Lesson 29. Capstone project: document similarity

After reading lesson 29, you'll be able to

- Take as input two files and determine their similarity
- Write organized code by using functions
- Understand how to work with dictionaries and lists in a real-life setting

How similar are two sentences? Paragraphs? Essays? You can write a program incorporating dictionaries and lists to calculate the similarity of two pieces of work. If you're a teacher, you could use this to check for similarity between essay submissions. If you're making changes to your own documents, you can use this program as a sort of version control, comparing versions of your documents to see where major changes were made.

The problem

You're given two files containing text. Using the names of the files, write a program that reads the documents and uses a metric to determine how similar they are. Documents that are exactly the same should get a score of 1, and documents that don't have any words in common should get a score of 0.

Given this problem description, you need to decide a few things:

- Do you count punctuation from the files or only words?
- Do you care about the ordering of the words in files? If two files have the same words but in different order, are they still the same?
- What metric do you use to assign a numerical value to the similarity?

These are important questions to answer, but when given a problem, a more important action is to break it into subtasks. Each subtask will become its own module, or a *function* in Python terms.

29.1. BREAKING THE PROBLEM INTO TASKS

If you reread the problem statement, you can see that a few natural divisions exist for self-contained tasks:

1. Get the filename, open the file, and read the information.
2. Get all the words in a file.
3. Map each word to how often it occurs. Let's agree that order doesn't matter for now.
4. Calculate the similarity.

Notice that in breaking down the tasks, you haven't made any specific decisions about implementations. You have only broken down your original problem.

Thinking like a programmer

When thinking about how to break down your problem, choose and write tasks in such a way that they can be reusable. For example, make a task that reads a filename and gives you back the file contents, as opposed to a task that reads exactly two filenames and gives you back their contents. The idea is that the function that reads one filename is more versatile and, if needed, you can call it two (or more) times.

29.2. READING FILE INFORMATION

The first step is to write a function that takes in a filename, reads the contents, and gives you back the contents in useable form. A good choice for the

functions to open the file by using the filename given, read the entire contents into a string, and return that string. When the function is called on the name of a file, it'll return a string with all the file contents.

Listing 29.1. Reading a file

```
def read_text(filename):
    """
    filename: string, name of file to read
    returns: string, contains all file contents
    """
    inFile = open(filename, 'r')
    line = inFile.read()
    return line

text = read_text("sonnet18.txt")
print(text)
```

- ***1* Docstring**
- ***2* Python function to open the file by using the filename**
- ***3* Python function to read all contents as a string**
- ***4* Return string**
- ***5* Function call**

After you write a function, you should test and, if necessary, debug it. To test this function, you need to create a file with contents. Create an empty text document in the same folder where you have your .py file for this lesson. Populate the text file with content and save it; I used Shakespeare's "Sonnet 18." Now, in the .py file, you can call the function with

```
print(read_text("sonnet18.txt"))
```

When you run the file, the console should print the entire contents of the file.

Thinking like a programmer

The point of writing functions is to make your life easier. Functions should be self-contained pieces of code that you need to debug only once but that you can reuse many times. When you're integrating more than one function, you need to debug only the way they interact as opposed to debugging the functions themselves.

29.3. SAVING ALL WORDS FROM THE FILE

Now you have a function that returns a string containing all the contents of a file. One giant string isn't helpful to a computer. Remember that Python works with objects, and a large string containing a bunch of text is one object. You'd like to break this large string into parts. If you're comparing two documents, a natural breakdown of the string would be to separate it into words.

Thinking like a programmer

When faced with a task, you'll often need to decide which data structures (types) to use. Before beginning to code, think about each data type you've learned about and decide whether it's an appropriate one to use. When more than one may work, pick the simplest one.

This task of breaking down a string will be done using a function. Its input is a string. Its output can be one of many things. With more coding practice, you'll more quickly recognize when to use certain object types and why. In this case, you'll separate all the words in the string into a list, with each word being an element in the list. Listing 29.2 shows the code. It first does a bit of cleanup by replacing newlines with a space and removes all special characters. The expression `string.punctuation` is a string itself whose value is the set of all the punctuation characters that a string object could have:

```
"!#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~
```

After the text has been cleaned up, you use the `split` operation to split the string on the space character and give back a list of all words (because all words are separated by a space).

