Note:

I have included an appendix at the end of the document with information regarding my general approach process, the creation of a test data frame and the link to the GitHub repo containing the source code. The text in each challenge part details the structure of the code and how it functions. I have included a test dataset for proof of concept. However, the script should work for any input you might desire, so please feel free to try it out!

Part 1

Identify task:

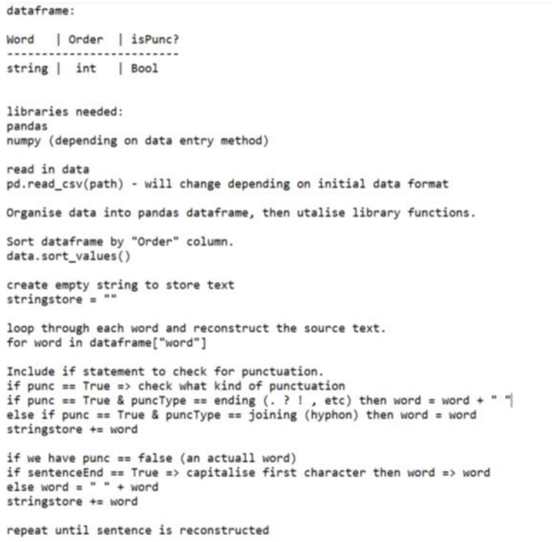
Reconstruct source text from individual text components stored within a data frame.

Bullet point key features:

- Reconstruct source text from components
- Given word and position of word in sentence.
- Told if component is punctuation

Create sudo-code:

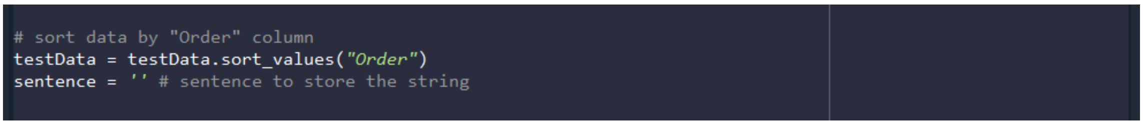
I create my sudo-code in a notepad document. For this exercise I have included an example of what this would look like when working on this task. Though I appreciate it is difficult to interpret for another person. The idea of the sudo-code is just to give me a framework to build my code off.



```
dataframe:

Word   | Order  | isPunc?
-------------------------
string |  int   | Bool


libraries needed:
pandas
numpy (depending on data entry method)

read in data
pd.read_csv(path) - will change depending on initial data format

Organise data into pandas dataframe, then utalise library functions.

Sort dataframe by "Order" column.
data.sort_values()

create empty string to store text
stringstore = ""

loop through each word and reconstruct the source text.
for word in dataframe["word"]

Include if statement to check for punctuation.
if punc == True => check what kind of punctuation
if punc == True & puncType == ending (. ? ! , etc) then word = word + " "
else if punc == True & puncType == joining (hyphon) then word = word
stringstore += word

if we have punc == false (an actuall word)
if sentenceEnd == True => capitalise first character then word => word
else word = " " + word
stringstore += word

repeat until sentence is reconstructed
```

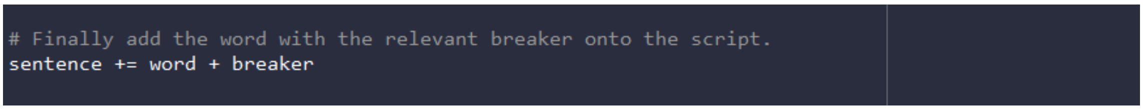
Create script:

Starting with individual words and no punctuation, we would sort the data using the “Order” column such that each word would be in the order it would appear in the sentence.



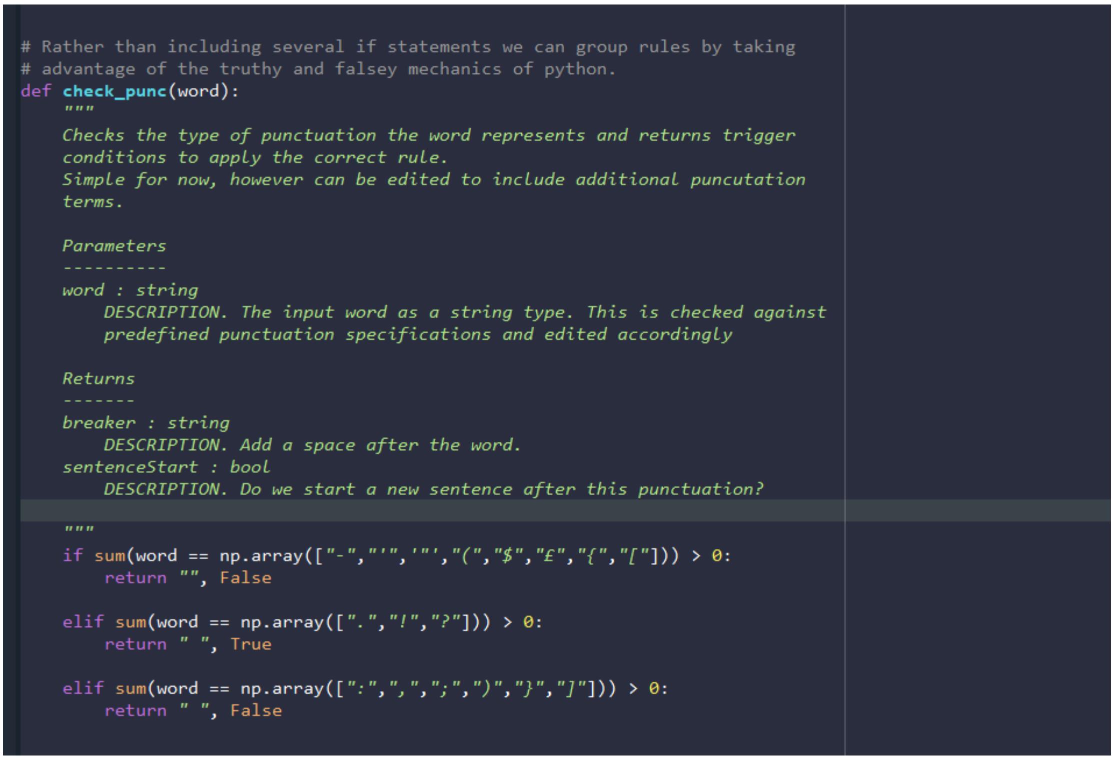
```
# sort data by "Order" column
testData = testData.sort_values("Order")
sentence = '' # sentence to store the string
```

Then we would simply append a word to a sentence using list concatenation. We could call the individual word “word” and append it to an initially empty sentence called “sentence”. We would separate each word with a “breaker”, in this case it would be a single space.



```python
# Finally add the word with the relevant breaker onto the script.
sentence += word + breaker
```

The next layer of complexity would be to consider punctuation. Depending on the punctuation we would apply a different breaker before or after the word. We can write a function to check the punctuation type and decide what breaker we should add. e.g for a period, we would have no breaker before the period. For a hyphen we would have no breaker before and after the punctuation.



