6 Words count: Reading files and 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 out 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 in this chapter, we're going to push a little further into reading files and manipulating strings.

We're going to write a Python version of the venerable `wc` ("word count") program. Ours will be called wc.py, and it will count the lines, words, and bytes found in each input supplied as one or more positional arguments. The counts will appear in columns eight characters wide, and they will be followed by the name of the file. For instance, here is what wc.py 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 chapter 5 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, it will read from `sys.stdin`.

`STDIN` and `STDOUT` are common file handles that many command-line programs recognize. We can chain the `STDOUT` from one program to the `STDIN` of another to create ad hoc programs. For instance, the `cat` program will print the contents of a file to `STDOUT`. We can use the pipe operator (`|`) to funnel that output as input into our program via `STDIN`:

```
$ 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>
```

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 files in the inputs directory, we could use this command:

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

On the command line, the asterisk (`*`) is a wildcard that will match anything, so `*.txt` will match any file ending with ".txt." If you run the preceding 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
     108    1192    9201 <stdin>
```

We can verify that this matches what `wc` finds:

```
$ grep scarlet ../inputs/*.txt | wc
     108    1192    9201
```

In this chapter, you will

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

6.1 Writing wc.py

Let's get started! Create a program called wc.py in the 06_wc directory, 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 nonexistent file, your program should print an error message and exit with a nonzero 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'
```

Figure 6.1 is a string diagram that will help you think about how the program should work.

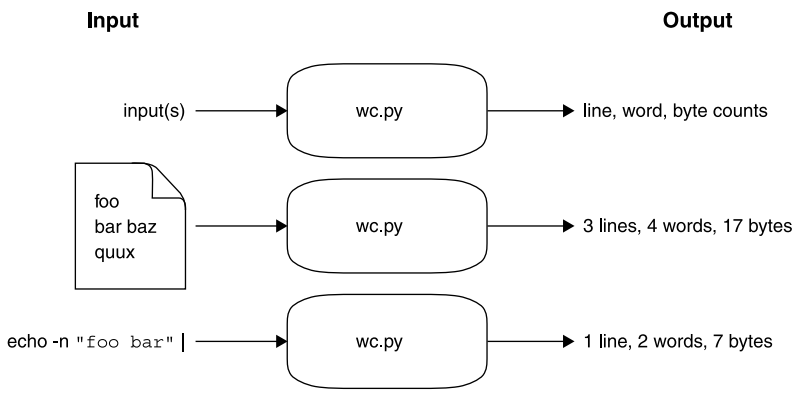


Figure 6.1 A string diagram showing that wc.py will read one or more file inputs or possibly `STDIN` and will produce a summary of the words, lines, and bytes contained in each input.

6.1.1 Defining file inputs

Let's talk about how we can define the program's parameters using `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 chapter 3 we used `nargs='+'` to indicate one or more items for our picnic. Here we want to use `nargs='*'` to indicate *zero* or more. When there are no arguments, the default value will be `None`. For this program, we'll read `STDIN` when there are no arguments.

All of the possible values for `nargs` are listed in table 6.1.

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 chapter 5 you learned how to test whether the input argument was a file by using `os.path.isfile()`. The input was allowed to be either plain text or a filename, so you had to check this yourself.

In this program, the input arguments are required to be readable text files, so we can define our arguments using `type=argparse.FileType('rt')`. This means that `argparse` takes on all the work of validating the inputs from the user and producing useful error messages. If the user provides valid input, `argparse` will provide a `list` of *open file handles*. All in all, this will save us quite a bit of time. (Be sure to review section A.4.6 on file arguments in the appendix.)

In chapter 5 we used `sys.stdout` to write to `STDOUT`. To read from `STDIN` here, 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 reading.

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

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

Zero or more of this argument

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

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.

6.1.2 Iterating lists

Your program will end up with a `list` of file handles that will need to be processed. In chapter 4 we used a `for` loop to iterate through the characters in the input text. Here we can use a `for` loop over the `args.file` inputs, which will be open file handles:

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

You can give whatever name you like to the variable you use in your `for` loop, but I think it's very important to give it a semantically meaningful name. Here the variable name `fh` reminds me that this is an open file handle. You saw in chapter 5 how to manually `open()` and `read()` a file. Here `fh` is already open, so we can use it directly to read the contents.

