# Homework 3: Working with Files

Due February 23, 11:59 pm

Worth 15 points

Instructions on writing and submitting your homework can be found on the course webpage at `http://pages.stat.wisc.edu/~kdlevin/teaching/Spring2022/STAT679/hw_instructions.html`. *Failure to follow these instructions will result in lost points.* Please direct any questions the instructor.

**Read this first.** A few things to bring to your attention:

1. Start early! If you run into trouble or you have questions, it's best to find those problems well in advance, not in the hours before your assignment is due!
2. If you have clarifying questions or you run into issues, please do not email the instructor directly. Instead, post to the discussion board so that your classmates can benefit as well if they have the same question.
3. **Make sure you back up your work!** I recommend, at a minimum, doing your work in a Dropbox folder or, better yet, using `git`, which is well worth your time and effort to learn.

## 1 Basics of Files: Reading and Writing (5 points)

This problem will give you some practice with the basics of reading and writing text files in Python.

1. Write a function `duplicate_lines`, called as `duplicate_lines(infile, outfile, n)`, where `infile` and `outfile` are strings specifying file names and `n` is a non-negative integer. Calling `duplicate_lines(infile, outfile, n)` should overwrite `outfile` so that it is a copy of `infile`, except that each line of `infile` is repeated `n` times. So, for example, if the file `infile` has the contents

```
It is a capital mistake to theorize before one has data.
Insensibly one begins to twist facts to suit theories,
instead of theories to suit facts.
```

then, with `n=2`, the file `outfile` should read

```
It is a capital mistake to theorize before one has data.
It is a capital mistake to theorize before one has data.
Insensibly one begins to twist facts to suit theories,
Insensibly one begins to twist facts to suit theories,
instead of theories to suit facts.
instead of theories to suit facts.
```

The case `n=1` should result in `outfile` being identical to `infile` (when testing your code, you might try using `diff` to verify that this is the case), and the case `n=0` should result in `outfile` being an empty file (you can use `cat` or `wc` to check this).

You may assume that every line of `infile` ends with a newline. Your program should perform type checking and raise an appropriate error in the event that either of `infile` or `outfile` are not strings or the event that `n` is not a non-negative integer. `duplicate_lines` should open `infile` for reading and open `outfile` for writing (and raise an error if either of these operations fail). Your program should overwrite `outfile` if it already exists.

2. There are at least two obvious ways (to me, anyway) of implementing `duplicate_lines`:

   - **Option 1:** Open `infile` for reading and `outfile` for writing. Read the lines of `infile` one at a time, and save them in the entries of a list, say, `line_list`. Then, once we have finished reading `infile`, iterate over the entries of `line_list`, writing each one `n` times to `outfile`.
   - **Option 2:** Open `infile` for reading and `outfile` for writing. Read the lines of `infile` one at a time, and each time we read a line, write it to `outfile` `n` times immediately.

   There is at least one good reason to prefer Option 2 over Option 1. What is that reason? **Hint:** if `infile` is especially large, what happens to the list `line_list`?

   Define two new functions, `duplines_storage` and `duplines_direct`, that have the same signature (i.e., take the same arguments) and perform the same function as `duplicate_lines`. `duplines_storage` should follow the design in Option 1 and `duplines_direct` should follow the design outlined in Option 2. If you happened to follow one of these designs in your solution for `duplicate_lines`, you are free to copy-paste your code, but make sure you are still defining both `duplines_direct` and `duplines_storage`. There is no need to perform error checking in these two functions.

3. To compare `duplines_direct` and `duplines_storage`, we need a file to duplicate. Write a function `generate_test_file` that takes three arguments: a string `filename`, a non-negative integer `nlines` and another non-negative integer `nchars`, in that order. Your function should open `filename` for writing, and write `nlines` lines of text, with each line containing `nchars` characters, each chosen uniformly at random from the lower-case letters (`a,b,c,...,z`). Your function should raise an appropriate error in the event that `filename` is not a string, or in the event that either of `nlines` or `nchars` is not a non-negative integer. **Hint:** you can choose a random integer in a given range using the `randint` function in the `random` module.[1] Then you can use that random integer to index into `string.ascii_lowercase`, which is a string consisting of the lower-case letters of the English alphabet.