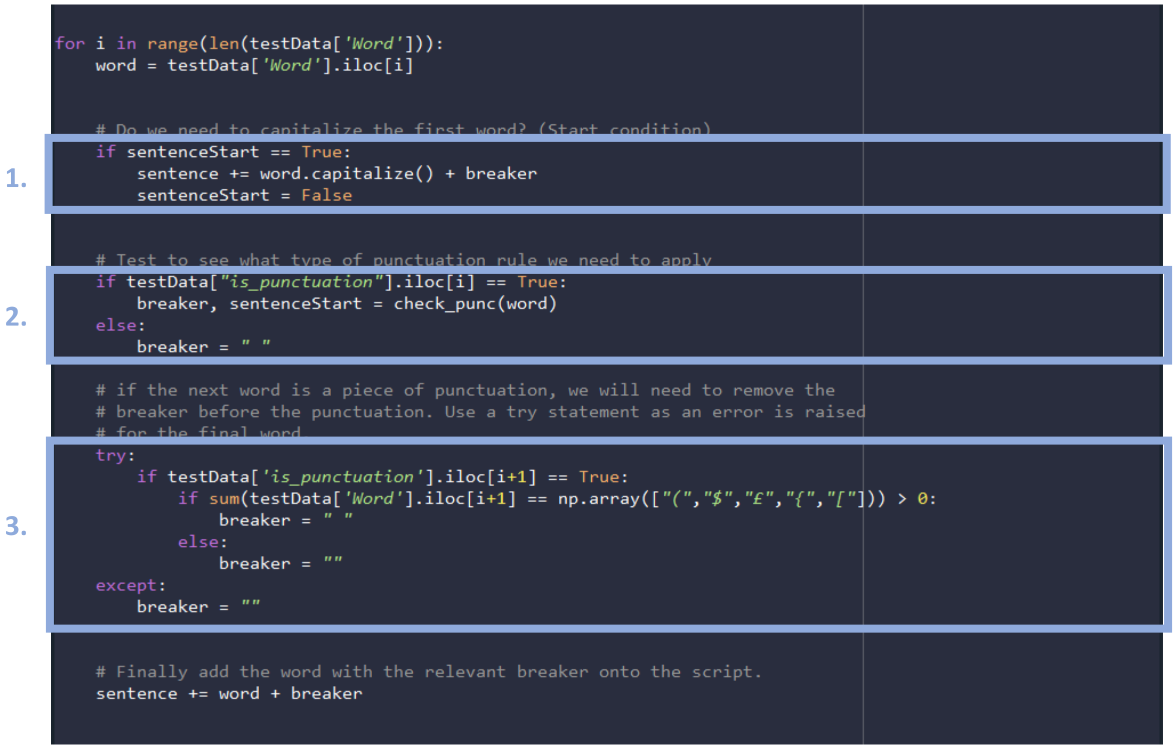
```python
# Rather than including several if statements we can group rules by taking
# advantage of the truthy and falsey mechanics of python.
def check_punc(word):
    """
    Checks the type of punctuation the word represents and returns trigger
    conditions to apply the correct rule.
    Simple for now, however can be edited to include additional puncutation
    terms.

    Parameters
    ----------
    word : string
        DESCRIPTION. The input word as a string type. This is checked against
        predefined punctuation specifications and edited accordingly

    Returns
    -------
    breaker : string
        DESCRIPTION. Add a space after the word.
    sentenceStart : bool
        DESCRIPTION. Do we start a new sentence after this punctuation?

    """
    if sum(word == np.array(["-","'",'"',"(","$","£","{","["])) > 0:
        return "", False

    elif sum(word == np.array([".","!","?"])) > 0:
        return " ", True

    elif sum(word == np.array([":",",",";",")","}","]"])) > 0:
        return " ", False
```

Looping through each word, we perform a series of checks to determine the breaker before or after the current word. The three main checks in the loop work as follows:

1. We check to see if we need to capitalize the word.

2. We check to see if the word is a piece of punctuation. If it is, we assign the relevant breaker and assign “sentenceStart” to signal if we need to capitalize the next work. If there is no punctuation, we simply set a default breaker (an empty space).

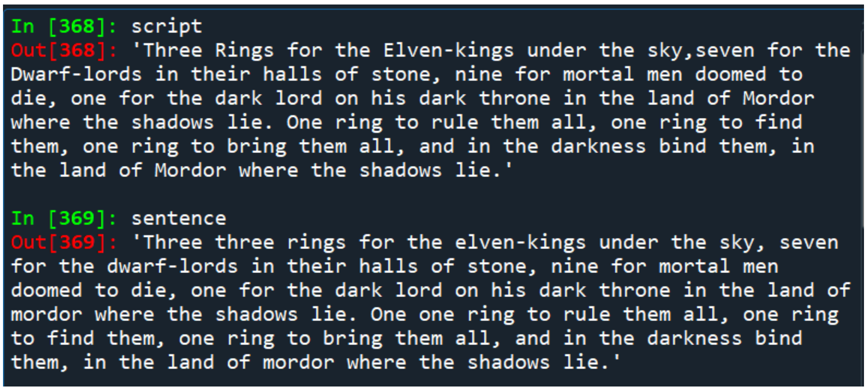
3. Depending on the type of punctuation we use, we may or may not need to include a breaker before the punctuation. E.g a period does not have a breaker between the word and the period, however a open bracket has a breaker between it and the last word. The try statement sets the breaker depending on the type of punctuation proceeding the current word. We use a try statement as trying to check the word after the last item in the dataframe throws an error.



```
for i in range(len(testData['Word'])):
    word = testData['Word'].iloc[i]


    # Do we need to capitalize the first word? (Start condition)
    if sentenceStart == True:
        sentence += word.capitalize() + breaker
        sentenceStart = False


    # Test to see what type of punctuation rule we need to apply
    if testData["is_punctuation"].iloc[i] == True:
        breaker, sentenceStart = check_punc(word)
    else:
        breaker = " "

    # if the next word is a piece of punctuation, we will need to remove the
    # breaker before the punctuation. Use a try statement as an error is raised
    # for the final word
    try:
        if testData['is_punctuation'].iloc[i+1] == True:
            if sum(testData['Word'].iloc[i+1] == np.array(["(","$","£","{","["])) > 0:
                breaker = " "
            else:
                breaker = ""
    except:
        breaker = ""


    # Finally add the word with the relevant breaker onto the script.
    sentence += word + breaker
```

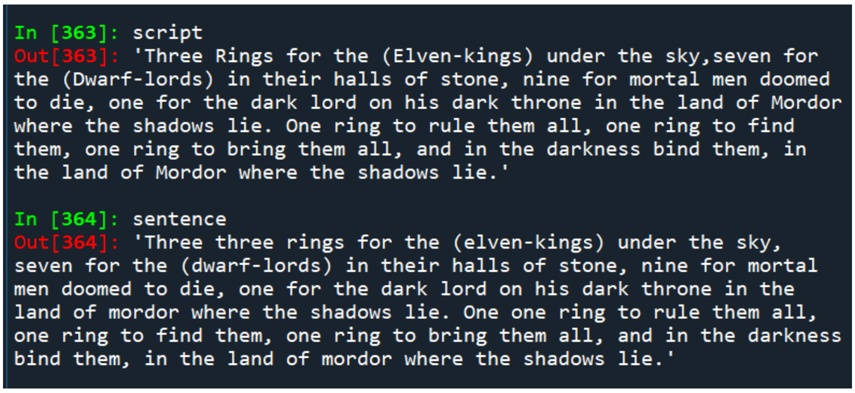
Proof of concept:

