Finding Things

Overview

Teaching: 25 min
Exercises: 20 min
Questions

- How can I find files?
- How can I find things in files?

Objectives

- Use `grep` to select lines from text files that match simple patterns.
- Use `find` to find files and directories whose names match simple patterns.
- Use the output of one command as the command-line argument(s) to another command.
- Explain what is meant by 'text' and 'binary' files, and why many common tools don't handle the latter well.

In the same way that many of us now use 'Google' as a verb meaning 'to find', Unix programmers often use the word 'grep'. 'grep' is a contraction of 'global/regular expression/print', a common sequence of operations in early Unix text editors. It is also the name of a very useful command-line program.

`grep` finds and prints lines in files that match a pattern. For our examples, we will use a file that contains three haikus taken from a 1998 competition in *Salon* magazine. For this set of examples, we're going to be working in the writing subdirectory:

Bash

```
$ cd
$ cd Desktop/data-shell/writing
$ cat haiku.txt
```

Output

```
The Tao that is seen
Is not the true Tao, until
You bring fresh toner.

With searching comes loss
and the presence of absence:
"My Thesis" not found.

Yesterday it worked
Today it is not working
Software is like that.
```

📌 Forever, or Five Years

We haven't linked to the original haikus because they don't appear to be on *Salon*'s site any longer. As Jeff Rothenberg said (https://www.clir.org/wp-content/uploads/sites/6/ensuring.pdf), 'Digital information lasts forever — or five years, whichever comes first.' Luckily, popular content often has backups (http://wiki.c2.com/?ComputerErrorHaiku).

Let's find lines that contain the word 'not':

Bash

```
$ grep not haiku.txt
```

Output

```
Is not the true Tao, until
"My Thesis" not found
Today it is not working
```

Here, `not` is the pattern we're searching for. The grep command searches through the file, looking for matches to the pattern specified. To use it type `grep`, then the pattern we're searching for and finally the name of the file (or files) we're searching in.

The output is the three lines in the file that contain the letters 'not'.

By default, grep searches for a pattern in a case-sensitive way. In addition, the search pattern we have selected does not have to form a complete word, as we will see in the next example.

Let's search for the pattern: 'The'.

Bash

```
$ grep The haiku.txt
```

Output

```
The Tao that is seen
"My Thesis" not found.
```

This time, two lines that include the letters ‘The’ are outputted, one of which contained our search pattern within a larger word, ‘Thesis’.

To restrict matches to lines containing the word ‘The’ on its own, we can give `grep` with the `-w` option. This will limit matches to word boundaries.

Later in this lesson, we will also see how we can change the search behavior of grep with respect to its case sensitivity.

Bash

```
$ grep -w The haiku.txt
```

Output

```
The Tao that is seen
```

Note that a ‘word boundary’ includes the start and end of a line, so not just letters surrounded by spaces. Sometimes we don’t want to search for a single word, but a phrase. This is also easy to do with `grep` by putting the phrase in quotes.

Bash

```
$ grep -w "is not" haiku.txt
```

Output

```
Today it is not working
```

We’ve now seen that you don’t have to have quotes around single words, but it is useful to use quotes when searching for multiple words. It also helps to make it easier to distinguish between the search term or phrase and the file being searched. We will use quotes in the remaining examples.

Another useful option is `-n`, which numbers the lines that match:

Bash

```
$ grep -n "it" haiku.txt
```

Output

```
5:With searching comes loss
9:Yesterday it worked
10:Today it is not working
```

Here, we can see that lines 5, 9, and 10 contain the letters ‘it’.

We can combine options (i.e. flags) as we do with other Unix commands. For example, let’s find the lines that contain the word ‘the’. We can combine the option `-w` to find the lines that contain the word ‘the’ and `-n` to number the lines that match:

Bash

```
$ grep -n -w "the" haiku.txt
```

Output

```
2:Is not the true Tao, until
6:and the presence of absence:
```

Now we want to use the option `-i` to make our search case-insensitive:

Bash

```
$ grep -n -w -i "the" haiku.txt
```

Output

```
1:The Tao that is seen
2:Is not the true Tao, until
6:and the presence of absence:
```

Now, we want to use the option `-v` to invert our search, i.e., we want to output the lines that do not contain the word 'the'.

