1. 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 lines, words, and bytes found in each input. The counts will appear in columns 8 characters wide and will be followed by the name of the file. The inputs for the program which may be given as one or more positional arguments. For instance, here is what it should print for one file:

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

When counting multiple files, there will be an additional "total" line summing each column:

```
$ ./wc.py ../inputs/const.txt ../inputs/sonnet-29.txt
     865    7620   44841 ../inputs/const.txt
      17     118     661 ../inputs/sonnet-29.txt
     882    7738   45502 total
```

There may also be *no* arguments, in which case we'll read from "standard in" which is often written as `STDIN`. We started talking about `STDOUT` in "Howler" when we used `sys.stdout` as a file handle. `STDIN` is the complement to `STDOUT` — it's the "standard" place to read input on the command line. When our program is given *no* positional arguments, we'll read from `sys.stdin`.

For instance, the `cat` program will print the contents of a file to `STDOUT`. We can use the pipe operator (`|`) to funnel that output into our program:

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

Another option is to use the `<` operator to redirect input from a file:

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

Tools that print to `STDOUT` and read from `STDIN` can be chained together to create novel, *ad hoc* programs! One of the handiest command-line tools is `grep` which can find patterns of text in files. If, for instance, we wanted to find all the lines of text that contain the word "scarlet" in all the the files in the `inputs` directory using this command:

```
$ grep scarlet ../inputs/*.txt
```

On the command line, the `*` is a wildcard that will match anything so `*.txt` will match any file ending with `.txt`. If you run that command, you'll see quite a bit of output. To count the lines found by `grep`, we can "pipe" that output into our `wc.py` program like so:

```
$ grep scarlet ../inputs/*.txt | ./wc.py
     104    1188    9182 <stdin>
```

In this exercise, you will:

- Learn how to process zero or more positional arguments
- Validate input files
- Read from files or from "standard in"
- Use multiple levels of `for` loops
- Break files into lines, words, and bytes
- Use counter variables
- Format string output

1.1. Writing `wc.py`

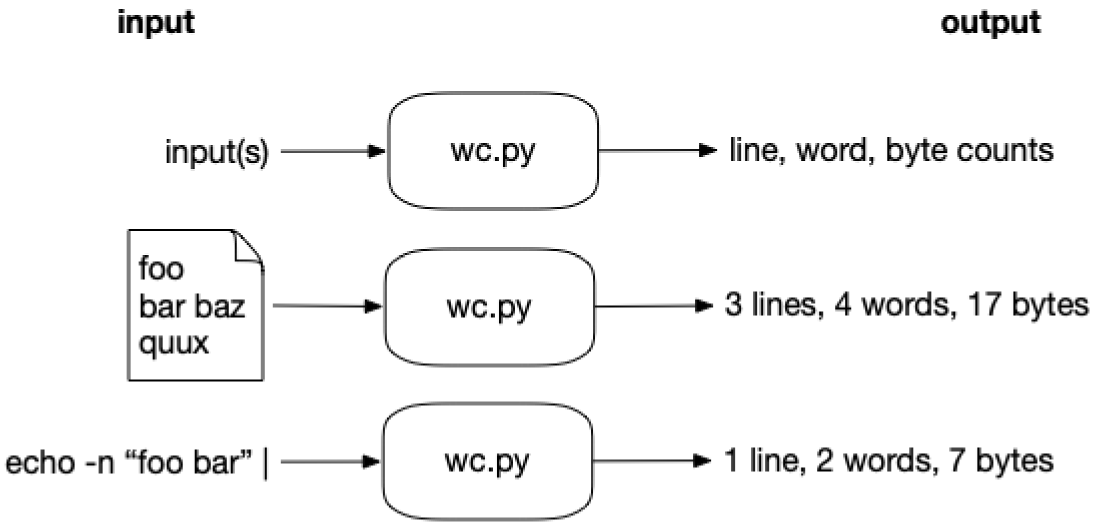
Let's get started! Create your program and modify the arguments until it will print the following usage if run with the `-h` or `--help` flags:

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

Emulate wc (word count)

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
```

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

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

Here is a string diagram to help you think about how the program should work:



1.1.1. Defining file inputs

The first step will be to define your arguments to `argparse`. This 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 6. 1. Possible values for `nargs`

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

Any arguments that are provided to our program *must be readable files*. In "Howler" we learned how to test if the input argument is a file by using `os.path.isfile`. The input was allowed be either plain text or a file name, so we had to check this ourselves.

In this program, the input arguments are required to be files, so we can define our arguments using `type=argparse.FileType('r')`. This means that `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. (Be sure to review the "File arguments" section in the `argparse` appendix.)

In "Howler," we used `sys.stdout` to write to `STDOUT`. To read from `STDIN`, we'll use Python's `sys.stdin` file handle. Like `sys.stdout`, the `sys.stdin` file handle does not need an `open` — it's always present and available for printing.

Because we are using `nargs` to define our argument, the result will always be a `list`. In order to set `sys.stdin` as the `default` value, we should place it in a `list` like so:

```
parser.add_argument('file',
                    metavar='FILE',
                    nargs='*',                   ①
                    type=argparse.FileType('r'), ②
                    default=[sys.stdin],         ③
                    help='Input file(s)')
```

① Zero or more of this argument.

② If arguments are provided, they must be readable files. The files will be opened by `argparse` and will be provided as file handles.

③ The default will be a `list` containing `sys.stdin` which is like an open file handle to `STDIN`. We do not need to `open` it.

1.1.2. 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 read the contents from it. There are many ways to read a file, however. The `fh.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 a `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: # ONE LOOP!
    for line in fh:  # TWO LOOPS!
        # process the line
```

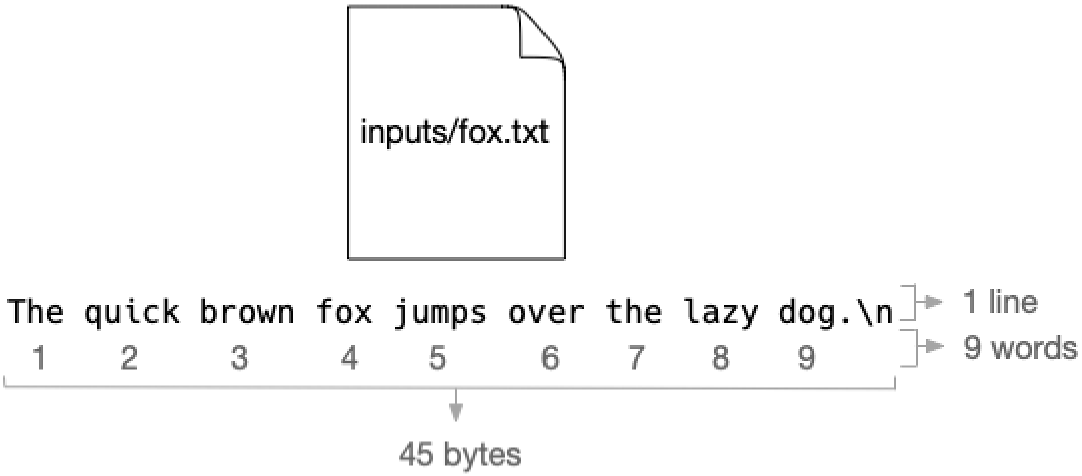
So that's two levels of `for` loops, one for each file handle and then another for each line in each file handle. TWO LOOPS! I LOVE TO COUNT!

