Chapter 7: Words Count: Reading files/STDIN, iterating lists, formatting strings

> *"I love to count!" — Count von Count*

Counting things is a surprisingly important programming skill. Maybe you're trying to find how many pizzas were sold each quarter or how many times you see certain words in a set of documents. Usually the data we deal with in computing comes to us in files, so we're going to push a little further into reading files and manipulating strings by writing a Python version of the venerable Unix `wc` (word count) program. We're going to write a program called `wc.py` that will count the characters, words, and lines for each input argument which should be files.

Given one or more valid files, it should print the number of lines, words, and characters, each in columns 8 characters wide, followed by a space and then the name of the file. Here's what it looks like for one file:

```
$ ./wc.py ../inputs/scarlet.txt
    7035   68061  396320 ../inputs/scarlet.txt
```

When there are many files, print the counts for each file and then print a "total" line summing each column:

```
$ ./wc.py ../inputs/s*.txt
    7035   68061  396320 ../inputs/scarlet.txt
      17     118     661 ../inputs/sonnet-29.txt
       3       7      45 ../inputs/spiders.txt
    7055   68186  397026 total
```

There may also be *no* arguments, in which case we'll read from "standard in" (`STDIN`) which is the complement to `STDOUT` we used in "Howler." You can use `STDIN` to chain programs together where the output of one program becomes the input for the next. To pass text via `STDIN` , you can use the `<` redirect from a file:

```
$ ./wc.py < ../inputs/fox.txt
       1       9      45 <stdin>
```

Or pipe (`|`) the output of one command into another:

```
$ cat ../inputs/fox.txt | ./wc.py
       1       9      45 <stdin>
```

Given a non-existent file, it should print an error message and exit with a non-zero exit value:

```
$ ./wc.py foo
usage: wc.py [-h] FILE [FILE ...]
wc.py: error: argument FILE: can't open 'foo': \
[Errno 2] No such file or directory: 'foo'
```

So our program is going to have to:

- Take 0 or more positional arguments.
- Validate that any given arguments are actually files.
- Read each file and count the lines, words, and characters.
- If no files are present, read from `STDIN`.
- Print the total number of lines, words, and characters if there is more than one file.

Defining file inputs

The first step will be to define your arguments to `argparse`. The program takes *zero or more* positional arguments and nothing else. Remember that you never have to define the `-h` or `--help` arguments as `argparse` handles those automatically.

In "Picnic," we used `nargs='+'` to indicate one or more items for our picnic. Here we want to use `nargs='\*'` to indicate *zero* or more. For what it's worth, there's one other value that `nargs` can take and that is `?` for *zero or one*. In all cases, the argument(s) will be returned as a `list`. Even if there are no arguments, you will still get an empty `list` (`[]`). For this program, if there are no arguments, we'll read `STDIN`.

Table 1. Possible values for `nargs`

Symbol	Meaning
?	zero or one
*	zero or more
+	one or more

In "Howler," we used the "standard out" (`STDOUT`) file handle with `sys.stdout`. To read `STDIN`, we'll use Python's `sys.stdin` file handle which is similarly always open and available to you. Because you are using `nargs='*'`, the values will be a list, and so the `default` should be a `list` as well. Can you figure out how to make the `default` value for your file argument be a `list` with `sys.stdin`?

Lastly, we should discuss the `type` of the positional arguments. If they are provided, they should be *readable files*. We saw in "Howler" how to test if the input argument was a file by using `os.path.isfile`. In that program, the input might be either plain text or a file name, so we had to check this ourselves. In this program, however, we will require that any arguments should be files, and so we can define the `type=argparse.FileType('r')`. When you do this, `argparse` takes on all the work to validate the inputs from the user and produce useful error messages. If the user provides valid input, then `argparse` will provide you with a `list` of *open file handles*. All in all, this saves you quite a bit of time.

