Lab 4: A composite scripting task

Student: abcde123

Student: abcde123

2. Introduction

In this lab, you will develop a small program that doubles as a somewhat reusable module and a command-line script. It will read a text file and produce some simple statistics about the contents, along with data that can be used to create a crude Markov text generator. The main focus is on Python as a scripting language, program structure and naming, and the usage of proper data structures.

At the end of this lab text, there are some technical hints.

3. A simple text extraction script

First, download the file `shakespeare.txt` to your home directory. Your first task is to produce a command-line script `text_stats.py` which reads the file and prints some basic information about its contents. **Read the entire lab before starting to code**.

Requirements

Software

- [ ] You may only do this with modules installed on the IDA system (python3, standard library, numpy and a few more). That is, you can't download for instance nltk (Natural Language Toolkit) just yet.

Running the script

- [ ] It should be invoked from the shell/terminal by `./text_stats.py <filename>` (so, for example `./text_stats.py shakespeare.txt`). On Windows, this might look like `python text_stats.py <filename>` (see below).
- [ ] Even if you write it and try it out on Windows, we should be able to run it on the IDA computers like above.
- [ ] If the user provides no such argument, an error message should be printed (not just an exception raised).
- [ ] If the user provides a second argument (a file name), the output should be written to that file.
- [ ] If the file does not exist, you need to print "The file does not exist!" (ie not just crash).
- [ ] The module shoudn't try to read any files or command-line arguments if you only import it as a module (eg write `import text_stats` in another Python file).

The tasks above will require you to explore some simple error handling, and how to read command line scripts. You do not need to use any special libraries for the parsing, but it is allowed. This part will help you practice some simple scripting, which can come in useful when eg writing scripts for batch data processing.

Output

- [ ] It should print, in some readable fashion, a frequency table for alphabetic letters, ordered from the most common to the least. Just writing something like `print(words)` is *not* enough (unless words is an instance of a specially constructed class with a suitable **repr** method, but this is overkill).
- [ ] It should print the number of words that the text contains, according to some definition of words.
- [ ] It should print the number of unique words that the text contains, according to this definition.
- [ ] It should also print the five most commonly used words, their frequency *and* the words that most commonly follow them (max three per word, ordered from most common to least, including information about the number of occurrences). So if the word "my" is followed by "king" five times, "lord" three times and "supper" ten times, this part of the printout might consist of

```
my (40 occurrences)
-- supper, 10
-- king, 5
-- lord, 3
```

Notice that you might want your program to generate all successors, but the printing part to print only the "top (at most) three".

Note: what data structures to use, and how to use them is not something with a single simple answer. You need to look at the context, what we'll need the data for, which operations will happen a lot, and what kind of lookups need to be fast. The requirements above will give you some hint as to this (once you've started coding).

Program structure and style

- [ ] The program shall have a logical structure. Functions should have a clear task that you can easily explain in and of themselves. For instance, "generate_text(...)" (with some appropriate parameters) is better than "compute_rest_data()" that is based around the knowledge that the function will be run in order to compute the rest of the data (though it doesn't make any sense by itself).
- [ ] If the user imports the module (writes `import text_stats` in another Python script), they should be able to call functions in the module which can extract the information above *in some useful format*. For example, one such function might take a string or a list of strings and return a frequency mapping that can be used for other purposes.
- [ ] It should be easy to convince oneself of what your program does, spot mistakes or figure out which definitions you use (eg what a word is). This might involve some commenting, this as a rule of thumb it should not be necessary (names, structure etc should be clear enough).

This part is to get you into the habit of writing somewhat reusable code, and code where you can reuse interesting parts for other projects or experiments without resorting to too much copy-paste.

Time and efficiency

- [ ] The program shall take less than a minute to run on the IDA computers.

Part of the point of this lab is to figure out efficient data structures and use them, and figure out what data needs to be saved. The point of this task is *not* to learn how to shave milliseconds off your computations, but to get *some* basic intuition about builtin datastructures (and beyond). The skills to make this run in seconds rather than minutes might translate to having *every attempt* (out of very many) when you try out small-scale experiments in other courses or projects run in seconds rather than minutes. Or minutes instead of hours. (This is of course not to say that there will be more efficient dedicated solutions, or that extremely big data will need a different skill set.)

