1. rm rmdir

If we try to remove the entire thesis directory using rm thesis, we get an error message:

$ rm thesis

rm: cannot remove `thesis': Is a directory

This happens because rm only works on files, not directories.

The right command is rmdir, which is short for “remove directory”.

It doesn’t work yet either, though,

because the directory we’re trying to remove isn’t empty:

$ rmdir thesis

rmdir: failed to remove `thesis': Directory not empty

This little safety feature can save you a lot of grief, particularly if you are a bad typist. To really get rid of thesis we must first delete the file draft.txt:

$ rm thesis/draft.txt

The directory is now empty, so rmdir can delete it:

$ rmdir thesis

With Great Power Comes Great Responsibility

Removing the files in a directory just so that we can remove the directory quickly becomes tedious. Instead, we can use rm with the -r flag (which stands for “recursive”):

$ rm -r thesis

This removes everything in the directory, then the directory itself. If the directory contains sub-directories, rm -r does the same thing to them, and so on. It’s very handy, but can do a lot of damage if used without care.

Let’s go into that directory with cd and run the command wc \*.pdb.

wc is the “word count” command: it counts the number of

lines,

words,

and characters in files. The \* in \*.pdb matches zero or more characters, so the shell turns \*.pdb into a complete list of .pdb files:

$ cd molecules

$ wc \*.pdb

20 156 1158 cubane.pdb

12 84 622 ethane.pdb

9 57 422 methane.pdb

30 246 1828 octane.pdb

21 165 1226 pentane.pdb

15 111 825 propane.pdb

107 819 6081 total

## Wildcards

\* is a **wildcard**. It matches zero or more characters, so \*.pdb matches ethane.pdb, propane.pdb, and so on. On the other hand, p\*.pdb only matches pentane.pdb and propane.pdb, because the ‘p’ at the front only matches itself.

? is also a wildcard, but it only matches a single character. This means that p?.pdb matches pi.pdb or p5.pdb, but not propane.pdb. We can use any number of wildcards at a time: for example, p\*.p?\* matches anything that starts with a ‘p’ and ends with ‘.’, ‘p’, and at least one more character (since the ‘?’ has to match one character, and the final ‘\*’ can match any number of characters). Thus, p\*.p?\* would match preferred.practice, and even p.pi (since the first ‘\*’ can match no characters at all), but not quality.practice (doesn’t start with ‘p’) or preferred.p (there isn’t at least one character after the ‘.p’).

When the shell sees a wildcard,

it expands the wildcard to create a list of matching filenames before running the command that was asked for. As an exception, if a wildcard expression does not match any file, Bash will pass the expression as a parameter to the command as it is. For example typing ls \*.pdf in the molecules directory (which contains only files with names ending with .pdb) results in an error message that there is no file called \*.pdf.

However, generally commands like wc and ls see the lists of file names matching these expressions, but not the wildcards themselves.

It is the shell, not the other programs, that deals with expanding wildcards, and this another example of orthogonal design.

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

$ wc -l \*.pdb

20 cubane.pdb

12 ethane.pdb

9 methane.pdb

30 octane.pdb

21 pentane.pdb

15 propane.pdb

107 total

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 six files,

but what if there were 6000?

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

$ wc -l \*.pdb > lengths.txt

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.txt

9 methane.pdb

12 ethane.pdb

15 propane.pdb

20 cubane.pdb

21 pentane.pdb

30 octane.pdb

107 total

We can put the sorted list of lines in another temporary file called sorted-lengths.txt

by putting > sorted-lengths.txt after the command,

just as we used > lengths.txt to put the output of wc into lengths.txt.