Running the input text saved to the variable “script” (see appendix A2) through the program, the output is:



```
In [368]: script
Out[368]: 'Three Rings for the Elven-kings under the sky,seven for the
Dwarf-lords in their halls of stone, nine for mortal men doomed to
die, one for the dark lord on his dark throne in the land of Mordor
where the shadows lie. One ring to rule them all, one ring to find
them, one ring to bring them all, and in the darkness bind them, in
the land of Mordor where the shadows lie.'

In [369]: sentence
Out[369]: 'Three three rings for the elven-kings under the sky, seven
for the dwarf-lords in their halls of stone, nine for mortal men
doomed to die, one for the dark lord on his dark throne in the land of
mordor where the shadows lie. One one ring to rule them all, one ring
to find them, one ring to bring them all, and in the darkness bind
them, in the land of mordor where the shadows lie.'
```

What if we include some brackets around “elven-kings” and “dwarf-lords”? How would our approach deal with brackets?



```
In [363]: script
Out[363]: 'Three Rings for the (Elven-kings) under the sky,seven for
the (Dwarf-lords) in their halls of stone, nine for mortal men doomed
to die, one for the dark lord on his dark throne in the land of Mordor
where the shadows lie. One ring to rule them all, one ring to find
them, one ring to bring them all, and in the darkness bind them, in
the land of Mordor where the shadows lie.'

In [364]: sentence
Out[364]: 'Three three rings for the (elven-kings) under the sky,
seven for the (dwarf-lords) in their halls of stone, nine for mortal
men doomed to die, one for the dark lord on his dark throne in the
land of mordor where the shadows lie. One one ring to rule them all,
one ring to find them, one ring to bring them all, and in the darkness
bind them, in the land of mordor where the shadows lie.'
```

Improvements:

Porting the script to C#, we could apply a switch statement with cases checking each punctuation type in order of common appearance. This would likely speed up this test, but benchmarking would give us a quantitative result.

Other punctuation rules could be incorporated into the check_punc function.

The nested if statements used in the loop could be condensed.

The script does not capitalise names or placenames. Incorporating a spell checker could help us determine names as well as correct potential spelling mistakes included in the input.

We should benchmark the code by testing the run time depending on text length and amount of punctuation.

Part 2.

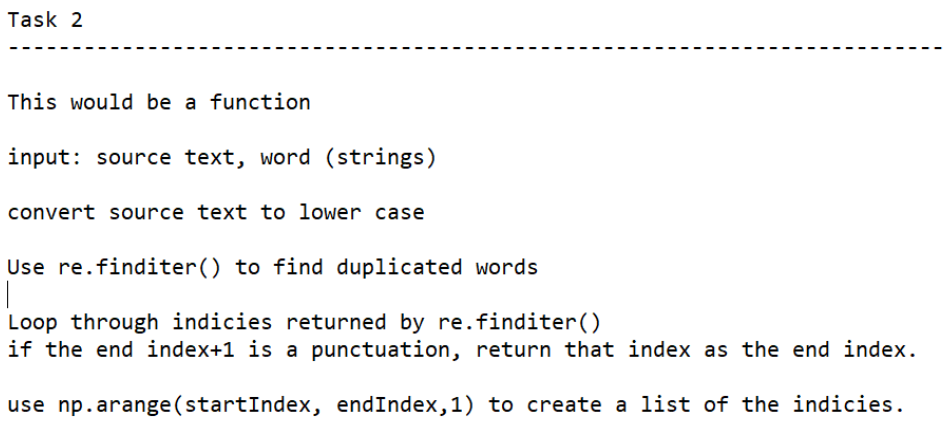
Identify task:

Return index of characters in searched word.

Bullet point key features:

- Should take input string
- Returns index, including any punctuation

Create sudo-code:

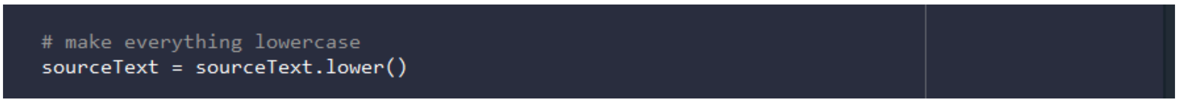


```
Task 2
---------------------------------------------------------------------------

This would be a function

input: source text, word (strings)

convert source text to lower case

Use re.finditer() to find duplicated words
|
Loop through indicies returned by re.finditer()
if the end index+1 is a punctuation, return that index as the end index.

use np.arange(startIndex, endIndex,1) to create a list of the indicies.
```

Create script:

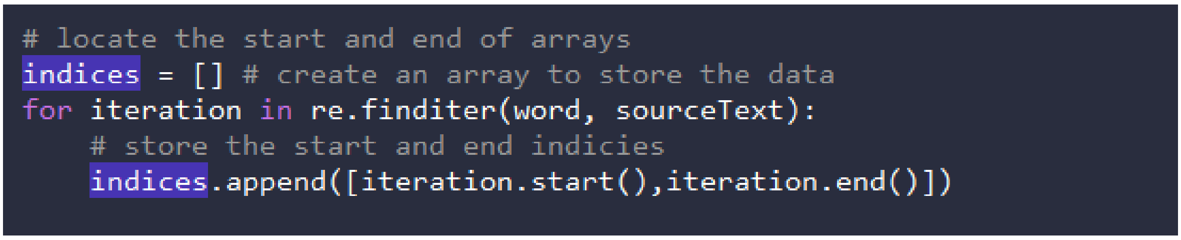
We create a function which takes an input of some source text and a word to search for.

Firstly we change all words in the source text and search term to lower case, this should remove any errors arising from capitalisation. This would be done using the built in .lower() function.



```
# make everything lowercase
sourceText = sourceText.lower()
```

Using re.finditer, which search for all occurrences of a given term, we return the start and end index of our search term “word”.

