Object-Oriented Eliza Technical Document

Lydia Noureldin
February 3 2017

Contents

The .zip folder must contain 4 Files:

- 2 classes: Eliza.py and Keyword.py
- 1 text document: ElizaScript.txt (this can be anything you want)
- 1 file to be executed: runEliza.py

File breakdown

runEliza.py

Includes:

1. Eliza

Global Variables:

1. Reflections
2. Psychobabble

Functions:

1. readFile
2. main

Eliza.py

Includes:

1. re
2. Keyword
3. string

Attributes:

1. Keyword[] keys
2. Dict reflections
3. String[][] psychobabble
4. Int[][] usedResponses
5. String[] memoryLis

Functions:

1. __init__
2. reflect

3. analyze
4. getRankedResponse
5. getKey
6. leastUsedKeyResp
7. getGeneralResponse
8. leastUsedResponse
9. findPatternIndex
10. getHighestRank
11. getWords
12. setUsedResponses
13. setKeys

Keyword.py

Attributes :

1. string name
2. int rank
3. int numRegex
4. string[] regexLis
5. string[][] reasmbLis
6. int[][] usedResp
7. String[] synonyms

Functions:

1. __init__
2. __str__
3. setUsedResp
4. setAttr
5. addSynonyms
6. updateUsedResp

Algorithm

Eliza and Keyword are classes used by the runEliza.py file. Eliza analyses and responds to any user input in 1 of 3 ways.

First of all, Eliza can match Keywords that are in the user's input and come up with a specific response. This is done by using a list of Keyword objects that are stored in an attribute called "keys". Each Keyword object has attributes that include a rank, regular expression(s) and reassembly rule(s). Eliza searches the user's input for the Keyword with the highest rank that has a regular expression that matches it. It generates a specific response using the Keyword's reassembly rules and returns it through the "analyze" function.

The Eliza object has an attribute called memoryLis, which is used like a queue. This list holds potential responses based on previous user input. For something to be saved in the list, it needs to pass a certain rank threshold. Once it passes that threshold, it is added to the end of memoryLis. These responses are used when Eliza can't find any Keywords in the user's input. Instead of returning a random response, it pops the the first element from memoryLis and returns it through the "analyze" function.

Finally, if no keyword is found and there is nothing in memoryLis, Eliza pattern matches the user's input to a general response. The regular expression and reassembly rules are in another one of Eliza's attributes, a list of lists called psychobabble. This general response is returned through the "analyze" function.

Code Breakdown

runEliza.py

Imports:

Only 1 import for runEliza.

```
8    from Eliza import Eliza
```

This is necessary to use the Eliza class in main()

Global Variables:

There are 2 global variables in runEliza.py, reflections and psychobabble.

```
10  reflections = {
11              "am": "are",
12              "was": "were",
13              "i": "you",
14              "i'd": "you would",
15              "i've": "you have",
16              "i'll": "you will",
17              "my": "your",
18              "are": "am",
19              "you've": "I have",
20              "you'll": "I will",
21              "your": "my",
22              "yours": "mine",
23              "you": "me",
24              "me": "you"
25          }
26
27  psychobabble = [
28              [r'I need (.*)',
29               ["Why do you need {0}?",
30                "Would it really help you to get {0}?",
31                "Are you sure you need {0}?"]],
32
33              [r'Why don\'?t you ([^\?]*)\??',
34               ["Do you really think I don't {0}?",
35                "Perhaps eventually I will {0}.",
36                "Do you really want me to {0}?"]],
37
38              [r'Why can\'?t I ([^\?]*)\??',
39               ["Do you think you should be able to {0}?",
```

I have only shown a little snippet of psychobabble here, but the full code is in the files. Reflections is a dictionary that maps first-person pronouns to second- person pronouns and vice-versa. Psychobabble is a list of lists of type string, where the first element is a regular expression, and the second element is a list of potential reassembly rules that can be used to generate a response.