Once we’ve done that,

we can run another command called head

 to get the first few lines in sorted-lengths.txt:

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.txt | head -1

9 methane.pdb

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 -l \*.pdb | sort -n | head -1

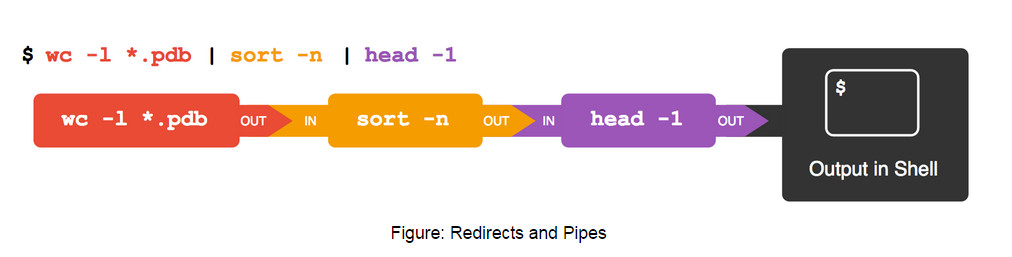
9 methane.pdb

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 line count of \*.pdb”.

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 -l \*.pdb > lengths.txt. 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 -l \*.pdb | sort -n instead, the shell creates two processes (one for each process in the pipe) so that wc andsort 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 -l \*.pdb | 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 ammonia.pdb, we could write wc < ammonia.pdb.

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 ammonia.pdb to wc’s standard input.

### Nelle’s Pipeline: Checking Files

Nelle has run her samples through the assay machines and created 1520 files in the north-pacific-gyre/2012-07-03directory described earlier. As a quick sanity check, starting from her home directory, Nelle types:

$ cd north-pacific-gyre/2012-07-03

$ wc -l \*.txt

The output is 1520 lines that look like this:

300 NENE01729A.txt

300 NENE01729B.txt

300 NENE01736A.txt

300 NENE01751A.txt

300 NENE01751B.txt

300 NENE01812A.txt

... ...

Now she types this:

$ wc -l \*.txt | sort -n | head -5

Whoops: one of the files is 60 lines shorter than the others. When she goes back and checks it, she sees that she did that assay at 8:00 on a Monday morning — someone was probably in using the machine on the weekend, and she forgot to reset it. Before re-running that sample, she checks to see if any files have too much data:

$ wc -l \*.txt | sort -n | tail -5

300 NENE02040A.txt

300 NENE02040B.txt

300 NENE02040Z.txt

300 NENE02043A.txt

300 NENE02043B.txt

Those numbers look good — but what’s that ‘Z’ doing there in the third-to-last line? All of her samples should be marked ‘A’ or ‘B’; by convention, her lab uses ‘Z’ to indicate samples with missing information. To find others like it, she does this:

Instead, we can use a **loop** to do some operation once for each thing in a list. Here’s a simple example that displays the first three lines of each file in turn:

$ for filename in basilisk.dat unicorn.dat

> do

> head -3 $filename

> done

COMMON NAME: basilisk

CLASSIFICATION: basiliscus vulgaris

UPDATED: 1745-05-02

COMMON NAME: unicorn

CLASSIFICATION: equus monoceros

UPDATED: 1738-11-24

When the shell sees the keyword for,

it knows it is supposed to repeat a command (or group of commands) once for each thing in a list.

In this case, the list is the two filenames.

Each time through the loop, the name of the thing currently being operated on is assigned to the **variable** called filename.

Inside the loop, we get the variable’s value

by putting $ in front of it:

$filename is basilisk.dat

the first time through the loop, unicorn.dat the second, and so on.

By using the dollar sign we are telling the shell interpreter to treat filename as a variable name and substitute its value on its place, but not as some text or external command. When using variables it is also possible to put the names into curly braces to clearly delimit the variable name: $filename is equivalent to ${filename}, but is different from ${file}name. You may find this notation in other people’s programs.

Finally, the command that’s actually being run is our old friend head, so this loop prints out the first three lines of each data file in turn.