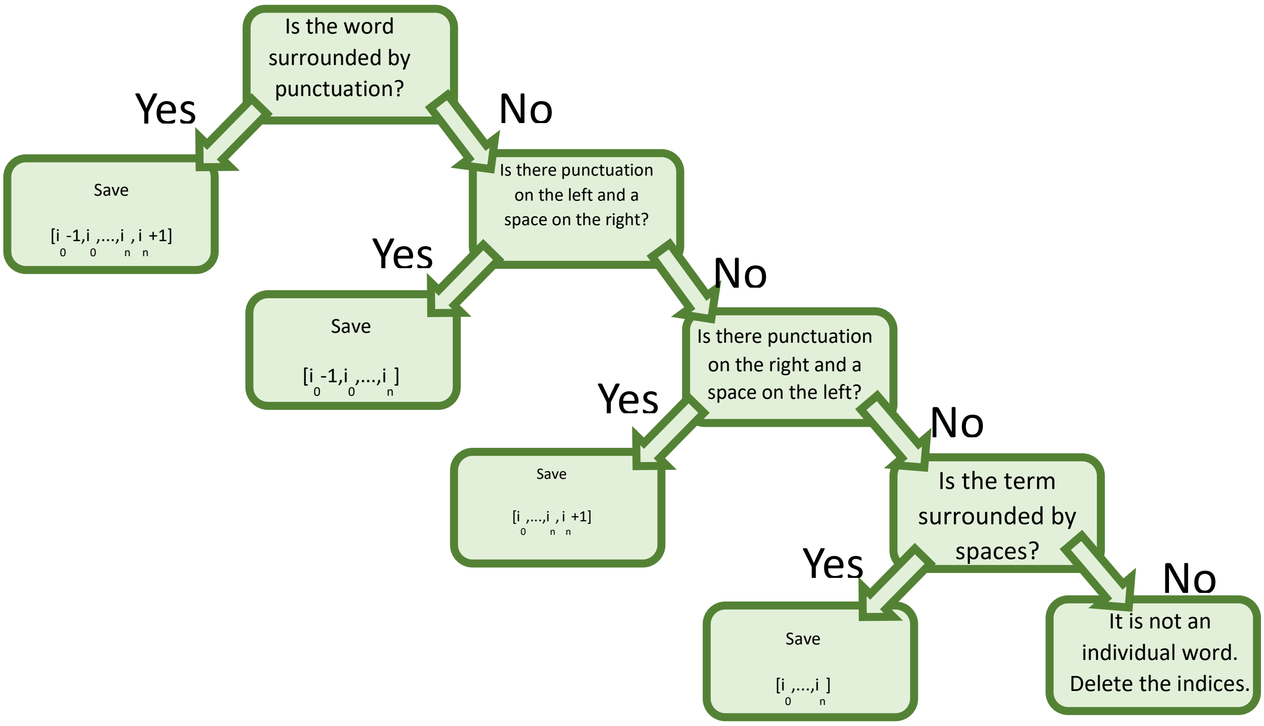


```
# locate the start and end of arrays
indices = [] # create an array to store the data
for iteration in re.finditer(word, sourceText):
    # store the start and end indicies
    indices.append([iteration.start(),iteration.end()])
```

Now we have the start and end index of the search term “word” saved into our indices array. Calling the first element in indices would return an array of two integers, e.g [4, 10]. Now we need to make a series of checks to determine if the word is surrounded by any punctuation, and also include a failsafe. The failsafe is needed as re.finditer() returns ANY instances of the search term. For example, if we search the word “ring”, the index corresponding to the 1st and 4th index of the word “bring” would also be returned. As such we need to account for this when deciding on which indices to return.

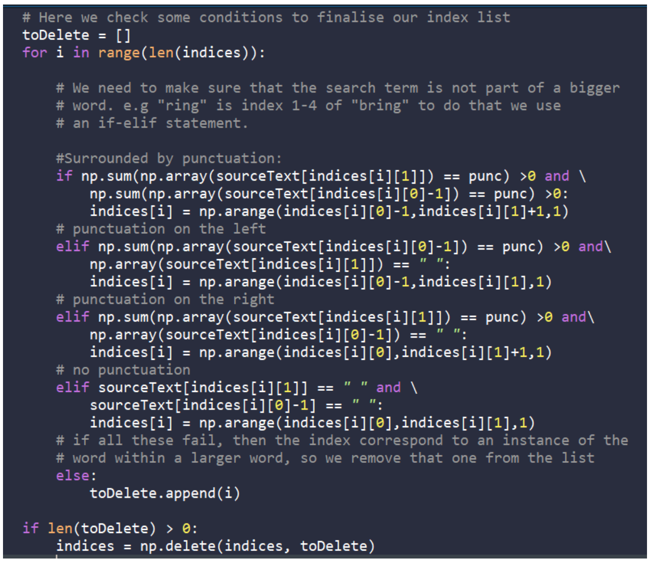
After each if/elif statement, we call np.arange(first index, last index, increment number) to create the list of indices corresponding to the returned search term.

Each section of the if statement checks the following:



Is the word surrounded by punctuation?
Yes
Save $[i_0-1,i_0,...,i_n,i_n+1]$
No
Is there punctuation on the left and a space on the right?
Yes
Save $[i_0-1,i_0,...,i_n]$
No
Is there punctuation on the right and a space on the left?
Yes
Save $[i_0,...,i_n,i_n+1]$
No
Is the term surrounded by spaces?
Yes
Save $[i_0,...,i_n]$
No
It is not an individual word. Delete the indices.

In the function, this piece of code looks like this.

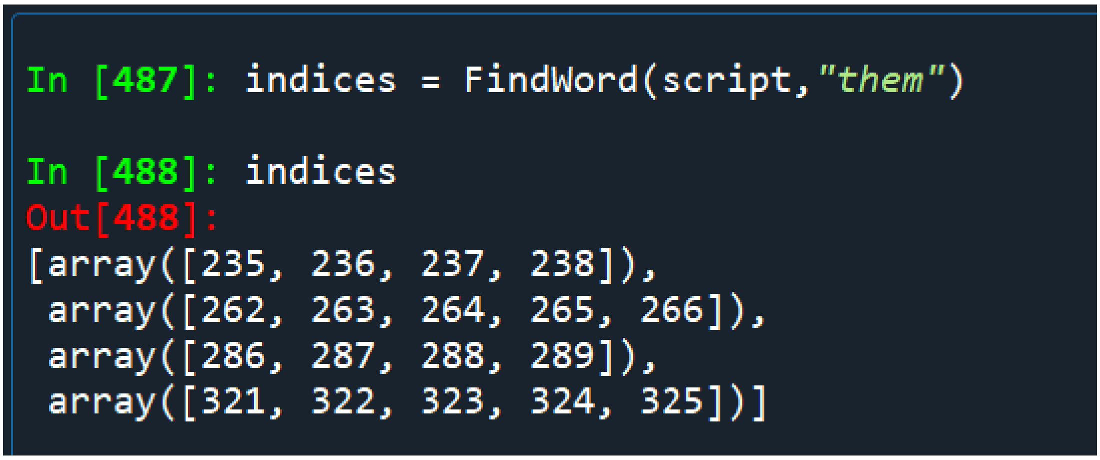


```
# Here we check some conditions to finalise our index list
toDelete = []
for i in range(len(indices)):

    # We need to make sure that the search term is not part of a bigger
    # word. e.g "ring" is index 1-4 of "bring" to do that we use
    # an if-elif statement.

    #Surrounded by punctuation:
    if np.sum(np.array(sourceText[indices[i][1]]) == punc) >0 and \
        np.sum(np.array(sourceText[indices[i][0]-1]) == punc) >0:
        indices[i] = np.arange(indices[i][0]-1,indices[i][1]+1,1)
    # punctuation on the left
    elif np.sum(np.array(sourceText[indices[i][0]-1]) == punc) >0 and\
        np.array(sourceText[indices[i][1]]) == " ":
        indices[i] = np.arange(indices[i][0]-1,indices[i][1],1)
    # punctuation on the right
    elif np.sum(np.array(sourceText[indices[i][1]]) == punc) >0 and\
        np.array(sourceText[indices[i][0]-1]) == " ":
        indices[i] = np.arange(indices[i][0],indices[i][1]+1,1)
    # no punctuation
    elif sourceText[indices[i][1]] == " " and \
        sourceText[indices[i][0]-1] == " ":
        indices[i] = np.arange(indices[i][0],indices[i][1],1)
    # if all these fail, then the index correspond to an instance of the
    # word within a larger word, so we remove that one from the list
    else:
        toDelete.append(i)

if len(toDelete) > 0:
    indices = np.delete(indices, toDelete)
```

Finally, we return the indices in an array.