There are many ways to read a file. The `fh.read()` method will give you the *entire contents* of the file in one go. If the file is large—if it exceeds the available memory on your machine—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 the file handle, one at a time.

```
for fh in args.file: # ONE LOOP!
    for line in fh:  # TWO LOOPS!
        # process the line
```

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

6.1.3 What you're counting

The output for each file will be the number of lines, words, and bytes (like characters and whitespace), each of which is printed in a field eight 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 it's 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 1 line, 9 words, and 45 bytes, which includes all the characters, spaces, and the trailing newline (see figure 6.2).

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
```

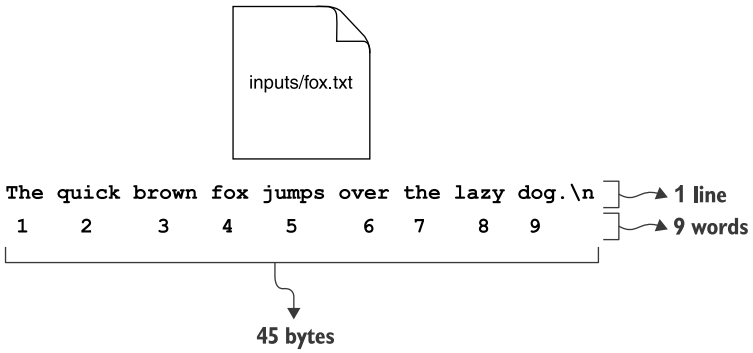


Figure 6.2 The fox.txt file contains 1 line of text, 9 words, and a total of 45 bytes.

We are going to emulate the behavior of this program. For each file, you will need to create variables to hold the numbers of lines, words, and bytes. For instance, if you use the `for line in fh` loop that I suggest, you will 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 1. The idiom in Python is to use the `+=` operator to add some value on the right side to the variable on the left side (as shown in figure 6.3):

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

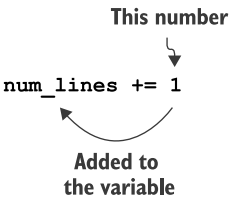


Figure 6.3 The += operator will add the value on the right to the variable on the left.

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. 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.

> **NOTE** 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. In chapter 15, we'll look at how to use a regular expression to separate strings that look like words from others that do not.

6.1.4 Formatting your results

This is the first exercise where the output needs to be formatted in a particular way. Don't try to 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 recommend you read PEP 3101 on advanced string formatting (www.python.org/dev/peps/pep-3101/).

The `str.format()` method uses a template that contains curly brackets (`{}`) to create placeholders for the values passed as arguments. For example, we can print the raw value of `math.pi` like so:

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

You can add formatting instructions after a colon (`:`) 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 `math.pi` with two numbers after the decimal by specifying `0.02f`:

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

In the preceding example, the colon (`:`) introduces the formatting options, and the `0.02f` describes two decimal points of precision.

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'
```

In this chapter's exercise, you need to use the formatting option `{:8}` to align each of the lines, words, and characters into columns. The `8` describes the width of the field. The text is usually left-justified, like so:

```
>>> '{:8}'.format('hello')
'hello   '
```

But the text will be right-justified when you are formatting numeric values:

```
>>> '{:8}'.format(123)
'     123'
```

You will need to place a single space between the last column and the name of the file, which you can find in `fh.name`.

Here are a few hints:

- Start with new.py and delete all the nonpositional arguments.
- Use `nargs='*'` to indicate zero or more positional arguments for your `file` argument.
- Try to pass one test at a time. Create the program, get the help right, and then worry about the first test, then the next, and so on.
- Compare the results of your version to the `wc` installed on your system. Note that not every system has the same version of `wc`, so results may vary.

It's time to write this yourself before you read the solution. Fear is the mind killer. You can do this.