Iterating lists

Your program will end up with a `list` of file handles. In "Jump The Five," we used a `for` loop to iterate through the characters in the input text. Here we can use a `for` loop over the `file` inputs.

```
for fh in args.file:
    # read each file
```

The `fh` is a "file handle." We saw in "Howler" how to manually `open` and `read` a file. Here the `fh` is already open, so we can just `read` it. There are many ways to read a file, however. The `read` method will give you the *entire contents* of the file in one go. If the file is large — say, if the size of the file exceeds your available memory on your machine — then your program will crash. I would recommend, instead, that you use another `for` loop on the `fh`. Python will understand this to mean that you wish to read each `line` of input, one-at-a-time.

```
for fh in args.file:
    for line in fh:
        # process the line
```

So that's two levels of `for` loops, one for each file handle and then another for each line in each file handle.

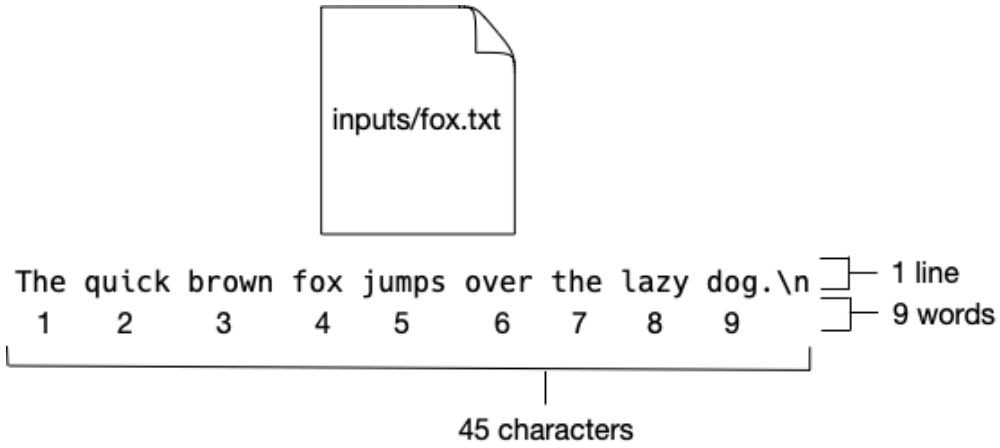
What you're counting

The output for each file will be the number of lines, words, and characters, each printed in a field 8 characters wide followed by the name of the file which will be available to you via `fh.name`. Let's take a look at the output from the standard `wc` program on my system. Notice that when run with just one argument, it produces counts only for that file:

```
$ wc fox.txt
       1       9      45 fox.txt
```

When run with multiple files, it also shows a "total" line:

```
$ wc fox.txt sonnet-29.txt
       1       9      45 fox.txt
      17     118     669 sonnet-
29.txt
      18     127     714 total
```



For each file, you will need to create variables that hold the numbers for lines, words, and characters. For instance, if you use the `for line in fh` loop that I suggest, then you need to have a variable like `num_lines` to increment on each iteration. That is, somewhere in your code you will need to set a variable to `0` and then, inside the `for` loop, make it go up by one. The idiom in Python for this is:

```
num_lines = 0
for line in fh:
    num_lines += 1
```

You also need to count the number of words and characters, so you'll need similar `num_words` and `num_chars` variables. In "Picnic," we discussed how we can convert back and forth between strings and lists. For the purposes of this exercise, we'll use the `str.split` method to break each `line` on spaces. You can then use the length of the resulting `list` as the number of words. For the number of characters, you can use the same length function on the `line` and add that to a `num_chars` variable.

Formatting your results

This is the first exercise where the output needs to be formatted in a particular way. Don't try handle this part manually. That way lies madness. Instead, you need to learn the magic of the `str.format` method. The `help` doesn't have much in the way of documentation, so I'd recommend you read PEP3101 (https://www.python.org/dev/peps/pep-3101/).