Additional questions

In addition, you should provide a brief answer to the following questions:

- [ ] In what way did you "clean up" or divide up the text into words (in the program; the text files should be left unaffected)? This does not have to be perfect in any sense, but it should at least avoid counting "lord", "Lord" and "lord." as different words.
- [ ] Which data structures have you used (such as lists, tuples, dictionaries, sets, ...)? Why does that choice make sense? You do not have to do any extensive research on the topics, or try to find exotic modern data structures, but you should reflect on which of the standard data types (or variants thereof) make sense. If you have tried some other solution and updated your code later on, feel free to discuss the effects!

Some hints

These also serve as suggestions as to what this actually tests:

- This is a composite task which contains several requirements, most of which are fairly simple in and of themselves. They might look overwhelming at first. However, consider:
  - Which ones can you develop or research separately?
  - Which ones actually require you to read a file for instance?
  - What might be good functions, in terms of treating the data, and what are only to do with input-output? Try to figure this out before diving in to the coding. It can also be helpful to write well-named function stubs (see for instance the `read_data` example of lecture 2), to help you figure out what kinds of data and transformations the program will need.
- Which modules might be useful? The Python ecosystem is huge, but you should only need to use standard library modules!
- Can you avoid performing costly operations many times?
- What would be good data structures to use?
- You might want to try the logic out on smaller data sets to start with, but do make sure that your solution "scales" at least to the complete-works-of-Shakespeare level (which of course is tiny by many measures). Project Gutenberg and Projekt Runeberg (sw text) might be helpful.

4. Text generator

You might have noticed that we could take the "what is the next word" structure above and interpret the numbers as probabilities. If "my" has 40 successors of which 10 are "supper", we can interpret this as the probability that "my" is succeeded by "supper" is 25% (10/40). We can then turn this on its head and generate new text (often quite silly) by walking around randomly in this system. (If there are no successors, we interpret this as a terminal node.)

Your task is to write a script `generate_text.py` which takes **three** arguments: a file name of a text file, a starting word and a maximum number of words. It then generates a new text by roughly the following algorithm:

```
cur_word <- the given word.
msg <- cur_word
until there are no successors of cur_word, or the maximum number of words has been
 picked:
    cur_word <- random choice of the successors of cur_word, weighted by how likely
cur_word is to be succeeded by it
    msg <- msg + " " + cur_word
print msg
```

Apart from generating at the very least interesting-looking text, notice how you can (ideally) use both `random` and your module from the previous task to make this script practically identical to the pseudo code above. All of the work of reading the file, generating the successor-structure etc can be done by calls to functions in the module above.

Try it out using a few different input files and numbers of words! For instance `./generate_text.py shakespeare.txt king 500`, `./generate_text.py nilsholgersson.txt akka 1500` (if you have downloaded a file called nilsholgersson.txt), ...

Additional requirements on generate_text

- [ ] The program should not take over roughly a minute on IDA computers to generate a 500 word text.
- [ ] Generating a 2000 word text should not take a lot longer than generating a 500 word text.

The last requirement might seem rather fuzzy, but might suggest (once you've started coding) which data should be computed when, and which data might be computed "on the fly", when required. Sometimes the former makes sense, sometimes the latter.

Technical hints

- As an editor, you can use any *text* editor. Do not try to write in Word or the like! If you don't have a favourite on Linux, gedit might be a good start (or pluma, which comes with some distributions). IDA systems also have Atom installed. The lecturer's preference is Emacs, which is ancient but very flexible. Unless you want to spend some time learning it, we would advise against it. On Windows, there is a variety of IDE:s, but regular Notepad++ is often enough.
- If you are testing this on windows, you might have to invoke the script by `python text_stats.py <filename>` (assuming that the Python directory is in PATH). The initial comment in the Python file should still be there, so that you can run it on IDA systems. If you think that this is complex, use IDA systems (eg via ThinLinc).
- You might be able to create a proper script file which you still cannot run. This might mean that you have missed setting it as executable, eg by `chmod u+x text_stats.py`.
- Naturally, you still need the initial `#!/...` comment.
- Unfamiliar with working with the terminal? Ask your lab assistant. Some particularly useful commands might be `man` (read manual page for a command, exit with q), `ls`, `cd`, `mv`, `cat`.

Handing in the scripts

Compress the files into a .zip or .tar.gz and email the archive to your assistant. The file manager on the Linux computers offers a simple right-click option for this.

Acknowledgments

Thanks to Project Gutenberg (https://www.gutenberg.org/) for providing the free ebook/out-of-copyright text.

Lab text by Anders Märak Leffler (2019, 2020), for 732A74.