Functions:

readFile:

Let's start with looking at the file input function, readFile. I will show the code first then explain it below.

```
268 def readFile(filename):
269     fo = open(filename)
270     scriptList = []
271     for line in fo:
272         line = line.strip(' \t\n\r')
273         if line != "":
274             scriptList.append(line)
275     fo.close()
276     return scriptList
```

Parameters: filename
Return: scriptList

We open the file that is passed in and create our list of lines called scriptList. In the for loop we strip the line of any spaces, \t, \n, and \r characters from both sides of the string. Then we check if the line is the empty string. If not, we add it to our list of input lines, scriptList.

main:

```
278 def main():
279     filename = input("Please enter the filename: ")
280     userInput = readFile(filename)
281     print("Hello. How are you feeling today?")
282     elly = Eliza(reflections, psychobabble)
283     for i in range(len(userInput)):
284         statement = userInput[i]
285         print(statement)
286         response = elly.analyze(statement)
287         print(response)
288         if statement == "quit":
289             break
290     while True:
291         statement = input("> ")
292         response = elly.analyze(statement)
293         print(response)
294         if statement == "quit":
295             break
296
297 main()
```

Parameters: None
Return: None

First we ask the user to give in the text file that contains the script that they want to use. Then we use the readFile function to read the file. We store the list of strings that readFile returns into

a variable called userInput. After that, we spit out the first Eliza prompt and create the Eliza object called “elly”. Now we hit the first loop

```
283     for i in range(len(userInput)):
284         statement = userInput[i]
285         print(statement)
286         response = elly.analyze(statement)
287         print(response)
288         if statement == "quit":
289             break
```

.

This loop is iterates through the lines from the script that the user provided. In each iteration, the user’s input is printed on the console, we generate a response from elly (our Eliza object), and print elly’s response to the screen. The loop is terminated once we run out of input lines in userInput or a “quit” statement is reached.

Let’s take a look at the second loop in main()

```
290     while True:
291         statement = input("> ")
292         response = elly.analyze(statement)
293         print(response)
294         if statement == "quit":
295             break
```

This loop is very similar to the one above. The only difference is that the loop is not going through a script. Instead, it is waiting for the user’s input in the console. Once again, the loop terminates if a “quit” statement is inputted.

Keyword.py

Keyword.py is a class that is used to store a Keyword object. Initially, I was not going to store the Keywords in a class, but it was much more organized to do so. A Keyword is a word that is potentially found in the user's input. These Keywords have a ranking associated with them that indicate their importance. For example, "sad" would have a rank of 1 but "suicidal" would have a rank of 5. They must also have at least 1 and at most 10 regular expressions and reassembly rules associated with them. Finally, they have a list of integers that keeps track of how many times each reassembly rule has been used. This is used to ensure that Eliza is not repetitive.

Functions:

__init__:

This is the constructor for the Keyword object. I will show the code then explain it after.

```
31  class Keyword:
32      def __init__(self, na, ra, reg, rea, reg2=None, rea2=None, reg3=None, rea3=None, reg4=None, rea4=None, \
33                   reg5=None, rea5=None, reg6=None, rea6=None, reg7=None, rea7=None, reg8=None, rea8=None, \
34                   reg9=None, rea9=None, reg10=None, rea10=None):
35          self.name = na
36          self.rank = ra
37          self.numRegex = 0
38          self.regexLis = []
39          self.reasmbLis = []
40          self.usedResp = []
41          self.synonyms = []
42          for i in range(10):
43              self.usedResp.append([])
44          potentialAttributes = []
45          potentialAttributes.extend([reg, rea, reg2, rea2, reg3, rea3, reg4, rea4, reg5, rea5, reg6, rea6, reg7, rea7 \
46                                      , reg8, rea8, reg9, rea9, reg10, rea10])
47          for i in range(0, (len(potentialAttributes)), 2):
48              self.setAttr(potentialAttributes[i], potentialAttributes[i + 1], i)
```