### Nelle’s Pipeline: Checking Files

Nelle has run her samples through the assay machines and created 1520 files in the north-pacific-gyre/2012-07-03 directory described earlier. As a quick sanity check, starting from her home directory, Nelle types:

$ cd north-pacific-gyre/2012-07-03

$ wc -l \*.txt

The output is 1520 lines that look like this:

300 NENE01729A.txt

300 NENE01729B.txt

300 NENE01736A.txt

300 NENE01751A.txt

300 NENE01751B.txt

300 NENE01812A.txt

... ...

Now she types this:

$ wc -l \*.txt | sort -n | head -5

240 NENE02018B.txt

300 NENE01729A.txt

300 NENE01729B.txt

300 NENE01736A.txt

300 NENE01751A.txt

Whoops: one of the files is 60 lines shorter than the others. When she goes back and checks it, she sees that she did that assay at 8:00 on a Monday morning — someone was probably in using the machine on the weekend, and she forgot to reset it. Before re-running that sample, she checks to see if any files have too much data:

$ wc -l \*.txt | sort -n | tail -5

300 NENE02040A.txt

300 NENE02040B.txt

300 NENE02040Z.txt

300 NENE02043A.txt

300 NENE02043B.txt

## Loops

Objective :

* Write a loop

that applies one or more commands separately to each file

in a set of files.

* Trace the values taken on

by a loop variable

during execution of the loop.

* Explain the difference between a variable’s name and its value.
* Explain why spaces and some punctuation characters shouldn’t be used in files’ names.
* Demonstrate how to see what commands have recently been executed.
* Re-run recently executed commands without retyping them.

Wildcards and tab completion are two ways to reduce typing (and typing mistakes). Another is to tell the shell to do something over and over again.

Suppose we have several hundred genome data files named basilisk.dat, unicorn.dat, and so on.

In this example, we’ll use the creatures directory which only has two example files, but the principles can be applied to many many more files at once.

We would like to modify these files,

but also save a version of the original files and rename them as original-basilisk.dat and original-unicorn.dat. We can’t use:

$ mv \*.dat original-\*.dat

because that would expand to:

$ mv basilisk.dat unicorn.dat original-\*.dat

This wouldn’t back up our files, instead we get an error

mv: target `original-\*.dat' is not a directory

This a problem arises when mv receives more than two inputs.

When this happens, it expects the last input to be a directory

where it can move all the files it was passed.

Since there is no directory named original-\*.dat in the creatures directory we get an error.

Instead, we can use a **loop** to do some operation once for each thing in a list.

Here’s a simple example that displays the first three lines of each file in turn:

$ for filename in basilisk.dat unicorn.dat

> do

> head -3 $filename

> done

COMMON NAME: basilisk

CLASSIFICATION: basiliscus vulgaris

UPDATED: 1745-05-02

COMMON NAME: unicorn

CLASSIFICATION: equus monoceros

UPDATED: 1738-11-24

When the shell sees the keyword for,

it knows it is supposed to repeat a command (or group of commands) once for each thing in a list.

In this case, the list is the two filenames.

Each time through the loop, the name of the thing currently being operated on is assigned to the **variable** called filename.

Inside the loop, we get the variable’s value

by putting $ in front of it:

 $filename is basilisk.dat the first time through the loop,

unicorn.dat the second, and so on.

By using the dollar sign we are telling the shell interpreter to treat filename as a variable name and substitute its value on its place, but not as some text or external command.

When using variables it is also possible to put the names into curly braces to clearly delimit the variable name: $filename is equivalent to ${filename}, but is different from ${file}name. You may find this notation in other people’s programs.

Finally, the command that’s actually being run is our old friend head, so this loop prints out the first three lines of each data file in turn.

## Follow the Prompt

The shell prompt changes from $ to > and back again

as we were typing in our loop.

The second prompt, >, is different to remind us that we haven’t finished typing a complete command yet.

We have called the variable in this loop filename in order to make its purpose clearer to human readers. The shell itself doesn’t care what the variable is called; if we wrote this loop as:

for x in basilisk.dat unicorn.dat

do

head -3 $x

done

or:

for temperature in basilisk.dat unicorn.dat

do

head -3 $temperature

done

it would work exactly the same way. *Don’t do this.* Programs are only useful if people can understand them, so meaningless names (like x) or misleading names (like temperature) increase the odds that the program won’t do what its readers think it does.

