title: The Unix Shell
subtitle: Pipes and Filters
minutes: 20

The Unix Shell: Pipes and Filters

Learning Objectives

- Redirect a command's output to a file.
- Process a file instead of keyboard input using redirection.
- Construct command pipelines with two or more stages.
- Explain what usually happens if a program or pipeline isn't given any input to process.
- Explain Unix's "small pieces, loosely joined" philosophy.

Now that we know a few basic commands, we can finally look at the shell's most powerful feature: the ease with which it lets us combine existing programs in new ways.

Counting

We'll start with a directory called `data/articles` that contains text files of news articles. They are organized by the region they cover. So africa1.txt is the first article about Africa, africa2.txt the second and so on.

```
$ ls ~/programming-fundamentals/data/articles
```

Wow, that's a lot of files!

Let's go into that directory with `cd` and run the command `wc africa*.txt`.
`wc` is the "word count" command: it counts the number of lines, words, and characters in files.
Remember that the `*` in `africa*.txt` matches zero or more characters, so the shell turns `africa*.txt` into a complete list of `.txt` files that start with `africa`:

```
$ cd ~/programming-fundamentals/data/articles
$ wc africa*.txt
```

```
     ...
      1     101     608 africa95.txt
      1    1576    9632 africa96.txt
      1     515    3218 africa97.txt
      1     653    4177 africa98.txt
      1     597    3748 africa99.txt
    136   70982  442767 total
```

If we run `wc -l` instead of just `wc`, the output shows only the number of lines per file:

```
$ wc -l africa*.txt
```

```
    ...
     1 africa94.txt
     1 africa95.txt
     1 africa96.txt
     1 africa97.txt
     1 africa98.txt
     1 africa99.txt
   136 total
```

So we see that each article has only 1 lines. How can this be, if the number of words vary so widely? Open a file and try to guess why.

We can also use `-w` to get only the number of words, or `-c` to get only the number of characters.

Which of these files is shortest? It's an easy question to answer when there are only a few files, but what if there were 6000?

Redirecting and Printing

Our first step toward a solution is to run the command:

```
$ wc -w africa*.txt > lengths
```

The > tells the shell to **redirect** the command's output to a file instead of printing it to the screen. The shell will create the file if it doesn't exist, or overwrite the contents of that file if it does.

(This is why there is no screen output: everything that wc would have printed has gone into the file `lengths` instead.) `ls lengths` confirms that the file exists:

```
$ ls lengths
```

```
lengths
```

We can now send the content of `lengths` to the screen using `cat lengths`.
`cat` stands for "concatenate": it prints the contents of files one after another. There's only one file in this case, so `cat` just shows us what it contains:

```
$ cat lengths
```

```
   ...
  101 africa95.txt
 1576 africa96.txt
  515 africa97.txt
  653 africa98.txt
  597 africa99.txt
  591 africa9.txt
```

Sorting

Now let's use the `sort` command to sort its contents. We will also use the `-n` flag to specify that the sort is numerical instead of alphabetical. This does *not* change the file; instead, it sends the sorted result to the screen:

```
$ sort -n lengths
```

```
  ...
 1143 africa134.txt
 1155 africa43.txt
 1227 africa25.txt
 1329 africa24.txt
 1576 africa96.txt
```

We can reverse sort with an addtional argument `sort -n -r lengths`. We can put the sorted list of lines in another temporary file called `sorted-lengths` by putting `sorted-lengths` after the command, just as we used `lengths` to put the output of `wc` into `lengths`. Once we've done that, we can run another command called `head` to get the first few lines in `sorted-lengths`:

```
$ sort -n lengths > sorted-lengths
$ head -1 sorted-lengths
```

```
   70 africa49.txt
```

Using the parameter `-1` with `head` tells it that we only want the first line of the file; `-20` would get the first 20, and so on. Since `sorted-lengths` contains the lengths of our files ordered from least to greatest, the output of `head` must be the file with the fewest lines.

What do you think `tail` does?

```
$ tail -1 sorted-lengths
```

```
   70982 total
```

Pipes

If you think this is confusing, you're in good company: even once you understand what `wc`, `sort`, and `head` do, all those intermediate files make it hard to follow what's going on. We can make it easier to understand by running `sort` and `head` together:

```
$ sort -n lengths | head -1
```

```
   70 africa49.txt
```

The vertical bar between the two commands is called a **pipe**. It tells the shell that we want to use the output of the command on the left as the input to the command on the right. The computer might create a temporary file if it needs to, or copy data from one program to the other in memory, or something else entirely; we don't have to know or care.

We can use another pipe to send the output of `wc` directly to `sort`, which then sends its output to `head`:

```
$ wc -w africa*.txt | sort -n | head -1
```

```
   70 africa49.txt
```

This is exactly like a mathematician nesting functions like *log(3x)*
and saying "the log of three times *x*". In our case, the calculation is "head of sort of word count of `africa*.txt`".

We can use this logic in many different combinations. For instance, to see how many files are in this directory, we can command:

```
$ ls -1 | wc -l
```

```
  958
```

This uses wc to do a count of the number of lines (`-l`) in the output of `ls -1`.