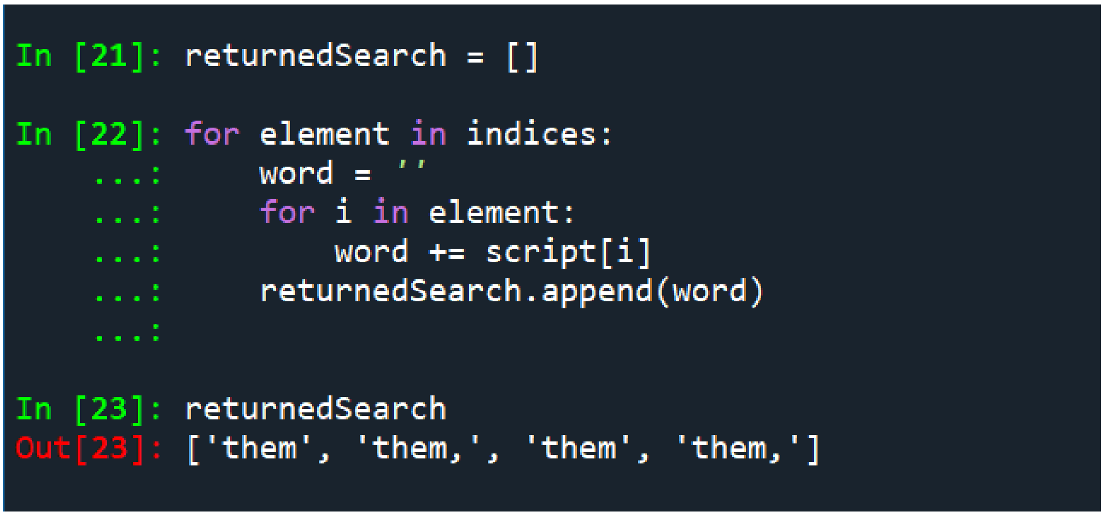
Proof of Concept:

Using our text script, we call our function, searching for the word “them”, which has occurrences including punctuation.



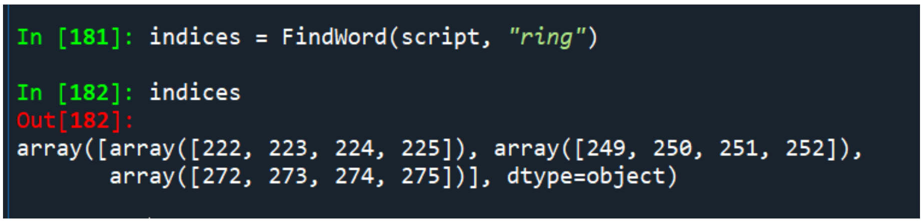
```
In [487]: indices = FindWord(script,"them")

In [488]: indices
Out[488]:
[array([235, 236, 237, 238]),
 array([262, 263, 264, 265, 266]),
 array([286, 287, 288, 289]),
 array([321, 322, 323, 324, 325])]
```

Obtaining the characters corresponding to these indices, and printing the results shows the function works as expected for the test source text.



```
In [21]: returnedSearch = []

In [22]: for element in indices:
    ...:     word = ''
    ...:     for i in element:
    ...:         word += script[i]
    ...:     returnedSearch.append(word)
    ...:

In [23]: returnedSearch
Out[23]: ['them', 'them,', 'them', 'them,']
```

Let’s check all instances of the word “ring” as we know that in the text are the words “bring” and “rings”. These should not be returned due to our criteria.



```
In [181]: indices = FindWord(script, "ring")

In [182]: indices
Out[182]:
array([array([222, 223, 224, 225]), array([249, 250, 251, 252]),
       array([272, 273, 274, 275])], dtype=object)
```

As expected, the code does not return the indices of the corresponding characters in “rings” or “bring”.

Improvements:

We include a test to check if the inputs are in a string format, or we could just convert any input to strings.

We can return information about how many instances are found.

We can omit/include punctuation consisting of brackets.

We could edit this to accept a list of words to search for and return a data frame of the results.

Part 3.

Identify Task:

How can we extend the previous work to include a bigram if the searched word was within the bigram?

Bullet point key features:

- Function updated to include list of bigrams
- Should return the index of the bigram
- Should also return punctuation?
- Word could be first or second in bigram order

Create sudo-code:

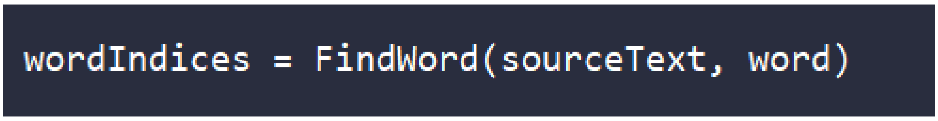
```
Task 3
---------------------------------------------------------------------------

Similar function to task 2, with the update:

Call FindWord() to obtain indices of the word we are looking for.

Check to see if word is included in any of the bigrams.

If it is, call FindWord on the bigram to get its indices.
Then check its indices against the words indices, if any of them match
we know if is part of the bigram

update the word indices list to account for the bigram.
```

Create script:

We can use the function created in part 2 to help accomplish this task.

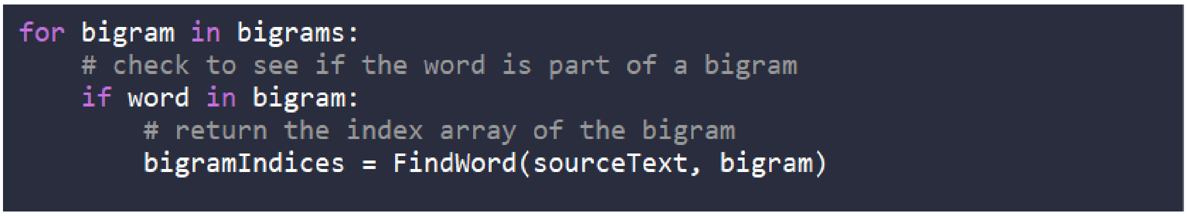
Our overall aim is to check the indices of our returned words against those of the bigrams. If they match, then we know that at those index values we have a bigram.

First we return the indices of the word we are looking for.



