info has been parsed, the method looks for the tag "condition" within the xml data. Once found, the code looks for the attribute "text", and stores the value associated with the attribute as the value associated with the key 'Condition' in a new dictionary, *currentWeather*. Then, the value associated with the attribute "temp" is found, and stored as the value for the key 'Temp' in *currentWeather*. The weather info is then displayed to the user in a conversational format.

The *getNews* method only takes in self as a parameter. Then, the url for getting the current news is used to obtain the related xml code. Then, the method looks through the xml code for the “title” tag. Then, for each news item is individually appended to a new list *newsList*. The method then returns *newsList*

The *getConcertsDash* method takes in a location (a zipcode), and uses this location to modify the url associated with the api for obtaining the xml code with information about concerts, in order to obtain the xml code for info about concerts in the user’s location. Once the xml code has been parsed, the method looks through the code for the “event” tag. In this tag, the value assosciated with the string ‘name’ is stored as the value for ‘Concert’ in the newly created dictionary *concertList*. Similarly, the values associated with the start date/time, end date/time, venue_address, and other important information for the user to get to know where the concert is playing and between which times, are store in *concertList*, with appropiate keys assigned to them.

Then, if *concertList* is found to be empty, the user is notified that there aren’t any concerts in the area. However, if *concertList* is not empty, the user is given information about the concert closest to the user’s location.

The *menuChoice* method allows the user to decide between using the conversation mode or menu mode for navigating through the function. If the user chooses conversation mode (the user can type a whole sentence, but the method uses an if function to ensure that either the string ‘conversation’ or the string ‘quick’ is present in the user-input string). If the string ‘conversations’ is found, then the user is directed to the *conversation* method. If the string ‘quick’ is found, then the user is directed to the *menu* method. If neither is found, the user is presented with the error message: “I’m sorry, I didn’t get that” and then the *menuChoice* method is recalled to allow the user to re-enter the choice.

The *conversation* method asks the user to input what they want to do. This string is then passed into the *selector* method.

The *selector* method makes the user-input string lower-cased. The string is then split, and then a list *optionList* is created to label the new version of the string.Then, the method checks whether the string ‘*weather’* is in *optionList*. If ‘*weather’* is found, it then checks whether the string ‘*in’* is found in *optionList*.

While ‘weather’ is found

If ‘in’ is found, then the *loc* is used to label [the index of ‘in’ in *optionList*+1]. ‘city*’* is used to label the value located at the index value of *loc* in *optionList*. ‘State’ is used to label the value located at the index value of *(loc+1)* in *optionList*. Then the method calls the api needed to access location info (to check that the location actually exists), and modifies it, so that the web-address used accounts for the specified *city* and *state*. Then, the code goes through the xml code and checks for the tag “uzip”, and gets the value found at index value[0] within the tag “uzip”. Then *zipcode* is created to label the value assosciated with: zipGet.childNodes[0].nodeValue Then, zipcode is passed into the *weather* method.

If ‘in’ is not found, then it checks whether the string “current” is found within the string. If “current” is found, then self.getLocation() is passed into the *weather* method.

If neither ‘in’ nor ‘current’ is found in the string, the function prints an error message,”sorry I didn’t get that”, and then sends the user to the *menu* method.

