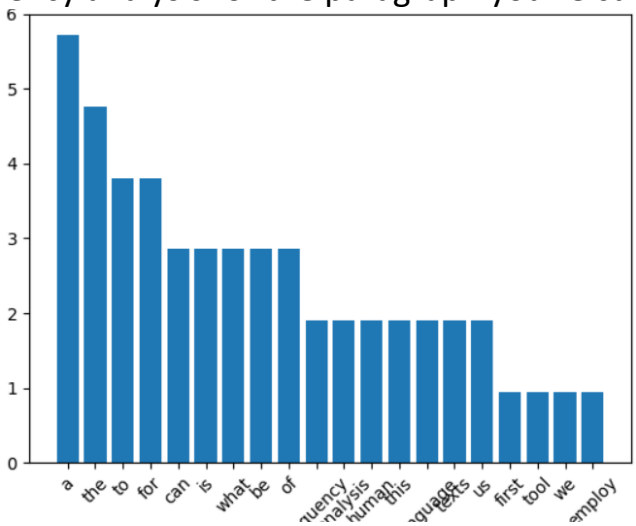
Book Analysis

Leo Tolstoy's "War and Peace" contains more instances of the word "the" than Jack London's "Call of the Wild" contains words altogether. One is a rolling epic tale of French-Russian politics and history, while the other is the story of a sled dog prospecting for gold in the Canadian wilderness. But there are some remarkable similarities in these texts that we can only discover through a mathematical analysis…

Word Frequency

The first tool we can employ is a frequency analysis. What would be an impossibly time-consuming (not to mention error-prone) task for a human is the work of microseconds for a computer. This can give insights into the author's use of language that are often too subtle even for human literary analysts to identify. This method can be used to verify the authenticity and authorship of ancient texts, but in addition to showing us what may be unique about a text, it also shows us what is common across all texts within a given language. Here's a frequency analysis for the paragraph you're currently reading:



20 most common words (accounting for 50% of the text)

Recap: writing to a text file

Firstly, we will need to revisit how to deal with text files in Python. The best method is to use a **with open** statement. Try this simple example, which generates a tables test:

```
with open("Tables.txt", "w") as out_file:
    for i in range(2, 13):
        for j in range(i, 13):
            out_file.write(f"{i} x {j} = \n")
```

The **with** block takes care of opening the document for you, but also ensuring it is saved and closed properly before the code terminates.

The **"w"** stands for 'write' mode: use **"r"** instead to open documents in 'read-only' mode, or **"a"** to append your text to the end without overwriting existing data in a file. The **write** command adds a line to the document, and the special **\n** represents a (hidden) 'new line' character. See what happens if you miss it out.

Recap: reading from a text file

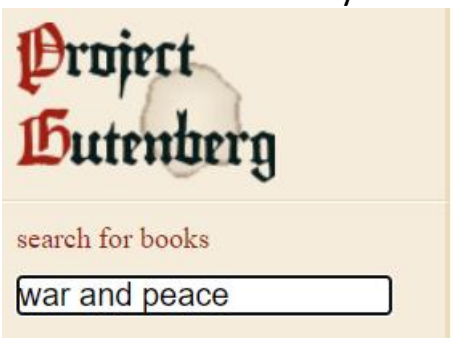
```
with open("Tables.txt", "r") as in_file:
    lines = in_file.readlines()

for line in lines:
    print(line.strip())
```

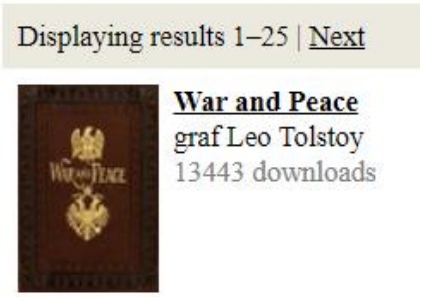
Our document is opened in 'read-only' mode, and the **readlines** method is used to build a list (which I've called **lines**) containing the text string of each line (including the new line character, \n). **.strip()** gives a version without the \n character.