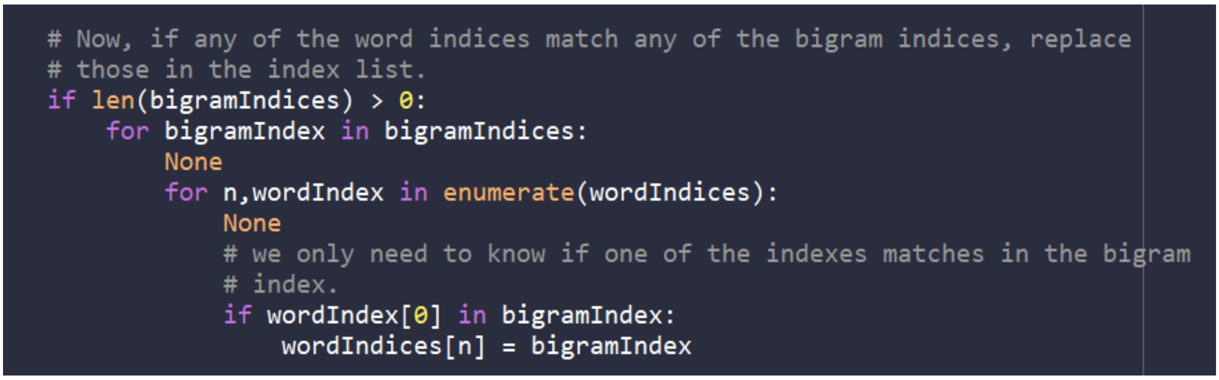
```
wordIndices = FindWord(sourceText, word)
```

Then, we loop through each bigram and check to see if our search word is in any of them. If it is, we can return the index of the bigram by using our FindWord() function. We will later compare this index array to the index array for our word search.



```
for bigram in bigrams:
    # check to see if the word is part of a bigram
    if word in bigram:
        # return the index array of the bigram
        bigramIndices = FindWord(sourceText, bigram)
```

Use an if statement just to make sure we actually have a returned bigram array, as if our words were not part of a bigram we would have a NoneType error thrown. If we do have an array of bigram indices, we then compare those to the word indices. We only use one of the word indices from each instance though, as we only need to know if they overlap in any way. If they do, we then replace that element if the word indices array with the corresponding bigram indices array.

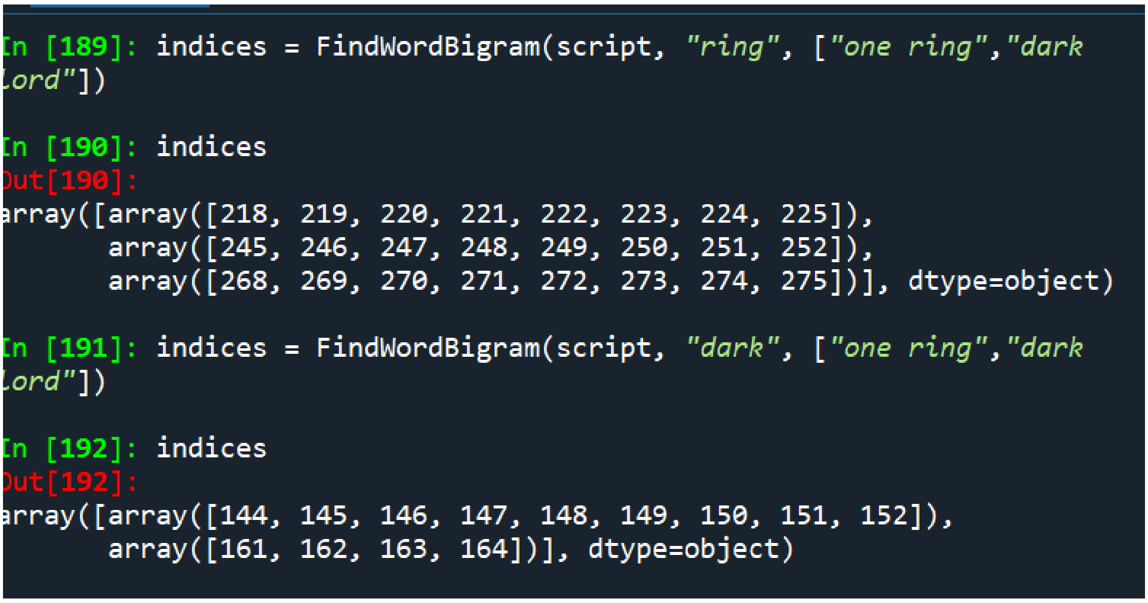


```
# Now, if any of the word indices match any of the bigram indices, replace
# those in the index list.
if len(bigramIndices) > 0:
    for bigramIndex in bigramIndices:
        None
        for n,wordIndex in enumerate(wordIndices):
            None
            # we only need to know if one of the indexes matches in the bigram
            # index.
            if wordIndex[0] in bigramIndex:
                wordIndices[n] = bigramIndex
```

We then end the function by returning the final array.

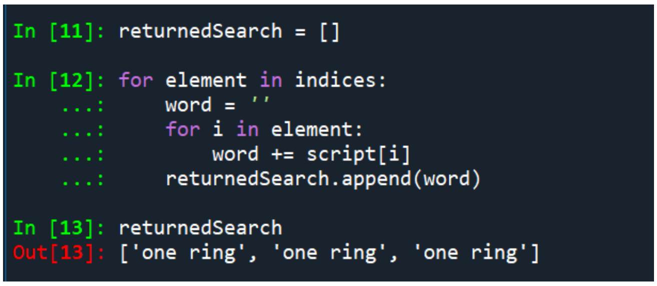
Proof of concept:

What happens if we call the word “ring”, or “dark” in our function, with the bigram [“one ring”, “dark lord”], thereby testing how the function works when our word is the first or second in the bigram.

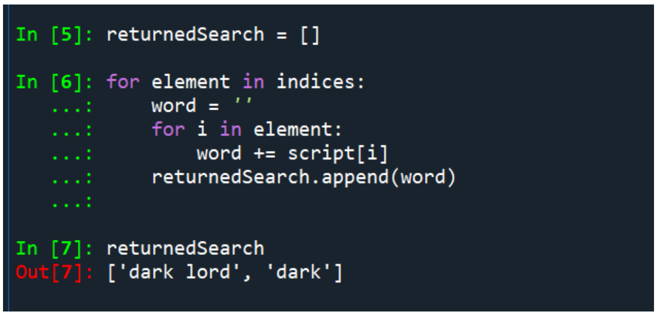


```
In [189]: indices = FindWordBigram(script, "ring", ["one ring","dark
Lord"])

In [190]: indices
Out[190]:
array([array([218, 219, 220, 221, 222, 223, 224, 225]),
       array([245, 246, 247, 248, 249, 250, 251, 252]),
       array([268, 269, 270, 271, 272, 273, 274, 275])], dtype=object)

In [191]: indices = FindWordBigram(script, "dark", ["one ring","dark
Lord"])

In [192]: indices
Out[192]:
array([array([144, 145, 146, 147, 148, 149, 150, 151, 152]),
       array([161, 162, 163, 164])], dtype=object)
```

Printing out the values shows the words we have recorded.