[1] See `https://docs.python.org/3/library/random.html#random.randint` for documentation.

4. Now, let's use the Python `time` module to compare how fast `duplines_direct` and `duplines_storage` are. Write a function called `time_trial` that takes three non-negative integers, `nlines`, `nchars` and `ntimes` as its arguments, and returns a 2-tuple of floats. Your function should

   (a) Use `generate_test_file` to generate a test file of `nlines` lines of text, each containing `nchars` random characters.

   (b) Run `duplines_storage` on the test file, with `ntimes` as the `n` argument passed to `duplines_storage`, using the Python `time` module to measure how long the program took.

   (c) Run `duplines_direct` on the test file, with `ntimes` as the `n` argument, using the Python `time` module to measure how long the program took.

   (d) Return a tuple `(t1, t2)` where `t1` is a float representing the number of seconds that it took to run `duplines_storage` on the test file and `t2` is a float representing the number of seconds that it took to run `duplines_direct` on the test file.

   **Hint:** to time an operation, you can modify the following code (I assume that you have imported the `time` module with `import time`).

   ```
   start_time = time.time()
   do_operation()
   end_time = time.time()
   t1 = end_time - start_time
   ```

5. Use `time_trial` to compare the runtimes of `duplines_storage` and `duplines_direct`. By choosing suitable values of `nlines`, `nchars` and `ntimes`, you should be able to see a notable difference between the two runtimes. What do you think would account for this difference? **Note:** there is no need to make a plot or anything like that for this. Just run the program for a few different choices of arguments, describe the difference you see, and try to explain the difference. **Note also:** you may need to set the arguments to be quite large before you see a difference between the two different methods, especially on newer, faster machines or if you have many other processes running on your computer (e.g., many browser tabs open).

## 2 Counting Word Bigrams (5 points)

In your previous homework, you wrote a function for counting character bigrams. Now, let's write a function for counting word bigrams. That is, for each pair of words, say, `cat` and `dog`, we want to count how many times the word "cat" occurred immediately before the word "dog". We will represent this bigram by a tuple, `('cat', 'dog')`. For our purposes, we will ignore all newlines, punctuation and capitalization in our counting. So, as an example, the fragment of poem,

> Half a league, half a league,
> Half a league onward,
> All in the valley of Death
> Rode the six hundred.

includes the bigrams `('half', 'a')` and `('a', 'league')` both three times, the bigram `('league', 'half')` appears twice, while the bigram `('in', 'the')` appears only once.

1. Write a function `count_bigrams_in_file` that takes a filename as its only argument. Your function should read from the given file, and return a dictionary whose keys are bigrams (given in the tuple form above), and values are the counts for those bigrams. Again, your function should ignore punctuation, spaces, newlines and capitalization. The strings in your key tuples should be lower-case. Your function should use a try-catch statement to raise an error with an appropriate message to alert the user in the event that the given file cannot be opened, and a different error in the event that the provided argument isn't a string at all. **Hint:** you will find the Python function `str.strip()`, along with the string constants defined in the string documentation (`https://docs.python.org/3/library/string.html`), useful in removing punctuation. **Hint:** be careful to check that your function handles newlines correctly. For example, in the poem above, one of the `('league', 'half')` bigrams spans a newline, but should be counted nonetheless. **Note:** be careful that your function does not accidentally count the empty string as a word (this is a common bug if you aren't careful about splitting the input text). Solutions that merely delete "bad" keys from the dictionary at the end will not receive full credit.

2. Download the file `WandP.txt` from the course webpage: `http://pages.stat.wisc.edu/~kdlevin/teaching/Spring2021/STAT679/WandP.txt`. This is an ASCII copy of all of Tolstoi's novel *War and Peace*. Run your function on this file, and pickle the resulting dictionary in a file called `WandP.bigrams.pickle`. Please include this file in your submission, along with `WandP.txt`, so that we can run your notebook directly from your submission.