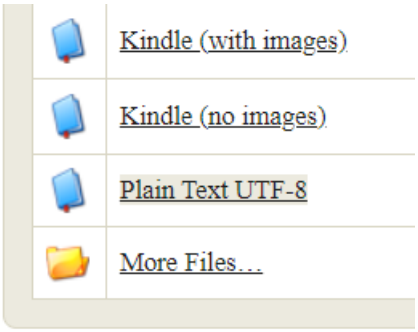
Word Frequency Analysis with Python

Fetching a book

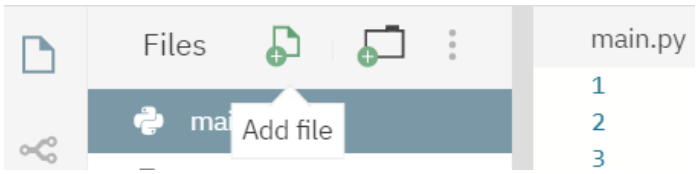
Project Gutenberg has a vast collection of books which are in the public domain, made readily available in convenient formats, including .txt. Head over to Gutenberg.org and search for War and Peace (or Pride and Prejudice, Call of the Wild, or whatever). Once you've found the book you want, right-click the Plain Text version to download the file:







Finally, upload your document to your repl either by dragging it into the files pane on the left or selecting 'upload file':



Parsing the file contents

The phrase 'parse' means to break down and interpret the different elements of something. In programming, it is usually used to mean the act of processing and interpreting the input, and putting it in a format we can more easily use. We begin by pulling out each line of text and creating a list within the program's memory:

```
1   with open("war_and_peace.txt", "r") as in_file:
2     lines = in_file.readlines()
3
4   text = [line.strip() for line in lines]
5
6   print(text[55529])
7
8   print(text[1460:1465])
```

This opens the file in 'read-only' mode, and produces a list of every line of text, **lines**.
A new list, **text**, is then created from those lines, but removing the new line character \n. You can then print out a line or two to see what you're dealing with.

Extracting the words

```
10   all_words = []
11   for line in text:
12     words = line.split()
13     for word in words:
14       all_words.append(word.lower())
```

Looping through every line, we use the string method **split** to turn a string into a list of separate strings, using spaces by default to indicate where breaks occur.

We also take this opportunity to convert all characters to lower case using the **lower** method – this will make it easier to count words later, since "Of" is the same word as "of".

Cleaning the words

```
16   abc = "abcdefghijklmnopqrstuvwxyz"
17   cleaned_words = []
18   for word in all_words:
19     chars = [char for char in word if char in abc]
20     cleaned = "".join(chars)
21     cleaned_words.append(cleaned)
```

For each word, only letter characters are included. This code uses the string method **join** to do the opposite of **split**: turning a list into a single string.

Counting the words

If we were counting words by hand, we might start with a blank sheet, and write down every new word we encounter. If we come across a word that is already in our list, we could just add a tally mark to it. We can do a very similar thing using a Python dictionary:

```
23  word_dict = {}
24  for word in cleaned_words:
25      if word in word_dict:
26          word_dict[word] += 1
27      else:
28          word_dict[word] = 1
```

This loop goes through every word in our list of cleaned words, checks to see if it is already in our dictionary and, if so, adds to the tally already there. If not, it makes a new entry with a tally of 1.

*As you perform each of these steps, it is a really good idea to throw a few more **print** statements in to make sure the result is what you are expecting.*

A note on sorting

Python can sort lists in the traditional way: **sorted(a)** will return a sorted version of the list **a**, for instance, provided the elements are all comparable (strings in case-sensitive alphabetical order, ints and floats in ascending order of size by default)*:

```
phonetic = ["Echo", "Alpha", "Bravo", "charlie", "Delta"]
print(sorted(phonetic))
```

```
['Alpha', 'Bravo', 'Delta', 'Echo', 'charlie']
```

This list is sorted according to the rules of strings: upper before lower case.

```
numbers = [5, 2, 1.3, -0.7, -12]
print(sorted(numbers))
```

```
[-12, -0.7, 1.3, 2, 5]
```

Python can compare ints and ints, floats to floats, and even ints to floats.

```
strings = ["hi", "1", "two", "4", "33"]
print(sorted(strings))
```

```
['1', '33', '4', 'hi', 'two']
```

Python can compare alphanumeric strings, but in alphabetical order.

```
mixture = [1, 2, "hi", "4"]
print(sorted(mixture))
```

```
    print(sorted(mixture))
TypeError: '<' not supported between instances of 'str' and 'int'
```

Strings and ints are not compatible, however, since they can't be compared.

There are two optional key-word arguments ("kwargs") that can be passed to **sorted**: a **key** and **reverse**. Setting **reverse** to **True** puts the list in reverse order:

```
numbers = [5, 2, 1.3, -0.7, -12]
print(sorted(numbers, reverse=True))
```

```
[5, 2, 1.3, -0.7, -12]
```

Numbers will now be in descending order; strings in reverse alphabetical order.

The **key** argument is the cleverest: it can be used to specify your own sorting criteria:

```
phonetic = ["Echo", "Alpha", "Bravo", "charlie", "Delta"]
print(sorted(phonetic, key = lambda x: x.lower()))
```

```
['Alpha', 'Bravo', 'charlie', 'Delta', 'Echo']
```

The **key** is used here to compare the *lowercase* version of each element.