Parameters: self, na, ra, reg, rea, reg2=None, rea2=None, reg3=None, rea3=None, reg4=None, rea4=None, reg5=None, rea5=None, reg6=None, rea6=None, reg7=None, rea7=None, reg8=None, rea8=None, reg9=None, rea9=None,reg10=None, rea10=None
Return: None

As you may expect, all the constructor does is set the attributes. Recall, that the Keyword object can have 1-10 regular expressions and reassembly rules. Thanks to Python, these can be default parameters set to None (Python's equivalent to NULL). The name and rank are set using values supplied by the user. numRegex is the number of regular expressions used by the Keyword. regexLis is a list of the regular expression(s), reasmbLis is a list of lists that holds the reassembly rules for the regular expressions. Synonyms is a list of strings that contains synonyms of the Keyword's name. This is an optional attribute that can be set using the addSynonyms function. usedResp is a list of lists that keeps track of which responses have been used for each regular expression.

In the first for loop, usedResp is initialized to a list that contains exactly 10 empty lists (one for each potential regular expression). The second for loop uses the list called potentialAttributes and passes them into a mutator called setAttr.

setAttr:

```
51      def setAttr(self, reg, rea, numR):
52          if reg is not None and rea is not None:
53              self.regexLis.append(reg)
54              self.reasmbLis.append(rea)
55              self.setUsedResp(rea, (int(numR / 2)))
56              self.numRegex += 1
```

Parameters: self, reg, rea, numR
Return: None

This function initializes regexLis, reasmbLis, usedResp, and numRegex attributes. First, it checks that the attributes are not None then adds them to the regexLis and reasmbLis lists. The next step is to update the setUsedResp list. This is done using the setUsedResp mutator. Note that the second parameter, which represents the regular expression's index in usedResp, is divided by 2.
Recall the loop in __init__:

```
53          for i in range(0, (len(potentialAttributes)), 2):
54              self.setAttr(potentialAttributes[i], potentialAttributes[i+1], i)
```

In this loop, we are taking steps of size 2 (specified by the third parameter in the "range" built in function). To get the regular expression index to use as a parameter in the setUsedResp function, we must divide by 2. Lastly, we increment the numRegex attribute.

setUsedResp:

```
92      def setUsedResp(self, reasmb, regExIndex):
93          for i in range(len(reasmb)):
94              self.usedResp[regExIndex].append(0)
```

Parameters: self, reasmb, regExIndex
Return: None

This function sets up usedResp list of lists. For each reassembly rule for a particular regular expression, usedResp will have a 0 at that location. The values are set to 0 because no reassembly rules have been used yet.

addSynonyms:

```
79      def addSynonyms(self, synonymsLis):
80          self.synonyms = synonymsLis
```

Parameters: self, synonymsLis
Return: None

This is a function that the user can call to add a list of synonyms to the Keyword. This is completely optional. If the user does not call this function, synonyms is set to an empty list in __init__.

__str__:
This is the string representation of the Keyword. I added this to help with debugging.

```
83      def __str__(self):
84          output = "\nKeyword name: " + self.name + "\nRank: " + str(self.rank) +\
85                   "\nNumber of Regular Expressions: " + str(self.numRegex)
86          for i in range(self.numRegex) :
87              output += "\nRegular Expression " + str(i + 1) + ": " + str(self.regexLis[i]) + "\nReasembly " +\
88                        str(i + 1) + ": " + str(self.reasmbLis[i]) + "\nUsed Responses " + str(i + 1) +\
89                        ": " + str(self.usedResp[i])
90          if len(self.synonyms) > 0 :
91              output += "\nSynonyms: " + str(self.synonyms)
92          output += "\n\n"
93          return output
```

Parameters: self
Return: output

This is a pretty straightforward string composition which displays the attributes of the Keyword. The for loop only adds the regular expressions that exist in the object. Returns the complete string that can be used with print statements.

updateUsedResp:
This method is used to update the usedResp attribute.

```
110     def updateUsedResp(self, numArray, respUsedIndex):
111         self.usedResp[numArray][respUsedIndex] += 1
```

Parameters: self, numArray, RespUsedIndex
Return: None

This function updates the usedResp list that is needed to always select the least used response. numArray indicates which usedResp array we need to update. This depends on the regular expression that was matched. respUsedIndex represents the response that was used from the corresponding reasmb list. This is incremented by 1 each time it is used.