The *weather* method takes in a location as a parameter. This location is then added to a modified api url, in order to obtain the related xml information. Once the xml info has been parsed, the method looks for the tag “condition” within the xml data. Once found, the code looks for the attribute “text”, and stores the value associated with the attribute as the value associated with the key ‘Condition’ in a new dictionary, *currentWeather*. Then, the value associated with the attribute “temp” is found, and stored as the value for the key ‘Temp’ in *currentWeather*.
Then, the method looks for the tag “location” in the xml code. Once the tag is found, the value associated with the attribute “city” is stored as the value for the key “City” in *currentWeather*. Similarly, the value associated with the attribute ‘region’ is stored as the value for the key “State” in *currentWeather*.
Then, a string containing conversational language and the relevant information from *currentWeather* is printed out for the user to see.
Finally, the method calls the *secondary* method.
If ‘weather’ is not found
The method then looks for the sub-string “tweets” within *optionList*.
If ‘tweets’ is found
The method looks for the sub-string “favorite” in *optionList*. If the sub-string is found, the label *tweets* is created to represent the output of the *getTweets* method when the value associated obtained from the accessor method *self.getFavTit()* is entered into the method as a parameter.
If “favorite” is not found in *optionList*, the method looks for the sub-string ‘from’ and gets the associated index value .This is stored as *twitFrom*. Then, a new label, *twit,* is created that represents the value found at the index value right after that of *twitFrom* in *optionList.* The label *tweets* is created to represent the output of the *getTweets* method when the *twit* is entered into the method as a parameter.
Then, the method uses a for loop to go through and print each tweet found in *tweets*, using punctuation and line-spacing to maintain legibility.
Finally, the *secondary* method is called.
The *getTweets* method takes in a parameter *twit*, which is used to modify the url associated with the API for obtaining the relevant xml information.
Then, *tweetGet* is used to represent the values under the tag “title” in the xml code. The method then appends each value associated with the tag”title” in the xml code to a list *tweetList*. Hence *tweetList* now contains all the relevant tweets.
Then, the code checks *tweetList* to check whether the List contains the string: “from”+*twit*+”- Twitter Search” The above string is found when a search for a twitter profile is unsuccessful. Hence, a message informing the user that the person is not on twitter is displayed. If the above string is not found, TweetList is returned.
If ‘tweets’ is not found
The method then looks for the sub-string “directions” within *optionList*.
If ‘directions’ is found
The method then looks for the substring “from” in *optionList*. If it’s found, the method then looks for the substring “to” in *optionList*.
If both are found, the method stores the index value that shows where the substring “from” is found in *optionList* as *direction*.

Then, the method stores the index value that shows where the substring “to” is found in *optionList* as *from.*
Then the substring found right after “from” in *optionList* is stored as *directionFrom.* The substring found right after “to” in *optionList* is stored as *directionTo*.
The label *directions* is used to represent the value obtained by entering *directionFrom* and *directionTo* as parameters into the *getDirections* method.
Then the directions are made legible by allowing a step-by-step, line by line output of the directions.
If ‘from’ is not found in *optionList*, the user is then displayed a message indicating that a starting point and a destination are both required. The *secondary* method is finally called.
The *getDirections* module takes in *fromPlace* and *toPlace* as parameters. The url associated with the API for getting directions is then modified using these parameters , in order to obtain the required directions. Once the xml code is parsed, the method looks for the tag”html_instruction”. All values under the tag are stored as *directionsGet*.
The method then appends each potion of the code into a list *directionList*, which is finally returned.
If ‘directions’ is not found
The method then looks for the sub-string “news” within *optionList*.
If ‘news’ is found
The method *getNews* is called.Then each news item is printed in a legible format. Finally, the methd *secondary* is called.
If ‘news’ is not found
The method then looks for the sub-string “concerts” within *optionList*.
If ‘concerts’ is found
The method looks for index associated with the string ‘near’ in *optionList*, the integer 1 is added to this value to obtain the index value for the string right after the string ‘near’. This index value is stored as *loc*. Then, the string found at the index value of *loc* in *optionList* is stored as *city*. The string associated with the index value of (*loc*+1) in *optionList* is stored as *state.* Then, *city* and *state* are used to modify the url associated with the API for obtaining the relevant xml code about locations. If the location exists, the method looks for the tag *’uzip’*, and finds the appropriate attribute for obtaining the zipcode. This zipcode is then input as a parameter into the method *getConcerts*.
If ‘concerts’ is not found
The method then looks for the sub-string “no” within *optionList* **If ‘no’ is not found**
The string ’Goodbye’ is displayed, and the function ends.
If ‘no’ is not found
The file, learning is called on. Each line in the file is then split on the string “#$@” into a list of strings, separated by commas,*learningList*. This List is then converted to a dictionary,*learningDict*, by using the value at index 0 in the list as the key, and the value at index 1 as the dictionary value for each line of the file.
Then, if the user-entered string, *option*, is a key found in *learningDict*.
If found to be a key found in *learningDict*, the value at index 1 of ***learningList*** is split, to form a new list *mentList*.