Listing 29.2. Finding words from a string

```
import string                                          1
def find_words(text):
    """
    text: string
    returns: list of words from input text
    """
    text = text.replace("\n", " ")                     2
    for char in string.punctuation:                    3
        text = text.replace(char, "")                  4

    words = text.split(" ")                            5
    return words                                       6

words = find_words(text)                               7
```

- **1 Brings in functions related to strings**
- **2 Replaces newlines with a space**
- **3 Uses preset punctuation characters from string**
- **4 Replaces punctuation characters with the empty string**
- **5 Makes a list of all words by using a space separator**
- **6 Returns list of words**
- **7 Function call**

Thinking like a programmer

Before running a function on large input files, try it on a smaller test file with a couple of words. That way, if anything goes wrong, you don't have to look through hundreds of lines to figure out what's wrong.

You run this function on the text file sonnet18.txt:

```
Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date:
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimmed,
And every fair from fair sometime declines,
By chance, or nature's changing course untrimmed:
But thy eternal summer shall not fade,
Nor lose possession of that fair thou ow'st,
Nor shall death brag thou wander'st in his shade,
When in eternal lines to time thou grow'st,
So long as men can breathe, or eyes can see,
So long lives this, and this gives life to thee.
```

If you type the following code, you'll get back a list of all the words in the console:

```
print(find_words(text))
```

This prints the following list for sonnet18.txt:

```
['Shall', 'I', 'compare', ... LIST TRUNCATED ..., 'li
```

29.4. MAPPING WORDS TO THEIR FREQUENCY

Now that you have a list of words, you have a Python object with which you can work more in-depth to analyze its contents. At this point, you should be thinking about how to find the similarity between two documents. At the very least, you'll probably want to know the quantity of each word in the document.

Notice that when you created the list of words, the list contained all the words, in order, from the original string. If there were duplicate words, they were added in as another list element. To give you more information about the words, you'd like to pair up each word to how often it occurs. Hopefully, the phrase *pair up* led you to believe that a Python dictionary would be an appropriate data structure. In this particular case, you'll be building a frequency dictionary. The following listing shows you the code to accomplish this.

Listing 29.3. Making a frequency dictionary for words

```
def frequencies(words):
    """
    words: list of words
    returns: frequency dictionary for input words
```

```
    for word in words:                          2
        if word in freq_dict:                   3
            freq_dict[word] += 1                4
        else:                                   5
            freq_dict[word] = 1                 6
       return freq_dict                         7

freq_dict = frequencies(words)                  8
```

- **1 Initially empty dictionary**
- **2 Looks at each word in list**
- **3 If word already in dictionary…**
- **4 …adds one to its count**
- **5 Word not in dictionary yet**
- **6 Adds word and sets its count to 1**
- **7 Returns dictionary**
- **8 Function call**

A frequency dictionary is a useful application of dictionaries in this problem. It maps a word to the number of times you see it in the text. You can use this information when you compare two documents.

29.5. COMPARING TWO DOCUMENTS BY USING A SIMILARITY SCORE

Now you have to decide which formula you'd like to use to compare two documents, given the number of times each word occurs. To begin with, the formula doesn't need to be too complicated. As an initial pass, you can use a simple metric to make the comparison and see how well it does. Suppose these steps will calculate the score, by using a running sum over each word:

- Look for a word in both frequency dictionaries (one for each document).
- If it's in both, add the difference between the counts. If it appears in only one of them, add the count for that one (effectively adding the difference between the count from one dictionary and 0 from the other one).
- The score is the division between the total difference and the total number of words in both documents.

After coming up with a metric, it's important to do a sanity check. If the documents are exactly the same, the difference between all the word counts in both frequency dictionaries is 0. Dividing this by the total number of words in both dictionaries gives 0. If the documents don't have any words in common, the difference summed up will be "total words in one document" + "total words in other document." Dividing this by the total number of words in both documents gives a ratio of 1. The ratios make sense except that you want documents that are exactly the same to have a ratio of 1, and ones that are completely different to have a ratio of 0. To solve this, subtract the ratio from 1.

Listing 29.4 shows the code to calculate the similarity, given two input dictionaries. The code iterates over the keys of one dictionary; it doesn't matter which one, because you'll iterate over the other dictionary in another loop.

As you're going through the keys of one dictionary, you check whether the key is also in the other dictionary. Recall that you're looking at the value for each key; the value is the number of times the word occurs in one text. If the word is in both dictionaries, take the difference between the two frequency counts. If it isn't, take the count from the one dictionary in which it exists.

After you finish going through one dictionary, go through the other dictionary. You no longer need to look at the difference between the two dictionary values because you already counted that previously. Now you're just looking to see whether any words in the other dictionary weren't in the first one. If so, add up their counts.