Eliza.py

Imports:

```
16  import re
17  import string
18  from Keyword import Keyword
```

re is used for regular expression matching. String is used to get rid of punctuation efficiently. Keyword is the object described above.

Functions:

__init__:
Constructor for Eliza class

```
20  class Eliza :
21      def __init__(self, refl, psychob):
22          self.reflections = refl
23          self.psychobabble = psychob
24          self.keys = []
25          self.setKeys()
26          self.usedResponses = []
27          self.setUsedResponses()
28          self.memoryLis = []
```

Parameters: self, refl, psychob
Return: None

The constructor for the Eliza class sets its attributes. Refl is the dictionary that the user pases in to map first-person pronouns to second- person pronouns and vice-versa. Psychobabble is a list of lists of type string, where the first element is a regular expression, and the second element is a list of potential reassembly rules that can be used to generate a response. Keys is a list of Keyword objects. usedResponses is a list of lists of type int that is used to avoid repetitions. memoryLis is a list of type string that stores certain responses based on previous user statements. This is used to give Eliza memory capabilities.

setKeys:

This initializes the list of Keywords

```
156     def setKeys(self):
157         xnone = Keyword("xnone", 0, r'(.*)', [
158               "I'm not sure I understand you fully.",
159               "Please go on.",
160               "What does that suggest to you?",
161               "Do you feel strongly about discussing such things?",
162               "That is interesting.  Please continue.",
163               "Tell me more about that.",
164               "Does talking about this bother you?"])
165
166         sorry = Keyword("sorry", 0, r"(.*)",[
167          "Please don't apologise.",
168          "Apologies are not necessary.",
169          "I've told you that apologies are not required.",
170          "It did not bother me.  Please continue."])
171
172         remember = Keyword("remember", 5, r'i remember (.*)',
173                 ["Do you often think of {1}?",
174                  "Does thinking of {0} bring anything else to mind?",
```

...more keywords...

```
530             sorry.addSynonyms(["remorseful", "regretful"])
531             remember.addSynonyms(["reminded","recall","memorize","look back"])
532             dreamed.addSynonyms(["delusion","fantasy","imagination","thought"])
```

...more synonyms…

```
552         self.keys.extend([xnone, sorry, apologize, remember, forget, dreamed, dream, IF, perhaps, name, hello, computer,
553                           am, are, your, was, sad, happy, I, you, yes, no, my, can, what, who, when, where, how,
554                           because, why, everyone, everybody, nobody, noone, always, alike, like, different, family, want, belief])
555         self.keys.sort(key=lambda x: x.rank, reverse=True)
```

Parameters: self

Return: None

I have taken out some of the Keywords here but they are all in the file. This function creates all the Keyword objects, adds synonyms to each, and sorts them in decreasing order of rank in a list called keys. This is in decreasing order because it makes it more efficient to look for Keywords with the highest rank later on.

setUsedResponses:

```
135     def setUsedResponses(self):
136         self.usedResponses = []
137         for i in range(len(self.psychobabble)):
138             self.usedResponses.append([])
139             self.usedResponses[i].append(self.psychobabble[i][0])
140             self.usedResponses[i].append([])
141             for j in range(len(self.psychobabble[i][1])):
142                 self.usedResponses[i][1].append(0)
```

Parameters: self

Return: None

setUsedResponses sets up the usedResp attribute that is used to avoid repetitions. Zeroes indicate that none of the responses have been used before. In the for loop we set the first value

of the list to the regular expression. Then it continues to add lists filled with their corresponding number of reassembly rules. For example,

if first element of psychobable is:

['I need (.*)', ['Why do you need {0}?', 'Would it really help you to get {0}?', 'Are you sure you need {0}?']]