1.1.3. What you're counting

The output for each file will be the number of lines, words, and bytes (like characters and whitespace), each printed in a field 8 characters wide followed by a space and then 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
```

The `fox.txt` file is short enough that you could manually verify that it does in fact contain one line, nine words, and 45 bytes which includes all the characters, spaces, and the trailing newline:

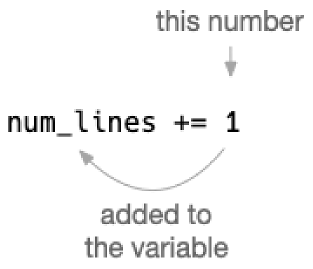


When run with multiple files, the standard `wc` program also shows a "total" line:

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

We are going to emulate the behavior of this program. For each file, you will need to create variables to hold the numbers for lines, words, and bytes. 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 is to use the `+=` operator to add some value on the right-hand side to the variable on the left-hand-side like so:



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

You will also need to count the number of words and bytes, so you'll need similar `num_words` and `num_bytes` variables. To get the "words," we'll use the `str.split()` method to break each `line` on spaces. [1] You can then use the length of the resulting `list` as the number of words. For the number of bytes, you can use the `len` (length) function on the `line` and add that to a `num_bytes` variable.

1.1.4. 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. Place a single space between the last column and the name of the file which you can find in `fh.name`.

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.

1.2. 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_bytes, total_words = 0, 0, 0  ③
    for fh in args.file:                             ④
        num_lines, num_words, num_bytes = 0, 0, 0    ⑤
        for line in fh:                              ⑥
            num_lines += 1                           ⑦
            num_bytes += len(line)                   ⑧
            num_words += len(line.split())           ⑨

        total_lines += num_lines                     ⑩
        total_bytes += num_bytes
        total_words += num_words

        print(f'{num_lines:8}{num_words:8}{num_bytes:8} {fh.name}') ⑪

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

```

```

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

① If you set the `default` 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 handles, 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 `bytes` 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.

1.3. Discussion

1.3.1. 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`.

1.3.2. 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. I could define them on three separate lines:

```
>>> total_lines = 0
>>> total_words = 0
>>> total_bytes = 0
```

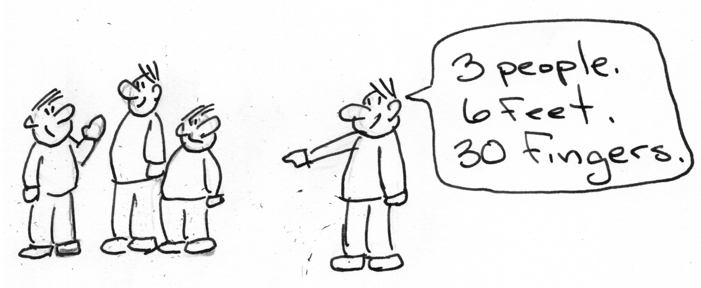
Or I can declare them on a single line. Technically I'm creating a `tuple` on the right-hand side by placing commas in between the three zeros, and I'm "unpacking" those three values into three variables on the left-hand side. We'll have more to say about tuples much later:

```
>>> total_lines, total_words, total_bytes = 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 `bytes`, I can add length of the line (`len(line)`) to track the number of "characters" (which may be printable characters or whitespace so it's easiest to call them "bytes"). 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, bytes = 0, 0, 0
...     for line in fh:
...         lines += 1
...         bytes += len(line)
...         words += len(line.split())
...     print(f'{lines:8}{words:8}{bytes:8} {fh.name}')
...     total_lines += lines
...     total_bytes += bytes
...     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 `bytes` 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_bytes:8} total')
```

1.4. 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.

1.5. Going Further

- By default, wc will print all the columns like our program, but it will also accept flags to print -c for number of characters, -l for number of lines, and -w for number of words. When any of these flags are present, only those columns for the given flags are shown, so wc.py -wc would show just the columns for words and characters. Add both short and long flags for these options to your program so that it behaves exactly like wc.
- Write your own implementation of other system tools like cat (to print the contents of a file to STDOUT), head (to print just the first n lines of a file), tail (to print the last n lines of a file), and tac (to print the lines of a file in reverse order).

[1] Splitting the text on spaces doesn't actually produce "words" because it won't separate the punctuation like commas and periods from the letters, but it's close enough for this program.