We've seen that the curlies (`{}`) inside the `str` part create placeholders that will be replaced by the values passed to the method:

```
>>> import math
>>> 'Pi is {}'.format(math.pi)
'Pi is 3.141592653589793'
```

You can put formatting information inside the curlies to specify how you want the value displayed. If you are familiar with `printf` from C-type languages, this is the same idea. For instance, I can print just two numbers of `pi` after the decimal. The `:` introduces the formatting options, and the `0.02f` describes two decimal points of precision:

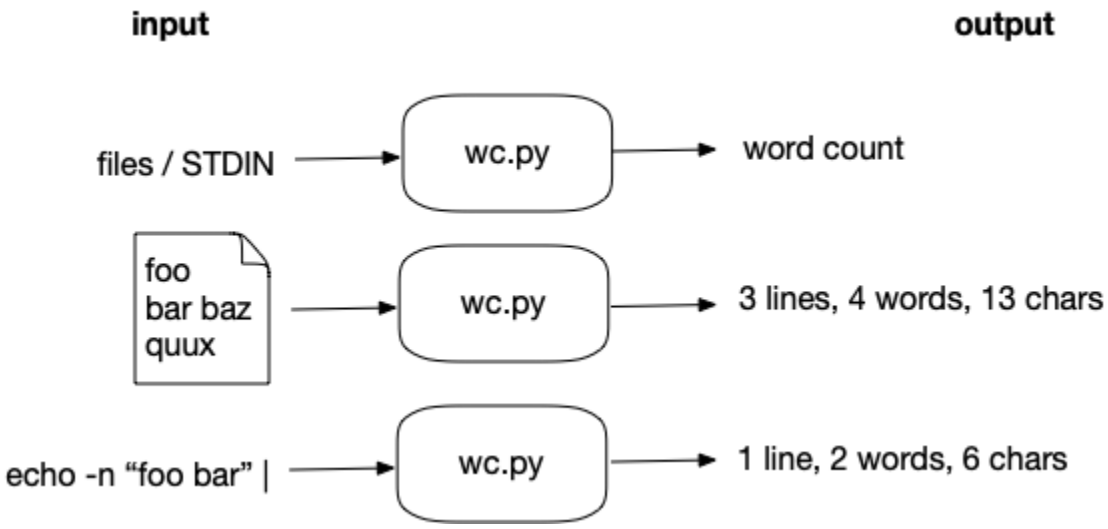
```
>>> 'Pi is {:0.02f}'.format(math.pi)
'Pi is 3.14'
```

The formatting information comes after the colon (`:`) inside the curlies. You can also use the f-string method where the variable comes *before* the colon:

```
>>> f'Pi is {math.pi:0.02f}'
'Pi is 3.14'
```

Here you need to use `{:8}` for each of lines, words, and characters so that they all line up in neat columns. The `8` describes the width of the field which is assumed to be a string. The text will be right-justified.

Writing `wc`



Now it's time to write out Python program called `wc.py` that will emulate the `wc` program in Unix that counts the lines, words, and characters in the given file arguments. If run with the `-h` or `--help` flags, the program should print usage:

```
$ ./wc.py -h
usage: wc.py [-h] [FILE [FILE ...]]

Argparse Python script

positional arguments:
  FILE        Input file(s) (default: [<_io.TextIOWrapper name='<stdin>'
              mode='r' encoding='UTF-8'>])

optional arguments:
  -h, --help  show this help message and exit
```

Hints:

- Start with `new.py` and delete all the non-positional arguments.
- Use `nargs='*'` to indicate zero or more positional arguments for your `file` argument.
- How could you use `sys.stdin` for the `default`? Remember that both `narg='*'` and `nargs='+'` mean that the arguments will be supplied as a `list`. How can you create a `list` that contains just `sys.stdin` for the `default` value?
- Remember that you are just trying to pass one test at a time. Create the program, get the help right, then worry about the first test.
- Compare the results of your version to the `wc` installed on your system. Note that not every Unix-like system has the same `wc`, so results may vary.