*Notice that we are using an anonymous function here. Lambda functions allow us to quickly define a function on the fly: for an input **x** we generate an output **x.lower()**.*

If we can choose how objects are compared, we could even compare tuples!

```
nations = [("USA", 331), ("China", 1439), ("India", 1380)]
print(sorted(nations, key = lambda x: x[1], reverse=True))
```

```
[('China', 1439), ('India', 1380), ('USA', 331)]
```

The **key** is used here to compare the *second* element of each tuple.

*If you're curious, the algorithm used by Python behind the scenes is *timsort* – it does remarkably well both on random lists and nearly-sorted lists (since these are very common in real world applications). You can read about it at https://bugs.python.org/file4451/timsort.txt

Sorting the words

Now that we know how to sort tuples, we can use the same idea to generate a sorted version of a dictionary. Note that dictionaries have no inherent order, so the object we create will be a list of tuples. We want something like this:

```
words = {"if": 2, "she": 4, "said": 7, "what": 2, "you": 1}
print(sorted(words.items(), key = lambda x: x[1], reverse=True))
```

```
[('said', 7), ('she', 4), ('if', 2), ('what', 2), ('you', 1)]
```

The **sorted** function is asked to work on **words.items()** which is effectively a list of tuples, each being a (key, value) pair from the dictionary **words**. The sorting kwarg **key** instructs it to compare tuples with one another by looking at the second element (index 1), and finally the kwarg **reverse** ensures the list produced is in descending order, giving the words with the highest frequency first. Note that the result will be a list of tuples, not a dictionary. Let's incorporate this into our code:

```
30    words_in_order = sorted(word_dict.items(), key = lambda x: x[1], reverse=True)
```

This generates for us a list of tuples of the form (word, frequency) with the most common word appearing first. If you try printing the first few elements, you'll get this:

```
[('the', 34565), ('and', 22151), ('to', 16709), ('of', 14989), ('a', 10495)]
```

Saving and displaying our results

Last but not least, we can save the results in a text file, or represent them in a bar graph:

```
32    with open("war_and_peace results.txt", "w") as out_file:
33        n = len(cleaned_words)
34        for (w, f) in words_in_order:
35            out_file.write(f"{w}: {f} ({round(100 * f / n,2)}%)\n")
```

This writes the word, frequency and percentage in a file.

If we want a bar graph, we'll need to start by importing **matplotlib.pyplot**:

```
1    import matplotlib.pyplot as plt
```

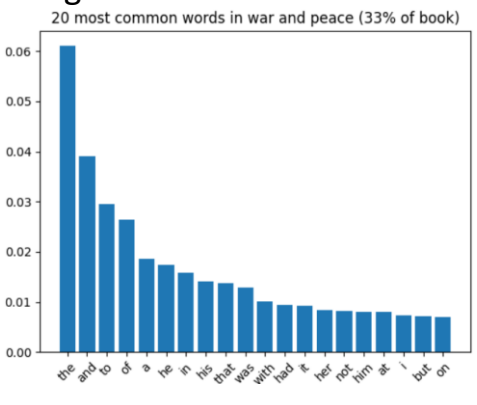
…

```
40    words = [w for (w,f) in words_in_order]
41    props = [f / len(cleaned_words) for (w,f) in words_in_order]
42    limit = 20
43    plt.bar(words[:limit], props[:limit])
44    total = int(100 * sum(props[:limit]))
45    plt.title(f"{limit} most common words in war and peace ({total}% of book)")
46    plt.xticks(rotation=45)
47    plt.savefig(f"Word Frequencies.png")
48    print("Done.")
```

The code above does a few things: firstly, it generates separate lists for the **matplotlib** function **bar** to work with, from the list of tuples we created. Next, we choose how many words to display (you can play around with this, but 20 fits nicely enough on the graph). Then we plot a bar graph, add a suitable title, rotate the labels on the x axis to make them more legible, and save the file. You should get something like this:

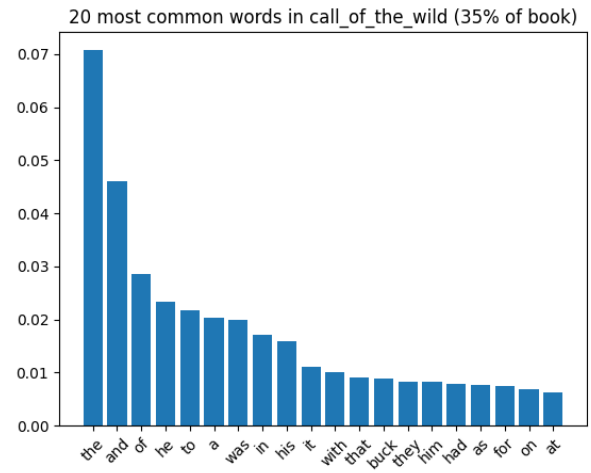
war_and_peace results.txt

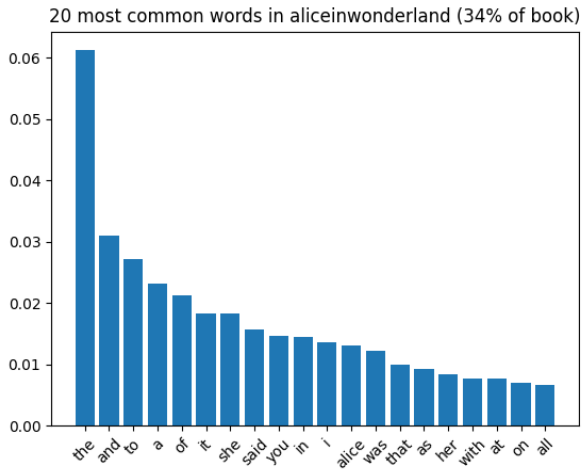
```
1    the: 34565 (6.1%)
2    and: 22151 (3.91%)
3    to: 16709 (2.95%)
4    of: 14989 (2.65%)
5    a: 10495 (1.85%)
6    he: 9812 (1.73%)
7    in: 8927 (1.58%)
8    his: 7965 (1.41%)
9    that: 7806 (1.38%)
10   was: 7328 (1.29%)
```

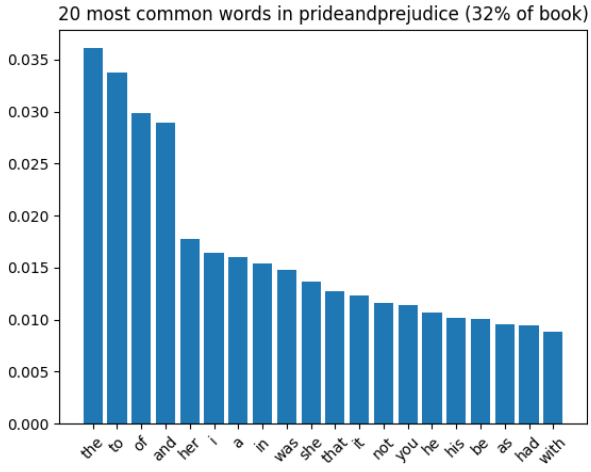


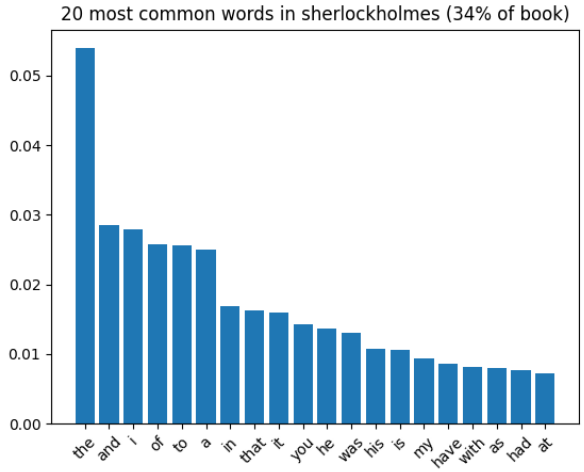
More books

There's nothing stopping you downloading a bunch of books from Project Gutenberg and performing the same analysis on all of them. You'll probably want to put your code into a loop or a function so you can quickly run through a whole list of books. For comparison, here are a few:









Remarkably, a third of every book you've ever read is made up of just twenty words. And while there are interesting differences between these four (can you tell which was written in the third person?) the overall similarities are striking. I'm as mystified as you are that Jane Austen uses the word 'the' only about half as often as anyone else, to be honest.

Even better if

- If you compile a list of the top 100 words from a bunch of different books, can you perform a rank correlation analysis to see how similar they are?
- Cryptoanalysis may rely on letter frequency analysis, especially if a substitution cipher has been used (replacing every instance of each letter with a different one). Can you adapt your code to count the number of As, Bs, Cs, etc that occur, and draw a graph for that?
- Predictive text often makes use of not simply the most common words we use, but the most common next word used. Can you adapt your program to look for consecutive pairs or triplets of words?
- Look up **Zipf's Law** – according to Mr Zipf, the second most common word in any text should appear half as often as the most common, the next a third as often, etc. How closely does this distribution match the data you've collected?