Bash

```
$ grep -n -w -v "the" haiku.txt
```

Output

```
1:The Tao that is seen
3:You bring fresh toner.
4:
5:With searching comes loss
7:"My Thesis" not found.
8:
9:Yesterday it worked
10:Today it is not working
11:Software is like that.
```

`grep` has lots of other options. To find out what they are, we can type:

Bash

```
$ grep --help
```

Output

```
Usage: grep [OPTION]... PATTERN [FILE]...
Search for PATTERN in each FILE or standard input.
PATTERN is, by default, a basic regular expression (BRE).
Example: grep -i 'hello world' menu.h main.c

Regexp selection and interpretation:
  -E, --extended-regexp     PATTERN is an extended regular expression (ERE)
  -F, --fixed-strings       PATTERN is a set of newline-separated fixed strings
  -G, --basic-regexp        PATTERN is a basic regular expression (BRE)
  -P, --perl-regexp         PATTERN is a Perl regular expression
  -e, --regexp=PATTERN      use PATTERN for matching
  -f, --file=FILE           obtain PATTERN from FILE
  -i, --ignore-case         ignore case distinctions
  -w, --word-regexp         force PATTERN to match only whole words
  -x, --line-regexp         force PATTERN to match only whole lines
  -z, --null-data           a data line ends in 0 byte, not newline

Miscellaneous:
...        ...        ...
```

Using `grep`

Which command would result in the following output:

Output

```
and the presence of absence:
```

1. `grep "of" haiku.txt`
2. `grep -E "of" haiku.txt`
3. `grep -w "of" haiku.txt`
4. `grep -i "of" haiku.txt`

Solution

Wildcards

`grep`'s real power doesn't come from its options, though; it comes from the fact that patterns can include wildcards. (The technical name for these is **regular expressions**, which is what the 're' in 'grep' stands for.) Regular expressions are both complex and powerful; if you want to do complex searches, please look at the lesson on our website (http://v4.software-carpentry.org/regexp/index.html). As a taster, we can find lines that have an 'o' in the second position like this:

Bash

```
$ grep -E "^.o" haiku.txt
```

Output

```
You bring fresh toner.
Today it is not working
Software is like that.
```

We use the `-E` option and put the pattern in quotes to prevent the shell from trying to interpret it. (If the pattern contained a `*`, for example, the shell would try to expand it before running `grep`.) The `^` in the pattern anchors the match to the start of the line. The `.` matches a single character (just like `?` in the shell), while the `o` matches an actual 'o'.

Tracking a Species

Leah has several hundred data files saved in one directory, each of which is formatted like this:

Code

```
2013-11-05,deer,5
2013-11-05,rabbit,22
2013-11-05,raccoon,7
2013-11-06,rabbit,19
2013-11-06,deer,2
```

She wants to write a shell script that takes a species as the first command-line argument and a directory as the second argument. The script should return one file called `species.txt` containing a list of dates and the number of that species seen on each date. For example using the data shown above, `rabbit.txt` would contain:

Code

```
2013-11-05,22
2013-11-06,19
```

Put these commands and pipes in the right order to achieve this:

Bash

```bash
cut -d : -f 2
>
|
grep -w $1 -r $2
|
$1.txt
cut -d , -f 1,3
```

Hint: use `man grep` to look for how to grep text recursively in a directory and `man cut` to select more than one field in a line.

An example of such a file is provided in `data-shell/data/animal-counts/animals.txt`