Time to write this yourself before you read the solution. Fear is the mind-killer. You can do this.

Solution

```
#!/usr/bin/env python3
"""Emulate wc (word count)"""

import argparse
import sys


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Emulate wc (word count)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        nargs='*',
                        default=[sys.stdin],          1
                        type=argparse.FileType('r'),  2
                        help='Input file(s)')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()

    total_lines, total_chars, total_words = 0, 0, 0  3
    for fh in args.file:                             4
        lines, words, chars = 0, 0, 0                5
        for line in fh:                              6
            lines += 1                               7
            chars += len(line)                       8
            words += len(line.split())               9

        total_lines += lines                         10
        total_chars += chars
        total_words += words

        print(f'{lines:8}{words:8}{chars:8} {fh.name}')  11

    if len(args.file) > 1:                           12
        print(f'{total_lines:8}{total_words:8}{total_chars:8} total')  13


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

1 If you set the `defaut` to a `list` with `sys.stdin`, then you have handled the `STDIN` option.

2 If the user supplies any arguments, `argparse` will check if they are valid file inputs. If there is a problem, `argparse` will halt execution of the program and show the user an error message.

3 These are the variables for the "total" line, if we need them.

4 Iterate through the `list` of `arg.file` inputs. I use the variable `fh` to remind me that these are open file hanndles, even `STDIN`.

5 Initialize variables to count *just this file*.

6 Iterate through each `line` of `fh`.

7 For each line, we increment `lines` by 1.

8 The number of `chars` is incremented by the length of the `line`.

9 To get the number of words, we can `split` the `line` on spaces (the default). We length of that `list` is added to the `words`.

10 We add the numbers for this file to the `total_` variables.

11 Print the counts for this file using the `{:8}` option to print in a field 8 characters wide.

12 Check if we had more than 1 input.

13 Print the "total" line.

Discussion

Defining the arguments

This program is rather short and seems rather simple, but it's definitely not exactly easy. One part of the exercise is to really get familiar with `argparse` and the trouble it can save you. The key is in defining the `file` positional arguments. If you use `nargs='*'` to indicate zero or more arguments, then you know `argparse` is going to give you back a `list` with zero or more elements. If you use `type=argparse.FileType('r')`, then any arguments provided must be readable files. The `list` that `argparse` returns will be a `list` of *open file handles*. Lastly, if you use `default=[sys.stdin]`, then you understand that `sys.stdin` is essentially an open file handle to read from "standard in" (AKA `STDIN`), and you are letting `argparse` know that you want the default to be a `list` containing `sys.stdin`.

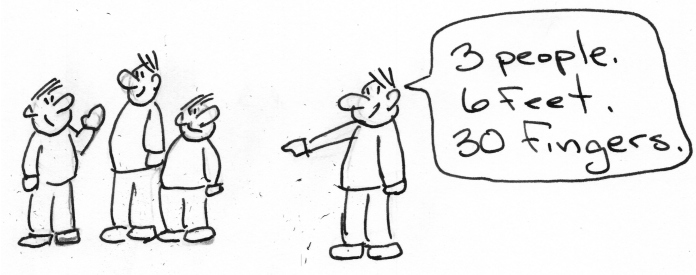
Reading a file using a `for` loop

I can create a list of open file handles in the REPL to mimic what I'd get from `args.file`:

```
>>> files = [open('../inputs/fox.txt')]
```

Before I use a `for` loop to iterate through them, I need to set up three variables to track the *total* number of lines, words, and characters:

```
>>> total_lines, total_chars, total_words = 0, 0, 0
```



Inside the `for` loop for each file handle, I initialize three more variables to hold the count of lines, characters, and words *for this particular file*. I then use another `for` loop to iterate over each line in the file handle (`fh`). For the `lines`, I can add `1` on each pass through the `for` loop. For the `chars`, I can add length of the line (`len(line)`) to track the number of characters. Lastly for the `words`, I can use `line.split()` to break the line on whitespace to create a `list` of "words." It's not actually a perfect way to count actual words, but it's close enough. I can use the `len` function on the `list` to add to the `words` variable. The `for` loop ends when the end of the file is reached, and that is when I can `print` out the counts and the file name using `{:8}` placeholders in the print template to indicate a text field 8 characters wide.