The stored usedResponses will be:

['I need (.*)', [0, 0, 0]]

The rest of the the functions are used to generate a response.

analyze:

```
53      def analyze(self, statement):
54          statement = statement.split(".")[0]
55          words = self.getWords(statement)
56          maxRankIndex = self.getHighestRank(words, statement)
57          if maxRankIndex == -1:
58              if len(self.memoryLis) != 0:
59                  response = self.memoryLis[0]
60                  self.memoryLis.pop(0)
61              else:
62                  response = self.getGeneralResponse(statement)
63          else:
64              if self.keys[maxRankIndex].rank > 2:
65                  responseInMemory = self.getRankedResponse(statement, maxRankIndex)
66                  self.memoryLis.append(responseInMemory)
67              response = self.getRankedResponse(statement, maxRankIndex)
68          return response
```

Parameters: self, statement

Return: response

Recall the line in main():

```
257         response = elly.analyze(statement)
```

First, we chop off anything after the first period in the user's input. This is necessary because Eliza is easily confused with multiple sentences. After that, we break up the statement (user input) into a list of words using the getWords function which splits the string with a " " delimiter. Then we use the getHighestRank function to find the word in the user's input with the highest rank. This word must also have reassembly rules that work with the user's input. This value is saved into maxRankIndex, and if no keyword was found then it is stored as -1. If there was no keyword found, then we try to find a specific response in the memory that we can use. If there something in the memory then we set response to that statement and pop it from memoryLis. If there is nothing in memoryLis, then we get a general response using the getGeneralResponse function. This function uses the general psychobabble array, which doesn't take into account keywords.

On the other hand, if a keyword was found, we check if it is worth saving in memory. I have set the threshold to a rank of at least 3. If it is worth saving, we generate a response and save it in

memoryLis. Using the maxRankIndex, we get a specific response from the getRankedResponse function and return it.

Analyze helper functions:
getRankedResponse:

```
75      def getRankedResponse(self, statement, keyIndex):
76          aKey = self.keys[keyIndex]
77          regExList = aKey.regexLis
78          for i in range(len(regExList)):
79              pattern = regExList[i]
80              match = re.match(pattern, statement.rstrip(".!"))
81              if match:
82                  response = self.leastUsedKeyResp(i, keyIndex)
83                  tokens = self.getWords(response)
84                  if "goto" in tokens :
85                      newKeyIndex = self.getKey(tokens[1])
86                      self.getRankedResponse(statement, newKeyIndex)
87                  else:
88                      return response.format(*[self.reflect(g) for g in match.groups()])
```

Parameters: self, statement, keyIndex
Return: response

This is a recursive function that gets a response that matches a Keyword. aKey is the Keyword in the user's input statement that has the highest rank. the keyIndex was determined from getHighestRank function. The for loop iterates through the list of regular expressions associated with the keyword and checks if the user's statement matches any of them. When a match is found, we use the leastUsedKeyResp to get the least used response. Note that there are Keywords with "goto" reassembly rules.
For example, the Keyword "am" second regular expression has the reassembly rule "goto i"

```
281         am = Keyword("am", 0, r'(.*)am i(.*)', [
282             "Do you believe you are {1}?",
283             "Would you want to be {1}?",
284             "Do you wish I would tell you you are {1}?",
285             "What would it mean if you were {1}?",
286             "goto what"], r'(.*)i am(.*)',["goto i"])
```

The "goto" statement allows Keyword's to have regular expressions that use reassembly rules from a different Keyword. getRankedResponses checks if the first word in the reassembly is "goto". If it is, it get's the key that we are supposed to go to using getKey function, and recursively looks for the reassembly rule that works with the user's input. Once a reassembly rule that is not "goto" is reached, we format the response with the reflect function and return it.