If the value at index 0 of *mentList* is the same as the string "weather", the user is asked to input the zip code again, this is stored as *loc*.
If *loc* has a value that is not the same as None, *loc* is input as a parameter to the method *weather*.
If the value at index 0 of *mentList* is not the same as the string "weather", the method checks whether it is the same as the string 'twit'.
If it is the same as 'twit', the user is asked to input the twitter handler again. This is stored as *twit.*Then, *twit* is entered as a parameter into the *getTweets* module. This is stored as *tweets*. Then, *tweets* is printed.
If the value at index 0 of *mentList* is not the same as the string "twit", the method checks whether it is the same as the string 'directions'. If it is the same as 'directions', the user is asked to input starting location. This is stored as *fromPlace.*Then, the user is asked to enter the destination. This is stored as *toPlace.* Then, *fromPlace* and *toPlace* are entered as parameters into the *getDirections* module.
If the value at index 0 of *mentList* is not the same as the string "directions", the method checks whether it is the same as the string 'concerts'. If it is the same as 'concerts', the user is asked to input the location (zipcode) of the concert they would like to attend. This is stored as *loc.* Then, *loc* is entered as a parameter into the *getConcerts* module.
If the value at index 0 of *mentList* is not the same as the string "directions", the method prints an error message, indicating that the request was not understood and that the *menu mode* will be implemented. Then, the *menu* method is called on, with the user-input string *option* passed into it as a parameter.
Menu method/ Menu mode
The menu mode takes in *errorText* as an input.
The *menu* method opens a file *myLifeLearning*, in append mode. This file is labeled *learning*. The user is then asked to choose a number corresponding to what they meant to choose in the conversation mode. This string is then converted to an integer. This integer is stored as *selectedItem*.
If *selectedItem* is the same as the integer 1, the error text, along with what the user had meant to say 'weather', would be added to the file *learning*. Then, the user is asked to input the zip code that they would like to get the weather about. This is stored as *location*.
Then, if *location* is not the same as None, it is input to the method *weather* in order to obtain the info on weather. Otherwise, the method *menu* is called, with *errorText* as a parameter.
Otherwise, if *selectedItem* is the same as the integer 2, the error text, along with what the user had meant to say 'tweet', would be added to the file *learning*. Then, the user is asked to input the twitter handler that that they would like to get the tweets from. This is stored as *twit*.
Then, *twit* is input to the method *getTweets* in order to obtain the tweets. This is stored as *tweets*. Then the tweets are printed in a legible manner, and the *secondary* method is called.
Then, if *selectedItem* is the same as the integer 3 the error text, along with what the user had meant to say 'directions', would be added to the file *learning*. The user is asked to input a starting location. This is stored as *fromSource*. The user is then asked to input a destination. This is stored as *toSource*.
Then *fromSource and toSource* are input as parameters into the method *getDirections* in order to get the relevant directions.

Then, if *selectedItem* is the same as the integer 4 the error text, along with what the user had meant to say ‘concert’, would be added to the file *learning*. The user is asked to input the location the concert they want to attend is. This is stored as *where.*
Then *where* is input as parameters into the method *getConcerts* in order to get the relevant concert information.
Finally, if *selectedItem* is the same as the integer 5 the error text, along with what the user had meant to say ‘news’, would be added to the file *learning*. Then the method *getNews* is called in order to get the relevant news. The news is then printed legibly.
Finally, we created a method *eliminator*, which gets rid of unneeded words. This method takes in a user-input string, *errorText* as a parameter. The string *errorText* is then converted to lower-case. Then, a new list consisting of strings of individual words from *errorText* is created. This list is called *correctList*. Then, a list of containing strings that correspond to the words to be eliminated is created. This list is called *errorWords*.
Then each word in *correctList* is matched with each word in *errorWords*. If a word in *correctList* is also found in *errorWords*, the word is removed from *correctList*. Then, *correctList* is returned.