```
>>> for fh in files:
...     lines, words, chars = 0, 0, 0
...     for line in fh:
...         lines += 1
...         chars += len(line)
...         words += len(line.split())
...     print(f'{lines:8}{words:8}{chars:8} {fh.name}')
...     total_lines += lines
...     total_chars += chars
...     total_words += words
...
       1       9      45 ../inputs/fox.txt
```

Notice that the `print` statement lines up with the inner `for` loop so that it will run after we're done iterating over the lines in `fh`. I chose to use the f-string method to print each of `lines`, `words`, and `chars` in a space 8 characters wide. After printing, I can add the counts to my "total" variables to keep a running total.

Lastly, if the number of file arguments is greater than 1, I need to print my totals:

```
if len(args.file) > 1:
    print(f'{total_lines:8}{total_words:8}{total_chars:8} total')
```

A low-memory version

There is a potentially serious problem waiting to bite us in this program. In the `get_args`, we're reading the entire file into memory with this line:

```
if os.path.isfile(args.text):
    args.text = open(args.text).read().rstrip()
```

We could, instead, only `open` the file:

```
if os.path.isfile(args.text):
    args.text = open(args.text)
```

And later read it line-by-line:

```
for line in args.text:
    out_fh.write(line.upper())
```

The problem, though, is how to handle the times when the `text` argument is actually text and not the name of a file. The `io` (input-output) module in Python has a way to represent text as *stream*:

```
>>> import io                               1
>>> text = io.StringIO('foo\nbar\nbaz\n')   2
>>> for line in text:                       3
...     print(line, end='')                 4
...
foo
bar
baz
```

1. Import the `io` module.
2. Use the `io.StringIO` function to turn the given `str` value into something we can treat like an open file handle.
3. Use a `for` loop to iterate the "lines" of text by separated by newlines.
4. Print the `line` using the `end=''` option to avoid having 2 newlines.

To make this work, we can change how we handle the `args.text` like so:

```
if os.path.isfile(args.text):    1
    args.text = open(args.text)  2
else:
    args.text = io.StringIO(args.text + '\n')  3
```

1. If `args.text` is a file...
2. Replace `args.text` with the file handle created by opening the file named by it.
3. Otherwise, replace `args.text` with an `io.StringIO` value that will act like an open file handle. Note that we need to add a newline to the text so that it will look like the lines of input coming from an actual file.

Review

- The `nargs` (number of arguments) option to `argparse` allows you to validate the number of arguments from the user. The star (`'*'`) means zero or more while `'+'` means one or more.
- If you define an argument using `type=argparse.FileType('r')`, then `argparse` will validate that the user has provided a readable file and will make the value available in your code as an open file handle.
- You can read and write from the Unix standard in/out file handles by using `sys.stdin` and `sys.stdout`.
- You can nest `for` loops to handle multiple levels of processing.
- The `str.split` method will split a string on spaces into words.
- The `len` function can be used on both strings and lists. For the latter, it will tell you the number of elements contained.
- The `str.format` and Python's f-strings both recognize the same printf-style formatting options to allow you to control how a value is displayed.

Going Further

- Add the same flags as your system `wc` and change your program to create the same output; i.e., print all the columns for no arguments but only print the "lines" when `-l` is present.
- Implement other system tools like `head`, `tail`, `cat`, and `tac` (the reverse of `cat`).

Last updated 2020-01-14 20:37:40 -0700