Finding Things

Objectives

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

You can guess someone's age by how they talk about search: young people use "Google" as a verb, while crusty old Unix programmers use "grep". The word 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:

```
$ cd
$ cd writing
$ cat haiku.txt
```

```
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 (http://www.clir.org/pubs/archives/ensuring.pdf), "Digital information lasts forever—or five years, whichever comes first."

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

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

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

Here, `not` is the pattern we're searching for. It's pretty simple: every alphanumeric character matches against itself. After the pattern comes the name or names of the files we're searching in. The output is the three lines in the file that contain the letters "not".

Let's try a different pattern: "day".

```
$ grep day haiku.txt
```

```
Yesterday it worked
Today it is not working
```

This time, the output is lines containing the words "Yesterday" and "Today", which both have the letters "day". If we give `grep` the `-w` flag, it restricts matches to word boundaries, so that only lines with the word "day" will be printed:

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

In this case, there aren't any, so `grep`'s output is empty.

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

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

```
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 flags as we do with other Unix commands. For example, since `-i` makes matching case-insensitive and `-v` inverts the match, using them both only prints lines that *don't* match the pattern in any mix of upper and lower case:

```
$ grep -i -v the haiku.txt
```

```
You bring fresh toner.

With searching comes loss

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

`grep` has lots of other options. To find out what they are, we can type `man grep`. `man` is the Unix "manual" command: it prints a description of a command and its options, and (if you're lucky) provides a few examples of how to use it:

```
$ man grep
```

```
GREP(1)
GREP(1)

NAME
grep, egrep, fgrep - print lines matching a pattern

SYNOPSIS
grep [OPTIONS] PATTERN [FILE...]
grep [OPTIONS] [-e PATTERN | -f FILE] [FILE...]

DESCRIPTION
grep  searches the named input FILEs (or standard input if no files are named, or if a single hyphen-
minus (-) is given as file name) for lines containing a match to the given PATTERN.  By default, grep
prints the matching lines.
...        ...        ...

OPTIONS
Generic Program Information
--help Print  a  usage  message  briefly summarizing these command-line options and the bug-reporting
address, then exit.

-V, --version
Print the version number of grep to the standard output stream.  This version number should be
included in all bug reports (see below).

Matcher Selection
-E, --extended-regexp
Interpret  PATTERN  as  an  extended regular expression (ERE, see below).  (-E is specified by
POSIX.)

-F, --fixed-strings
Interpret PATTERN as a list of fixed strings, separated by newlines, any of  which  is  to  be
matched.  (-F is specified by POSIX.)
...        ...        ...
```

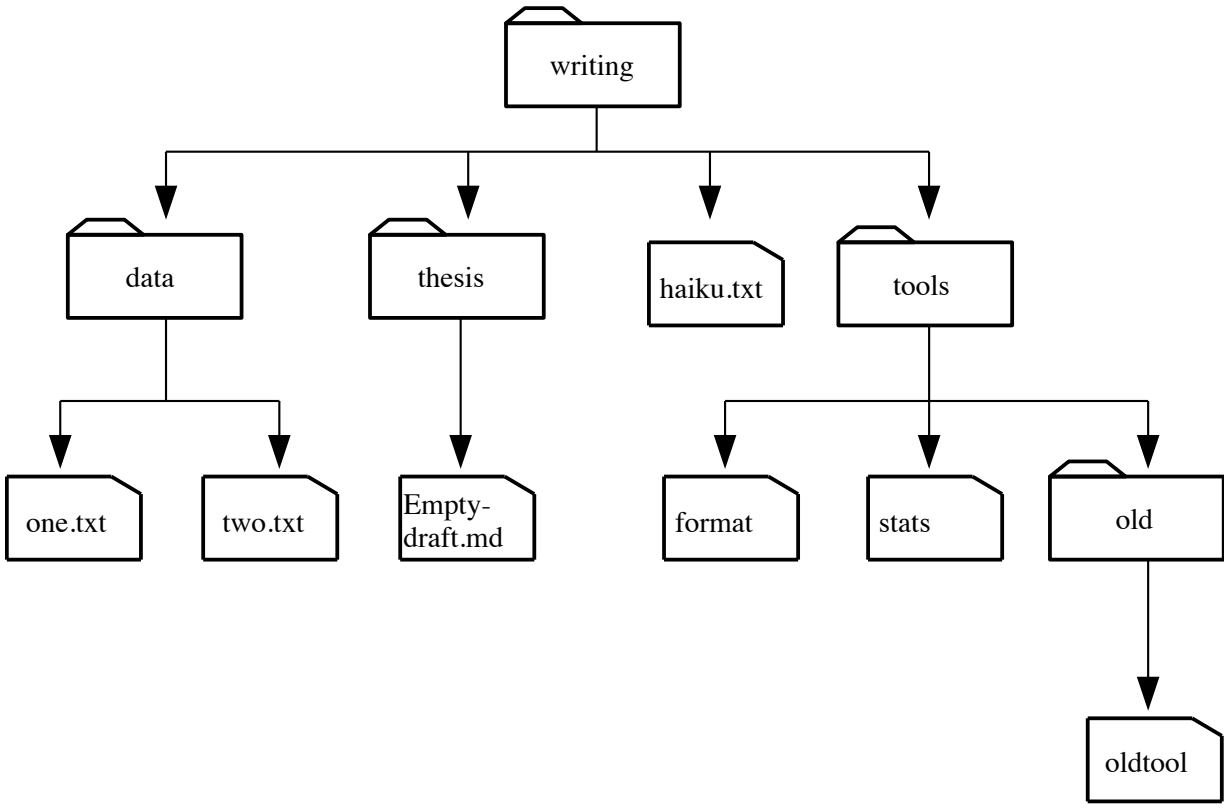
Wildcards

`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 (../../gloss.html#regular-expression), 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://software-carpentry.org). As a taster, we can find lines that have an 'o' in the second position like this:

```
$ grep -E '^.o' haiku.txt
You bring fresh toner.
Today it is not working
Software is like that.
```

We use the `-E` flag 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'.

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 four subdirectories: `thesis` (which is sadly empty), `data` (which contains two files `one.txt` and `two.txt`), a `tools` directory that contains the programs `format` and `stats`, and an empty subdirectory called `old`.

For our first command, let's run `find . -type d`. As always, the `.` on its own means the current working directory, which is where we want our search to start; `-type d` means "things that are directories". Sure enough, `find`'s output is the names of the five directories in our little tree (including `.`):

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

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

If we change `-type d` to `-type f`, we get a listing of all the files instead:

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

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

`find` automatically goes into subdirectories, their subdirectories, and so on to find everything that matches the pattern we've given it. If we don't want it to, we can use `-maxdepth` to restrict the depth of search:

```
$ find . -maxdepth 1 -type f
```

```
./haiku.txt
```

The opposite of `-maxdepth` is `-mindepth`, which tells `find` to only report things that are at or below a certain depth. `-mindepth 2` therefore finds all the files that are two or more levels below us:

```
$ find . -mindepth 2 -type f
```

```
./data/one.txt
./data/two.txt
./tools/format
./tools/stats
```

Now let's try matching by name:

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

```
./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:

```
$ 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 single quotes to prevent the shell from expanding the `*` wildcard. This way, `find` actually gets the pattern `*.txt`, not the expanded filename `haiku.txt`:

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

```
./data/one.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 `$()`:

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

```
11 ./haiku.txt
300 ./data/two.txt
70 ./data/one.txt
381 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 three filenames `./data/one.txt`, `./data/two.txt`, and `./haiku.txt`, the shell constructs the command:

```
$ wc -l ./data/one.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:

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

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

Binary Files

We have focused exclusively on finding things in text files. What if your data is stored as images, in databases, or in some other format? One option would be to extend tools like `grep` to handle those formats. This hasn't happened, and probably won't, because there are too many formats to support.
The second option is to convert the data to text, or extract the text-ish bits from the data. This is probably the most common approach, since it only requires people to build one tool per data format (to extract information). On the one hand, it makes simple things easy to do. On the negative side, 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?
The third choice is to recognize that the shell and text processing have their limits, and to use a programming language such as Python instead. When the time comes to do this, don't be too hard on the shell: many modern programming languages, Python included, have borrowed a lot of ideas from it, and imitation is also the sincerest form of praise.

Conclusion

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."

Key Points

- Use `find` to find files and directories, and `grep` to find text patterns in files.
- `$(command)` inserts a command's output in place.
- `man command` displays the manual page for a given command.

Write a short explanatory comment for the following shell script:

```
find . -name '*.dat' | wc -l | sort -n
```

The `-v` flag 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 `ose.dat` (e.g., `sucrose.dat` or `maltose.dat`), but do *not* contain the word `temp`?

1. `find /data -name '*.dat' | grep ose | grep -v temp`
2. `find /data -name ose.dat | grep -v temp`
3. `grep -v temp $(find /data -name '*ose.dat')`
4. None of the above.