Solution

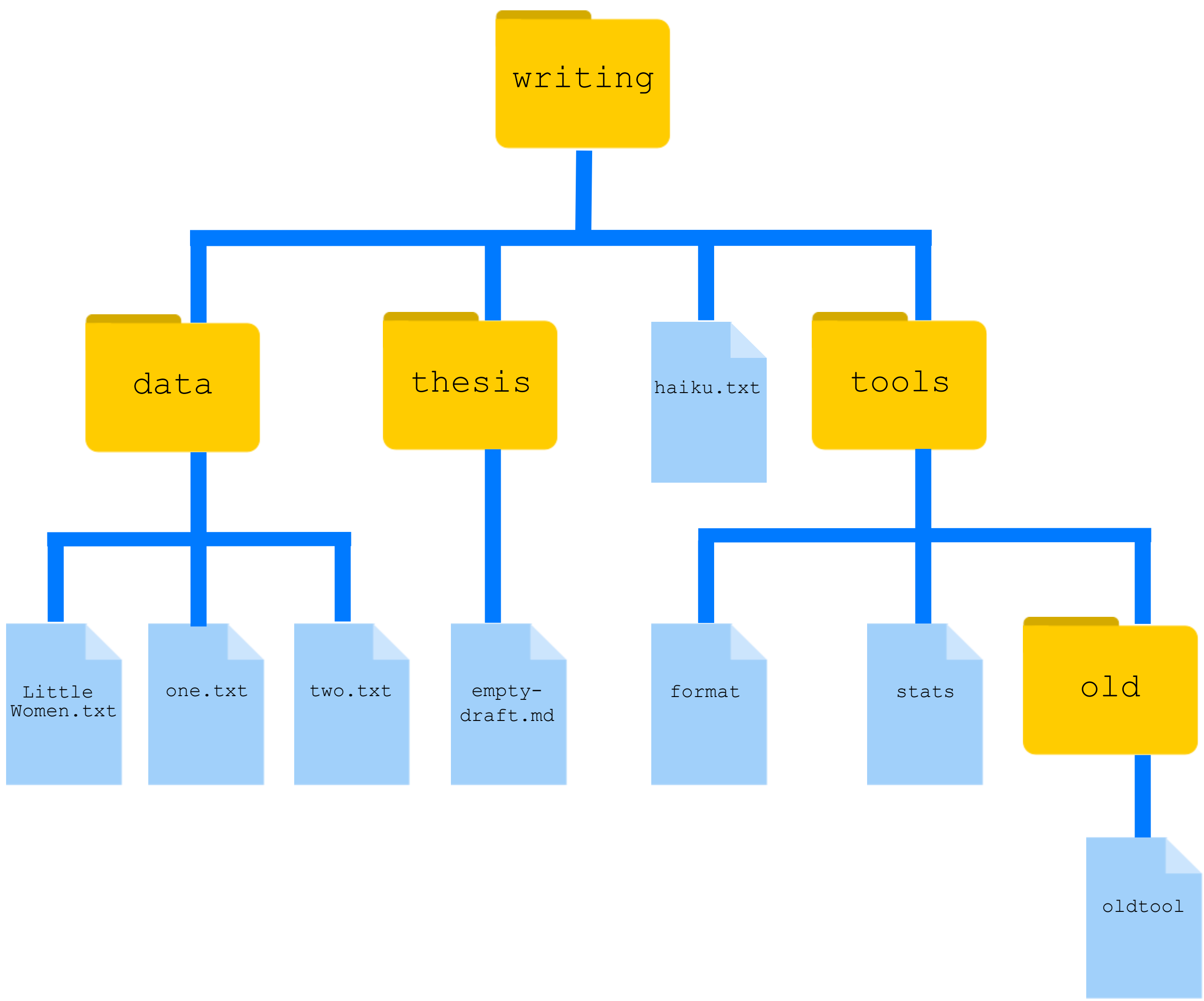
Little Women

You and your friend, having just finished reading *Little Women* by Louisa May Alcott, are in an argument. Of the four sisters in the book, Jo, Meg, Beth, and Amy, your friend thinks that Jo was the most mentioned. You, however, are certain it was Amy. Luckily, you have a file `LittleWomen.txt` containing the full text of the novel (`data-shell/writing/data/LittleWomen.txt`). Using a `for` loop, how would you tabulate the number of times each of the four sisters is mentioned?

Hint: one solution might employ the commands `grep` and `wc` and a `|` , while another might utilize `grep` options. There is often more than one way to solve a programming task, so a particular solution is usually chosen based on a combination of yielding the correct result, elegance, readability, and speed.

Solutions

While `grep` finds lines in files, the `find` command finds files themselves. Again, it has a lot of options; to show how the simplest ones work, we'll use the directory tree shown below.



Nelle's `writing` directory contains one file called `haiku.txt` and three subdirectories: `thesis` (which contains a sadly empty file, `empty-draft.md`); `data` (which contains three files `LittleWomen.txt` , `one.txt` and `two.txt`); and a `tools` directory that contains the programs `format` and `stats` , and a subdirectory called `old` , with a file `oldtool` .

For our first command, let's run `find .` (remember to run this command from the `data-shell/writing` folder).

Bash

```
$ find .
```

Output

```
.
./data
./data/one.txt
./data/LittleWomen.txt
./data/two.txt
./tools
./tools/format
./tools/old
./tools/old/oldtool
./tools/stats
./haiku.txt
./thesis
./thesis/empty-draft.md
```

As always, the `.` on its own means the current working directory, which is where we want our search to start. `find` 's output is the names of every file **and** directory under the current working directory. This can seem useless at first but `find` has many options to filter the output and in this lesson we will discover some of them.

The first option in our list is `-type d` that means 'things that are directories'. Sure enough, `find` 's output is the names of the five directories in our little tree (including `.`):

Bash

```
$ find . -type d
```

Output

```
./
./data
./thesis
./tools
./tools/old
```

Notice that the objects `find` finds are not listed in any particular order. If we change `-type d` to `-type f` , we get a listing of all the files instead:

Bash

```
$ find . -type f
```

Output

```
./haiku.txt
./tools/stats
./tools/old/oldtool
./tools/format
./thesis/empty-draft.md
./data/one.txt
./data/LittleWomen.txt
./data/two.txt
```

Now let's try matching by name:

Bash

```
$ find . -name *.txt
```

Output

```
./haiku.txt
```

We expected it to find all the text files, but it only prints out `./haiku.txt`. The problem is that the shell expands wildcard characters like `*` *before* commands run. Since `*.txt` in the current directory expands to `haiku.txt`, the command we actually ran was:

Bash

```
$ find . -name haiku.txt
```

`find` did what we asked; we just asked for the wrong thing.