Here’s a slightly more complicated loop:

for filename in \*.dat

do

echo $filename

head -100 $filename | tail -20

done

The shell starts by expanding \*.dat to create the list of files it will process. The **loop body** then executes two commands for each of those files. The first, echo, just prints its command-line parameters to standard output. For example:

$ echo hello there

prints:

hello there

In this case, since the shell expands $filename to be the name of a file,

echo $filename  just prints the name of the file. Note that we can’t write this as:

for filename in \*.dat

do

$filename

head -100 $filename | tail -20

done

because then the first time through the loop,

when $filename expanded to basilisk.dat,

the shell would try to run basilisk.dat as a program.

Finally, the head and tail combination selects lines 81-100 from whatever file is being processed.

## Spaces in Names

Filename expansion in loops is another reason you should not use spaces in filenames. Suppose our data files are named:

basilisk.dat

red dragon.dat

unicorn.dat

If we try to process them using:

for filename in \*.dat

do

head -100 $filename | tail -20

done

then the shell will expand \*.dat to create:

basilisk.dat red dragon.dat unicorn.dat

With older versions of Bash, or most other shells, filename will then be assigned the following values in turn:

basilisk.dat

red

dragon.dat

unicorn.dat

That’s a problem: head can’t read files called red and dragon.dat because they don’t exist, and won’t be asked to read the file red dragon.dat.

We can make our script a little bit more robust by **quoting** our use of the variable:

for filename in \*.dat

do

head -100 "$filename" | tail -20

done

but it’s simpler just to avoid using spaces (or other special characters) in filenames.

Going back to our original file renaming problem, we can solve it using this loop:

for filename in \*.dat

do

mv $filename original-$filename

done

## Measure Twice, Run Once

A loop is a way to do many things at once — or to make many mistakes at once if it does the wrong thing.

One way to check what a loop would do is to echo the commands it would run instead of actually running them.

For example, we could write our file renaming loop like this:

for filename in \*.dat

do

echo mv $filename original-$filename

done

Instead of running mv, this loop runs echo, which prints out:

mv basilisk.dat original-basilisk.dat

mv unicorn.dat original-unicorn.dat

without actually running those commands.

We can then use up-arrow to redisplay the loop, back-arrow to get to the word echo, delete it, and then press “enter” to run the loop with the actual mv commands.

This isn’t foolproof, but it’s a handy way

to see what’s going to happen

when you’re still learning how loops work.

## Nelle’s Pipeline: Processing Files

Nelle is now ready to process her data files.

Since she’s still learning how to use the shell,

she decides to build up the required commands in stages.

Her first step is to make sure that

she can select the right files — remember, these are ones whose names end in ‘A’ or ‘B’, rather than ‘Z’.

Starting from her home directory, Nelle types:

$ cd north-pacific-gyre/2012-07-03

$ for datafile in \*[AB].txt

> do

> echo $datafile

> done

NENE01729A.txt

NENE01729B.txt

NENE01736A.txt

...

NENE02043A.txt

NENE02043B.txt

Her next step is to decide what to call the files

that the goostats analysis program will create.

Prefixing each input file’s name with “stats” seems simple, so she modifies her loop to do that:

$ for datafile in \*[AB].txt

> do

> echo $datafile stats-$datafile

> done

NENE01729A.txt stats-NENE01729A.txt

NENE01729B.txt stats-NENE01729B.txt

NENE01736A.txt stats-NENE01736A.txt

...

NENE02043A.txt stats-NENE02043A.txt

NENE02043B.txt stats-NENE02043B.txt

She hasn’t actually run goostats yet, but now she’s sure she can select the right files and generate the right output filenames.

Typing in commands over and over again is becoming tedious,

though, and Nelle is worried about making mistakes, so instead of re-entering her loop, she presses the up arrow.