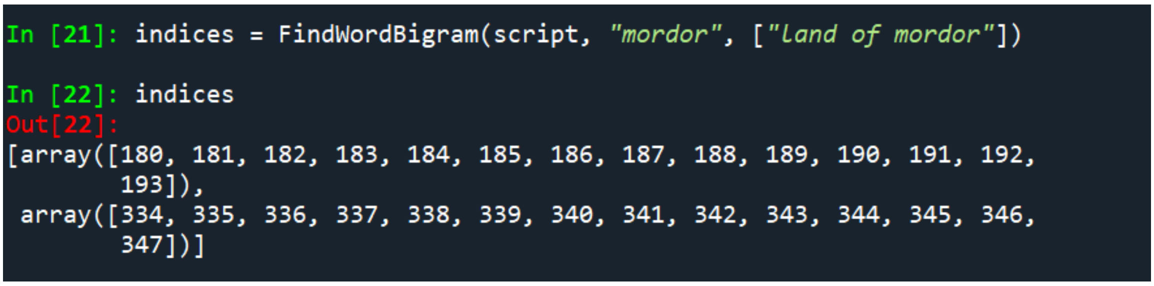


```
In [11]: returnedSearch = []

In [12]: for element in indices:
    ...:     word = ''
    ...:     for i in element:
    ...:         word += script[i]
    ...:     returnedSearch.append(word)

In [13]: returnedSearch
Out[13]: ['one ring', 'one ring', 'one ring']
```

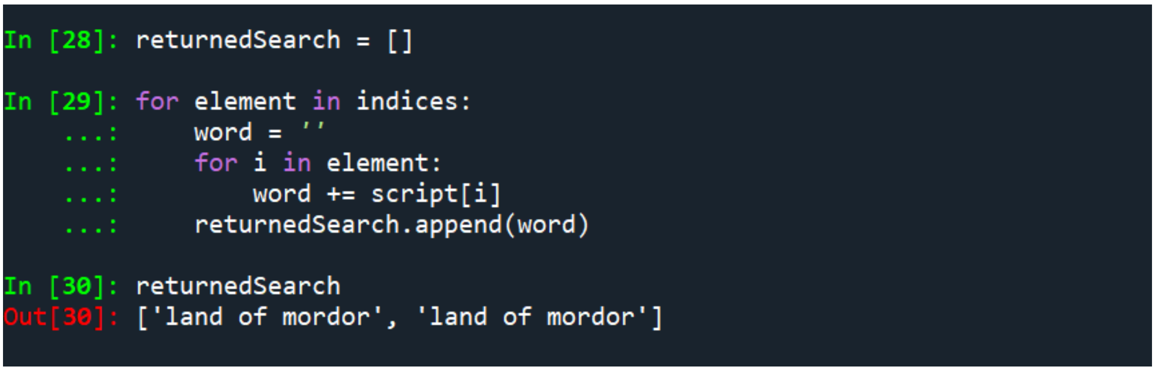


```
In [5]: returnedSearch = []

In [6]: for element in indices:
   ...:     word = ''
   ...:     for i in element:
   ...:         word += script[i]
   ...:     returnedSearch.append(word)
   ...:

In [7]: returnedSearch
Out[7]: ['dark lord', 'dark']
```

The function can take any input of the script, search term, and bigram. The way the program is written allows us to expand to search for a trigram or greater!



```
In [21]: indices = FindWordBigram(script, "mordor", ["land of mordor"])

In [22]: indices
Out[22]:
[array([180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192,
        193]),
 array([334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346,
        347])]
```



```
In [28]: returnedSearch = []

In [29]: for element in indices:
    ...:     word = ''
    ...:     for i in element:
    ...:         word += script[i]
    ...:     returnedSearch.append(word)

In [30]: returnedSearch
Out[30]: ['land of mordor', 'land of mordor']
```

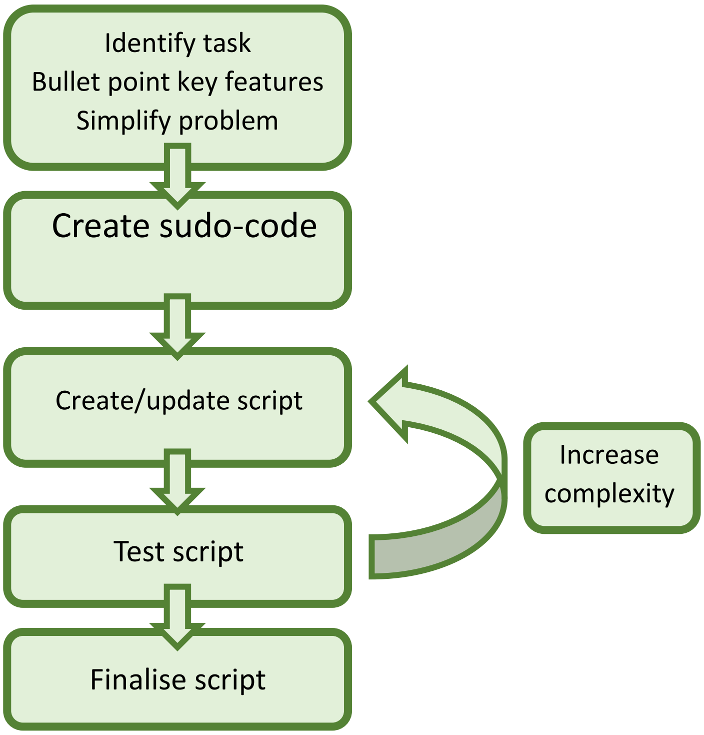
Improvements:

Other methods may be available to help vectorise the comparison between word indices and bigram indices.

Appendix

A1 Overall approach to tasks:

In any exercise I like to start with simplifying the problem, then introducing other components of the problem until I reach a full solution. My general thought/work process when tackling any problem like this is as follows:



A2 Setting up the test data:

To perform proof of concept tests on the question responses I set up a test data set in similar format to the one given. This is standard practice for any code I construct as it allows me to simultaneously test and benchmark my code whilst implementing any ideas.

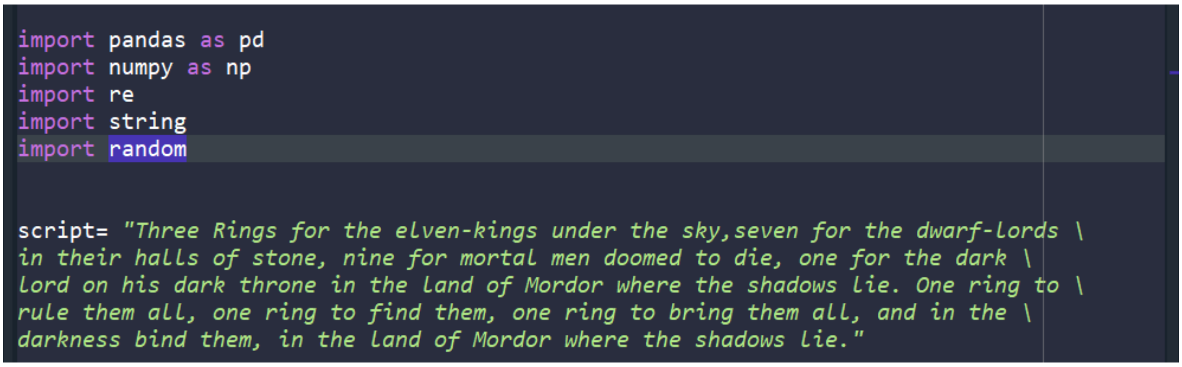
Being a massive Lord of the Rings nerd, I chose to use the one ring verse to test the code as it contains several types of punctuation, including hyphenation.

Source text:

"Three rings for the Elven-kings under the sky, seven for the Dwarf-lords in their halls of stone, nine for mortal men doomed to die, one for the dark lord on his dark throne in the land of Mordor where the shadows lie. One ring to rule them all, one ring to find them, one ring to bring them all, and in the darkness bind them, in the land of Mordor where the shadows lie."