To get what we want, let's do what we did with `grep`: put `*.txt` in quotes to prevent the shell from expanding the `*` wildcard. This way, `find` actually gets the pattern `*.txt`, not the expanded filename `haiku.txt`:

Bash

```
$ find . -name "*.txt"
```

Output

```
./data/one.txt
./data/LittleWomen.txt
./data/two.txt
./haiku.txt
```

📌 Listing vs. Finding

`ls` and `find` can be made to do similar things given the right options, but under normal circumstances, `ls` lists everything it can, while `find` searches for things with certain properties and shows them.

As we said earlier, the command line's power lies in combining tools. We've seen how to do that with pipes; let's look at another technique. As we just saw, `find . -name "*.txt"` gives us a list of all text files in or below the current directory. How can we combine that with `wc -l` to count the lines in all those files?

The simplest way is to put the `find` command inside `$()` :

Bash

```
$ wc -l $(find . -name "*.txt")
```

Output

```
11 ./haiku.txt
300 ./data/two.txt
21022 ./data/LittleWomen.txt
70 ./data/one.txt
21403 total
```

When the shell executes this command, the first thing it does is run whatever is inside the `$()` . It then replaces the `$()` expression with that command's output. Since the output of `find` is the four filenames `./data/one.txt` , `./data/LittleWomen.txt` , `./data/two.txt` , and `./haiku.txt` , the shell constructs the command:

Bash

```
$ wc -l ./data/one.txt ./data/LittleWomen.txt ./data/two.txt ./haiku.txt
```

which is what we wanted. This expansion is exactly what the shell does when it expands wildcards like `*` and `?` , but lets us use any command we want as our own 'wildcard'.

It's very common to use `find` and `grep` together. The first finds files that match a pattern; the second looks for lines inside those files that match another pattern. Here, for example, we can find PDB files that contain iron atoms by looking for the string 'FE' in all the `.pdb` files above the current directory:

Bash

```
$ grep "FE" $(find .. -name "*.pdb")
```

Output

```
../data/pdb/heme.pdb:ATOM     25 FE           1      -0.924   0.535  -0.518
```

Matching and Subtracting

The `-v` option to `grep` inverts pattern matching, so that only lines which do *not* match the pattern are printed. Given that, which of the following commands will find all files in `/data` whose names end in `s.txt` but whose names also do *not* contain the string `net`? (For example, `animals.txt` or `amino-acids.txt` but not `planets.txt`.) Once you have thought about your answer, you can test the commands in the `data-shell` directory.

1. `find data -name "*s.txt" | grep -v net`
2. `find data -name *s.txt | grep -v net`
3. `grep -v "net" $(find data -name "*s.txt")`
4. None of the above.

Solution

Binary Files

We have focused exclusively on finding patterns in text files. What if your data is stored as images, in databases, or in some other format?

A handful of tools extend `grep` to handle a few non text formats. But a more generalizable approach is to convert the data to text, or extract the text-like elements from the data. On the one hand, it makes simple things easy to do. On the other hand, complex things are usually impossible. For example, it's easy enough to write a program that will extract X and Y dimensions from image files for `grep` to play with, but how would you write something to find values in a spreadsheet whose cells contained formulas?

A last option is to recognize that the shell and text processing have their limits, and to use another programming language. When the time comes to do this, don't be too hard on the shell: many modern programming languages have borrowed a lot of ideas from it, and imitation is also the sincerest form of praise.

The Unix shell is older than most of the people who use it. It has survived so long because it is one of the most productive programming environments ever created — maybe even *the* most productive. Its syntax may be cryptic, but people who have mastered it can experiment with different commands interactively, then use what they have learned to automate their work. Graphical user interfaces may be better at the first, but the shell is still unbeaten at the second. And as Alfred North Whitehead wrote in 1911, 'Civilization advances by extending the number of important operations which we can perform without thinking about them.'

`find` Pipeline Reading Comprehension

Write a short explanatory comment for the following shell script:

Bash

```
wc -l $(find . -name "*.dat") | sort -n
```

Solution

Key Points

- `find` finds files with specific properties that match patterns.
- `grep` selects lines in files that match patterns.
- `--help` is an option supported by many bash commands, and programs that can be run from within Bash, to display more information on how to use these commands or programs.
- `man command` displays the manual page for a given command.
- `$(command)` inserts a command's output in place.

(../06-script/index.html)

(../)