3. We say that word $A$ is *collocated* with word $B$ in a text if words $A$ and $B$ occur immediately one after another (in either order). That is, words $A$ and $B$ are collocated if and only if either of `A B` or `B, A` are present in the text. Write a function `collocations` that takes a filename as its only argument and returns a dictionary. Your function should read from the given file (raising an appropriate error if the file cannot be opened or if the argument isn't a string at all) and return a dictionary whose keys are all the strings appearing in the file (again ignoring case and stripping away all spaces, newlines and punctuation) and the value of word $A$ is a Python set object[2] containing all the words collocated with $A$. Again using the poem fragment above as an example, the string `'league'` should appear as a key, and should have as its value the set `{'a', 'half', 'onward'}`, while the string `'in'` should have the set `{'all', 'the'}` as its value. **Hint:** we didn't discuss Python sets in lecture, because they are essentially just dictionaries without values. See the documentation for more information.

4. Run your function on the file `WandP.txt` and pickle the resulting dictionary in a file called `WandP.colloc.pickle`. Please include this pickle file in your submission.

## 3 Implementing `wc` in Python (5 points)

On UNIX/Linux systems, the shell command `wc` (short for "word count") counts the number of lines, words and characters in a file. In this context, a line is any amount of

[2]`https://docs.python.org/3/library/stdtypes.html#set`

text followed by a return (i.e., a "newline"; basically the "enter" key). A word is a group of one or more non-whitespace characters delimited by (i.e., immediately preceded and followed by) whitespace characters (basically, space, tab and newlines). A character is, well, a character– a string of length 1 in Python. As an example, consider a file called `hw3example.txt` with contents

```
In the town where I was born
There lived a man who sailed the seas
and he told us of his life
in the land of submarines
```

You can download a copy of this file at this link:

`https://pages.stat.wisc.edu/~kdlevin/teaching/Spring2022/STAT679/hw/hw3example.txt`

Alternatively, you can download this file directly from your command line using the command `wget X`, where `X` is the URL listed above.

Running the command `wc hw3example.txt` on the command line yields the following:

```
keith@Steinhaus$ wc hw3example.txt
       4      27     120 hw3example.txt
```

That is to say, this file consists of 4 lines, 27 words and 120 characters. You sould be able to count that there are indeed four lines, 27 words (7 on the first line, 8 on the second, 7 on the third and 5 on the fourth) and 120 characters. Note that the newline at the end of each of the first three lines all count toward this character count, but that there is *not* a newline at the end of the last line.

This problem will walk you through implementing your own version of `wc` that you can run from the command line. This will involve writing a bunch of code in a script called `wc.py`. Please do not forget to **include this file** in your homework submission!

1. In a file called `wc.py`, define a function called `count_chars` that takes a string as its only arguments and returns the number of characters in that string. There is no need to perform any error checking in this function. **Hint:** this one is kind of a freebie– the number of characters is just the length of a string!

2. Continuing to work in the file called `wc.py`, define a function called `count_words` that takes a string as its only argument and returns the number of words in that string. As an example, `count_words('hello')`, `count_words(' hello ')` and `count_words(' hello!goodbye!')` should all return 1. `count_words(' hello goodbye ')` should return 2. `count_words(' ')` and `count_words(' ')` should both return zero. You may find it useful to use the Python string method `split`, about which you can read here: TODO.

3. Modify your file `wc.py` so that it can be called from the command line, taking a single command line argument. This command line argument should be interpreted as a filename, which your script should open, and print, one per line,

   - the number of lines in the file
   - the number of words in the file
   - the number of characters in the file

- the name of the file.

**Hint:** you may find it useful to use a text editor to create files of your own to test on. You can try running the UNIX/Linux command line program `wc` on those files to check that your script yields the same output (with slightly different formatting, of course). For example, running your script on `hw3example.txt` above should look like the following:

```
keith@Steinhaus$ python3 wc.py hw3example.txt
4
27
120
hw3example.txt
```

You do not need to perform any error checking in your script. That is, you may assume that the user will always call the script correctly, with a single command line argument that names a file that actually exists. Of course, in practice, it would be a very good idea to include error checking to make sure that nothing weird is happening, and you are welcome to include error checking if you wish to do so.