Finally, when you have the sum of the differences, divide that by the total number of words in both dictionaries. Take 1 minus that value to match the original problem specifications for scoring.

Listing 29.4. Calculate similarity given two input dictionaries

```
def calculate_similarity(dict1, dict2):
    """
    dict1: frequency dictionary for one text
    dict2: frequency dictionary for another text
    returns: float, representing how similar both tex
    """
    diff = 0
    total = 0

    for word in dict1.keys():
```

```
            else:
                diff += dict1[word]

        for word in dict2.keys():
            if word not in dict1.keys():
                diff += dict2[word]

        total = sum(dict1.values()) + sum(dict2.values())
        difference = diff / total
        similar = 1.0 - difference

        return round(similar, 2)
```

- ***1*** **Iterates over words in one dictionary**
- ***2*** **Word is in both dictionaries**
- ***3*** **Adds the difference in frequencies**
- ***4*** **Word doesn't appear in the other dictionary**
- ***5*** **Adds the entire frequency**
- ***6*** **Iterates over words in other dictionary**
- ***7*** **Counted word in both dictionaries; looks only at words not in dict1**
- ***8*** **Adds entire frequency**
- ***9*** **Total number of words in both dictionaries**
- ***10*** **Divides difference by total number of words**
- ***11*** **Subtracts difference from 1**
- ***12*** **Rounds to 2 decimal places and returns score between 0 and 1**

The function returns a float number between 0 and 1. The lower the number, the less similar the documents are, and vice versa.

29.6. PUTTING IT ALL TOGETHER

The final step is to test the code on text files. Before using your program on two separate files, do a sanity check: first, use the same file as both texts to check that the score you get is 1.0, and then use the sonnet file and an empty file for the other to check that the score you get is 0.0.

Now, use Shakespeare's "Sonnet 18" and "Sonnet 19" to test two pieces of work, and then modify "Sonnet 18" by changing the word *summer* to *winter* to see if the program found them to be almost exactly the same.

The text of "Sonnet 18" was shown earlier. Here's the text for "Sonnet 19":

```
Devouring Time, blunt thou the lion's paws,
And make the earth devour her own sweet brood;
Pluck the keen teeth from the fierce tiger's jaws,
And burn the long-lived phoenix in her blood;
Make glad and sorry seasons as thou fleet'st,
And do whate'er thou wilt, swift-footed Time,
To the wide world and all her fading sweets;
But I forbid thee one most heinous crime:
O! carve not with thy hours my love's fair brow,
Nor draw no lines there with thine antique pen;
Him in thy course untainted do allow
For beauty's pattern to succeeding men.
Yet, do thy worst old Time: despite thy wrong,
My love shall in my verse ever live young.
```

The following listing opens two files, reads their words, makes the frequency dictionary, and calculates their similarity.

Listing 29.5. Code to run the document similarity program

```
text_1 = read_text("sonnet18.txt")
text_2 = read_text("sonnet19.txt")
words_1 = find_words(text_1)
words_2 = find_words(text_2)
freq_dict_1 = frequencies(words_1)
freq_dict_2 = frequencies(words_2)
print(calculate_similarity(freq_dict_1, freq_dict_2))
```

When I run the program on "Sonnet 18" and "Sonnet 19" the similarity score is 0.24. It makes sense that it's closer to 0 because they're two different pieces of work. When I run the program on "Sonnet 18" and my modified "Sonnet 18" (with three instances of the word *summer* changed to *winter*), the score is 0.97. This also makes sense because the two pieces are almost the same.

29.7. ONE POSSIBLE EXTENSION

words, called *bigrams*, and save them in a list. Looking at bigrams instead of words can improve your program because pairs of words often give a better indication of similarity in languages. This could lead to a more accurate setup and a better model of written text. If you want, you could also use a mixture of bigrams and words when you calculate a similarity score.

SUMMARY

In this lesson, my objective was to teach you how to write a program that reads in two files, converts their content to a string, uses a list to store all the words in a file, and then makes a frequency dictionary to store each word and the number of times it occurred in a file. You compared two frequency dictionaries by counting the differences between the word counts in each dictionary to come up with a score for how similar the files were. Here are the major takeaways:

- You wrote modular code by using functions that could be reused.
- You used lists to store individual elements.
- You used a dictionary to map a word to its count.

Recommended / Playlists / History / Topics / Settings / Get the App / Sign Out

PREV
Lesson 28. Aliasing and copying lists and dictionaries

NEXT
Unit 7. Making your own object types by using object-orient…