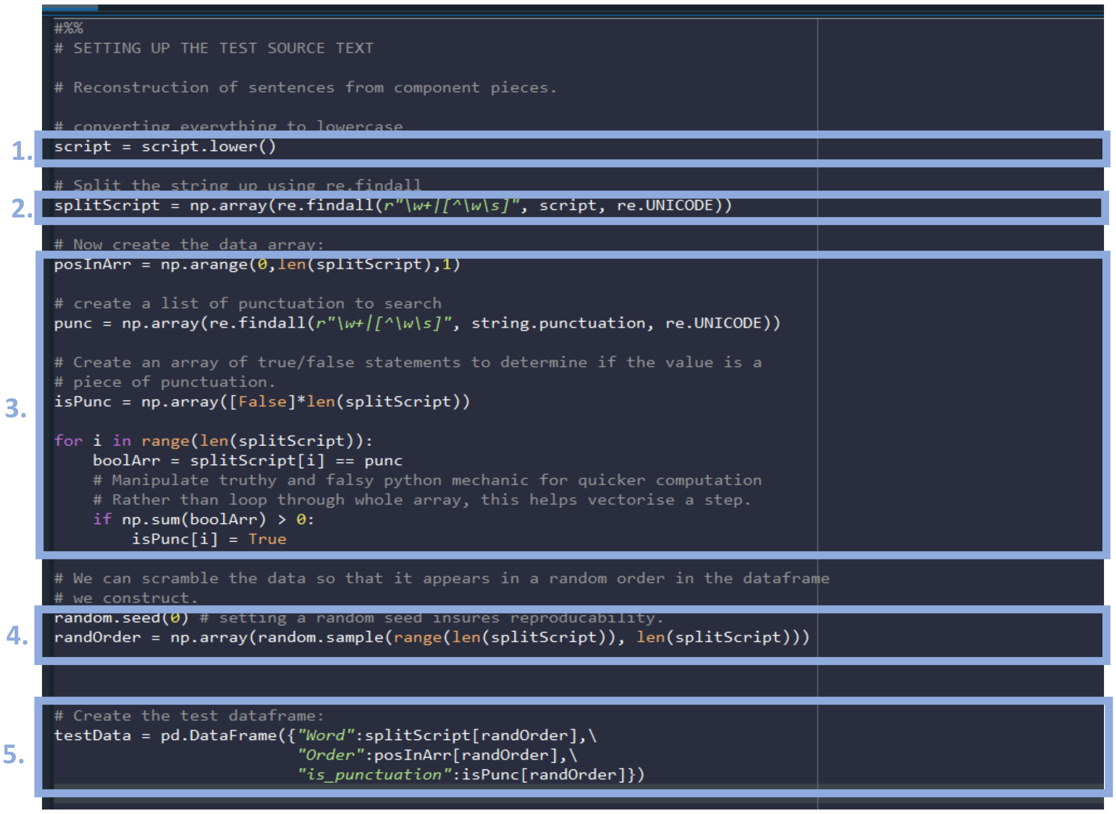
In order to generate the text I import 5 libraries which contain some useful functions:

- pandas is used to deal with data frames.
- numpy is used in vector operations and implementing them can speed up processing time.
- re is used to split strings, including punctuation. Using a regular inbuild split() function on the element "the sky," for example, would return ["the",”sky,”]. While using re would return ["the",”sky”,”,”]. This is much more akin to the example given.
- string is used to provide us with an array of different punctuation to use in our boolean identification of punctuation.
- random is used to randomise our dataset. numpy.random produces duplicate integers which would cause an error when we come to randomise the order of our data, hence why we use the random library instead. This will become more apparent later.



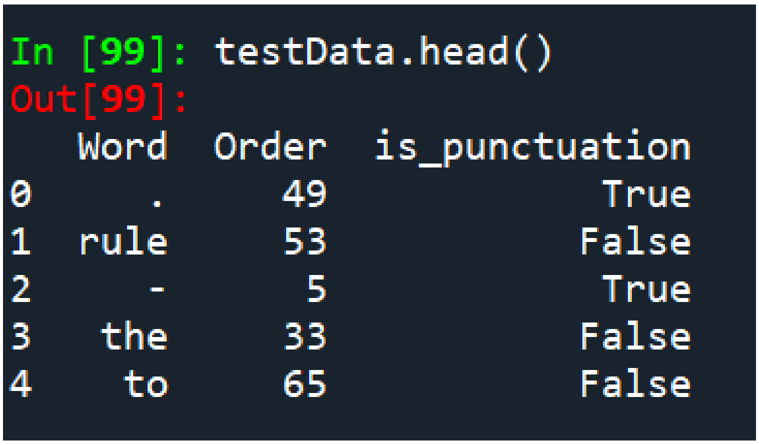
```
import pandas as pd
import numpy as np
import re
import string
import random


script= "Three Rings for the elven-kings under the sky,seven for the dwarf-lords \
in their halls of stone, nine for mortal men doomed to die, one for the dark \
lord on his dark throne in the land of Mordor where the shadows lie. One ring to \
rule them all, one ring to find them, one ring to bring them all, and in the \
darkness bind them, in the land of Mordor where the shadows lie."
```

Now we construct the source data by converting the text to lowercase (1.), splitting the data into individual characters/words (2.), labelling any punctuation (3.) and randomising the order of the data (4.) before finally storing it in a data frame (5.).



```
#%%
# SETTING UP THE TEST SOURCE TEXT

# Reconstruction of sentences from component pieces.

# converting everything to lowercase
script = script.lower()

# Split the string up using re.findall
splitScript = np.array(re.findall(r"\w+|[^\w\s]", script, re.UNICODE))

# Now create the data array:
posInArr = np.arange(0,len(splitScript),1)

# create a list of punctuation to search
punc = np.array(re.findall(r"\w+|[^\w\s]", string.punctuation, re.UNICODE))

# Create an array of true/false statements to determine if the value is a
# piece of punctuation.
isPunc = np.array([False]*len(splitScript))

for i in range(len(splitScript)):
    boolArr = splitScript[i] == punc
    # Manipulate truthy and falsy python mechanic for quicker computation
    # Rather than loop through whole array, this helps vectorise a step.
    if np.sum(boolArr) > 0:
        isPunc[i] = True

# We can scramble the data so that it appears in a random order in the dataframe
# we construct.
random.seed(0) # setting a random seed insures reproducability.
randOrder = np.array(random.sample(range(len(splitScript)), len(splitScript)))


# Create the test dataframe:
testData = pd.DataFrame({"Word":splitScript[randOrder],\
                         "Order":posInArr[randOrder],\
                         "is_punctuation":isPunc[randOrder]})
```

Calling testData.head() prints out the first 5 elements of the data frame. We see that this mimics the format given in the example.



```
In [99]: testData.head()
Out[99]:
   Word  Order  is_punctuation
0     .     49            True
1  rule     53           False
2     -      5            True
3   the     33           False
4    to     65           False
```

A3 source code

The source code used in this exercise can be accessed at the GitHub repository:

https://github.com/ChrisL1995/relativeInsightTechTest