Processes, Inputs, Outputs, and Filters

Here's what actually happens behind the scenes when we create a pipe. When a computer runs a program --- any program --- it creates a **process** in memory to hold the program's software and its current state. Every process has an input channel called **standard input**. (By this point, you may be surprised that the name is so memorable, but don't worry: most Unix programmers call it "stdin". Every process also has a default output channel called **standard output** (or "stdout").

The shell is actually just another program. Under normal circumstances, whatever we type on the keyboard is sent to the shell on its standard input, and whatever it produces on standard output is displayed on our screen. When we tell the shell to run a program, it creates a new process and temporarily sends whatever we type on our keyboard to that process's standard input, and whatever the process sends to standard output to the screen.

Here's what happens when we run `wc -w africa*.txt > lengths`. The shell starts by telling the computer to create a new process to run the wc program. Since we've provided some filenames as parameters, wc reads from them instead of from standard input. And since we've used > to redirect output to a file, the shell connects the process's standard output to that file.

If we run `wc -w africa*.txt | sort -n` instead, the shell creates two processes (one for each process in the pipe) so that wc and `sort` run simultaneously. The standard output of wc is fed directly to the standard input of `sort`; since there's no redirection with >, `sort`'s output goes to the screen. And if we run `wc -w africa*.txt | sort -n | head -1`, we get three processes with data flowing from the files, through wc to `sort`, and from `sort` through `head` to the screen.

This simple idea is why Unix has been so successful. Instead of creating enormous programs that try to do many different things, Unix programmers focus on creating lots of simple tools that each do one job well, and that work well with each other. This programming model is called "pipes and filters". We've already seen pipes; a **filter** is a program like wc or `sort` that transforms a stream of input into a stream of output. Almost all of the standard Unix tools can work this way: unless told to do otherwise, they read from standard input, do something with what they've read, and write to standard output.

The key is that any program that reads lines of text from standard input
and writes lines of text to standard output can be combined with every other program that behaves this way as well. You can *and should* write your programs this way so that you and other people can put those programs into pipes to multiply their power.

Redirecting Input

As well as using > to redirect a program's output, we can use < to redirect its input, i.e., to read from a file instead of from standard input. For example, instead of writing `wc africa1.txt`, we could write `wc < africa1.txt`. In the first case, wc gets a command line parameter telling it what file to open. In the second, wc doesn't have any command line parameters, so it reads from standard input, but we have told the shell to send the contents of `africa1.txt` to wc's standard input.

Rochelle's Pipeline: Concatenating Files.

Rochelle has her bulk text downloads in the `new-york-times/2015-01-01` directory described earlier.

As a quick sanity check, she types:

```
$ cd ~/programming-fundamentals/data/new-york-times/2015-01-01
$ wc -l *.TXT
```

```
  63661 human-rights-2000.TXT
  56035 human-rights-2001.TXT
  60045 human-rights-2002.TXT
  46873 human-rights-2003.TXT
  62611 human-rights-2004.TXT
  55557 human-rights-2005.TXT
  62905 human-rights-2006.TXT
  44866 human-rights-2007.TXT
  54580 human-rights-2008.TXT
  66540 human-rights-2009.TXT
 573673 total
```

Now she wants to concatenate (or merge) all of these text files into one big text file that she can later use to parse into a CSV.

```
$ cat *.TXT > all.TXT
$ wc -l *.TXT
```

```
 573673 all.TXT
  63661 human-rights-2000.TXT
  56035 human-rights-2001.TXT
  60045 human-rights-2002.TXT
  46873 human-rights-2003.TXT
  62611 human-rights-2004.TXT
  55557 human-rights-2005.TXT
  62905 human-rights-2006.TXT
  44866 human-rights-2007.TXT
  54580 human-rights-2008.TXT
  66540 human-rights-2009.TXT
1147346 total
```

Notice that `all.TXT` is the sum of all the lines of the other `*.TXT` files.

Exercises

Challenge 1

Create a file under my_files called "sorts.txt" that contains these data

```
10
2
19
22
6
```

Sort it numerically and place the output in a file called "nsorts.txt". Then, sort it alphabetically and place the output in a file called "asorts.txt". Are these files the same? Why or why not?

Challenge 2

The command `uniq` removes adjacent duplicated lines from its input.
For example, if a file `salmon.txt` contains:

```
coho
coho
steelhead
coho
steelhead
steelhead
```

then `uniq salmon.txt` produces:

```
coho
steelhead
coho
steelhead
```

You have a file called `data/animals.txt` contains the following data:

```
deer
rabbit
raccoon
rabbit
deer
fox
rabbit
bear
```

Process this list such that there is only one value for each animal, and save it in a file under my_files/ called "unique_animals.txt"

When you feel you have met these challenges successfully, cd into test/ and type

```
. 1-3_test.sh
```

Challenge 3

The command cut will cut out a sequence, for characters the argument -c must be added.

```
$ echo hello
hello
```

```
$ echo hello | cut -c 2-4
ell
```

How would you count all of the words contained in articles about asia found in data/articles/? Store this number under my_files/ in a file called "asia_count".

Adapted from: [Software Carpentry (http://software-carpentry.org/v5/novice/shell/03-pipefilter.html)](http://software-carpentry.org/v5/novice/shell/03-pipefilter.html)