getKey:

```
101     def getKey(self, keyName):
102         for i in range(len(self.keys)) :
103             if keyName == self.keys[i].name :
104                 return i
105         return -1
```

Parameters: self, keyName
Return: i

This function is a helper for getRankedResponse (above). Straightforward function that finds the index of a Keyword in keys given its name, returns -1 if no Keyword has that name.

leastUsedKeyResp:

```
88      def leastUsedKeyResp(self, patternIndex, keyIndex):
89          aKey = self.keys[keyIndex]
90          usedResp = aKey.usedResp[patternIndex]
91          leastIndex = usedResp.index(min(usedResp))
92          resp = aKey.reasmbLis[patternIndex][leastIndex]
93          self.keys[keyIndex].updateUsedResp(patternIndex, leastIndex)
94          return resp
```

Parameters: self, patternIndex, keyIndex
Return: resp

This function returns the response that has been used the least and updates the Keyword attribute, “usedResp” that keeps track of it. aKey is the Keyword that was matched and usedResp is the list of ints that corresponds to that pattern (regular expression). We get the minimum value’s index by using the min() built in function. Finally, update the Keywords usedResp attribute and return the least used response.

getGeneralResponse:

```
103     def getGeneralResponse(self, statement):
104         for pattern, responses in self.psychobabble:
105             match = re.match(pattern, statement.rstrip(".!"))
106             if match:
107                 patternIndex = self.findPatternIndex(pattern)
108                 response = self.leastUsedResponse(patternIndex)
109                 return response.format(*[self.reflect(g) for g in match.groups()])
```

Parameters: self, statement
Return: response

This function generates and returns a general response from the psychobble list. It is only called if no keywords are found in the user input statement. If a match is found with the regular expressions then the response is formatted with the reflect function and returned.

leastUsedResponse:

```
131     def leastUsedResponse(self, patternIndex):
132         leastIndex = self.usedResponses[patternIndex][1].index(min(self.usedResponses[patternIndex][1]))
133         self.usedResponses[patternIndex][1][leastIndex] += 1
134         return self.psychobabble[patternIndex][1][leastIndex]
```

Parameters: self, patternIndex
Return: response

This function determines which reassembly rule has been used the least for a certain regular expression. After finding it, it update usedResponses since we are going to use that response. The response is generated from the reassembly rules, formatted with the reflect function and returned.

findPatternIndex:

```
121     def findPatternIndex(self, pattern):
122         for i in range(len(self.psychobabble)):
123             if pattern == self.psychobabble[i][0]:
124                 return i
```

Parameters: self, pattern
Return: i

Pretty straightforward function that returns the index of the supplied "pattern" (regular expression) in the nested list from the psychobabble.

getHighestRank:

```
126     def getHighestRank(self, words, statement):
127         for keyIndex in range(len(self.keys)):
128             maxRank = self.keys[keyIndex]
129             for aWord in words:
130                 if aWord == maxRank.name and self.getRankedResponse(statement, keyIndex) is not None:
131                     return keyIndex
132         return -1
```

Parameters: self, words, statement
Return: keyIndex

This function gets the Keyword with the highest rank in the user's input statement. It returns the index of the Keyword in keys that has the highest rank or -1 if it does not exist. The loop does this by searching through the keys attribute. Once a match is found it will automatically be the Keyword with the highest rank because keys is ordered by decreasing rank.

getWords:

```
140     def getWords(self, statement):
141         exclude = set(string.punctuation)
142         s = ''.join(ch for ch in statement if ch not in exclude)
143         lowS = s.lower()
144         words = (lowS.split(" "))
145         return words
```

Parameters: self, statement
Return: words

This function formats the statement (removes punctuation, and lower case) and splits it into a list of words. The elements in the list “words” are each word from the statement (no punctuation, all lower case).