6.2 Solution

Here is one way to satisfy the tests. Remember, it's fine if you wrote it differently, as long as it's correct and you understand your code!

```
#!/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],
                        type=argparse.FileType('rt'),
                        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
    for fh in args.file:
        num_lines, num_words, num_bytes = 0, 0, 0
        for line in fh:
            num_lines += 1
            num_bytes += len(line)
            num_words += len(line.split())

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

If you set the default to a list with sys.stdin, you have handled the STDIN option.

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.

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

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

Initialize variables to count the lines, words, and bytes in just this file.

Iterate through each line of the file handle.

For each line, increment the number of lines by 1.

To get the number of words, we can call line.split() to break the line on whitespace. The length of that list is added to the count of words.

The number of bytes is incremented by the length of the line.

Add all the counts for lines, words, and bytes for this file to the variables for counting the totals.

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

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

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

Print the "total" line.

Check if we had more than 1 input.

Print the counts for this file using the {:8} option to print in a field 8 characters wide followed by a single space and then the name of the file.

6.3 Discussion

This program is rather short and seems rather simple, but it's not exactly easy. Let's break down the main ideas in the program.

6.3.1 Defining the arguments

One point of this exercise is to get familiar with `argparse` and the trouble it can save you. The key is in defining the `file` parameter. We use `type=argparse.FileType('rt')` to indicate that any arguments provided must be readable text files. We use `nargs='*'` to indicate zero or more arguments, and we set the `default` to be a list containing `sys.stdin`. This means we know that `argparse` will always give us a `list` of one or more open file handles.

That's really quite a bit of logic packed into a small space, and most of the work validating the inputs, generating error messages, and handling the defaults is all done for us!

6.3.2 Reading a file using a for loop

The values that `argparse` returns for `args.file` will be a `list` of *open file handles*. We can create such a list in the REPL to mimic what we'd get from `args.file`:

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

Before we use a `for` loop to iterate through them, we need to set up three variables to track the *total* number of lines, words, and characters. We could define them on three separate lines:

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

Or we can declare them on a single line like the following:

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

Technically we're creating a `tuple` on the right side by placing commas between the three zeros and then "unpacking" them into three variables on the left side. I'll have more to say about tuples much later.

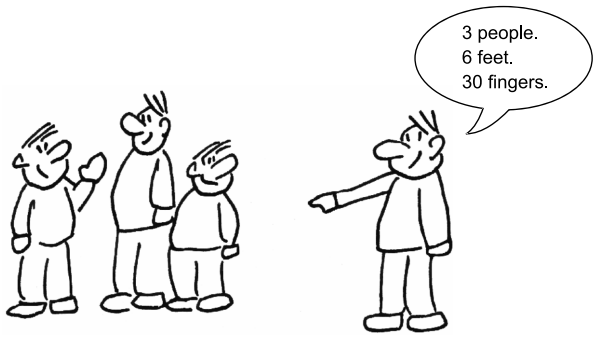
Inside the `for` loop for each file handle, we initialize three more variables to hold the count of lines, characters, and words *for this particular file*. We can then use another `for` loop to iterate over each line in the file handle (`fh`). For `lines`, we can add `1` on each pass through the `for` loop. For `bytes`, we can add the 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 `words`, we can use `line.split()` to break the line on whitespace to create a `list` of "words." It's not a perfect way to count actual words, but it's close enough. We 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. Next we can `print()` out the counts and the filename, 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 preceding call to `print()` lines up with the *second* `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 eight characters wide, followed by one space and then the `fh.name` of the file.

After printing, we can add the counts to the "total" variables to keep a running total.



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

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

6.4 Going further

- By default, `wc` will print all the columns like our program does, 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 columns for the specified flags are shown, so `wc.py -wc` would show just the columns for words and characters. Add 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).

Summary

- The `nargs` (number of arguments) option to `argparse` allows you to validate the number of arguments from the user. The asterisk (`'*'`) means zero or more, whereas `'+'` means one or more.
- If you define an argument using `type=argparse.FileType('rt')`, `argparse` will validate that the user has provided a readable text file and will make the value available in your code as an open file handle.
- You can read and write from the 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.
- The `len()` function can be used on both strings and lists. For lists, it will tell you the number of elements the list contains.
- Both `str.format()` and Python's f-strings recognize `printf`-style formatting options to allow you to control how a value is displayed.