In response, the shell redisplays the whole loop on one line (using semi-colons to separate the pieces):

$ for datafile in \*[AB].txt; do echo $datafile stats-$datafile; done

Using the left arrow key, Nelle backs up and changes the command echo to goostats:

$ for datafile in \*[AB].txt; do bash goostats $datafile stats-$datafile; done

When she presses enter, the shell runs the modified command. However, nothing appears to happen — there is no output.

After a moment, Nelle realizes that since her script doesn’t print anything to the screen any longer, she has no idea whether it is running, much less how quickly.

She kills the job by typing Control-C, uses up-arrow to repeat the command, and edits it to read:

$ for datafile in \*[AB].txt; do echo $datafile; bash goostats $datafile stats-$datafile; done

## Beginning and End

We can move to the beginning of a line in the shell by typing ^A (which means Control-A) and to the end using ^E.

When she runs her program now, it produces one line of output every five seconds or so:

NENE01729A.txt

NENE01729B.txt

NENE01736A.txt

...

1518 times 5 seconds, divided by 60, tells her that her script will take about two hours to run. As a final check, she opens another terminal window, goes into north-pacific-gyre/2012-07-03, and uses cat stats-NENE01729B.txt to examine one of the output files. It looks good, so she decides to get some coffee and catch up on her reading.

## Those Who Know History Can Choose to Repeat It

Another way to repeat previous work is to use the history command to get a list of the last few hundred commands that have been executed, and then to use !123 (where “123” is replaced by the command number) to repeat one of those commands. For example, if Nelle types this:

$ history | tail -5

456 ls -l NENE0\*.txt

457 rm stats-NENE01729B.txt.txt

458 bash goostats NENE01729B.txt stats-NENE01729B.txt

459 ls -l NENE0\*.txt

460 history

then she can re-run goostats on NENE01729B.txt simply by typing !458.

# [The Unix Shell](http://dgasmith.github.io/2015-08-24-stanford/shell/index.html)

Shell Scripts

## Learning Objectives

* Write a shell script

that runs a command or series of commands

for a fixed set of files.

* Run a shell script from the command line.
* Write a shell script

that operates on a set of files defined by the user on the command line.

* Create pipelines that include user-written shell scripts.

We are finally ready to see what makes the shell such a powerful programming environment.

We are going to take the commands

we repeat frequently and save them in files

so that we can re-run all those operations again later

by typing a single command.

For historical reasons, a bunch of commands saved in a file is usually called a **shell script**,

but make no mistake: these are actually small programs.

Let’s start by going back to molecules/ and putting the following line in the

file middle.sh:

$ cd molecules

$ cat middle.sh

head -15 octane.pdb | tail -5

This is a variation on the pipe we constructed earlier: it selects lines 11-15 of the file octane.pdb.

Remember, we are *not* running it as a command just yet: we are putting the commands in a file.

Once we have saved the file,

we can ask the shell to execute the commands it contains.

Our shell is called bash, so we run the following command:

$ bash middle.sh

ATOM 9 H 1 -4.502 0.681 0.785 1.00 0.00

ATOM 10 H 1 -5.254 -0.243 -0.537 1.00 0.00

ATOM 11 H 1 -4.357 1.252 -0.895 1.00 0.00

ATOM 12 H 1 -3.009 -0.741 -1.467 1.00 0.00

ATOM 13 H 1 -3.172 -1.337 0.206 1.00 0.00

Sure enough, our script’s output is exactly what we would get

if we ran that pipeline directly.

# [The Unix Shell](http://dgasmith.github.io/2015-08-24-stanford/shell/index.html)

## Finding Things

## Learning 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 inSalon 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, two lines that include the letters “day” are outputted.

However, these letters are contained within larger words.

To restrict matches to lines containing the word “day” on its own,

we can give grep with the -w flag.

This will limit matches to word boundaries.

$ grep -w day haiku.txt

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

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.

$ grep -w "is not" haiku.txt

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:

$ 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 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:

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

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:

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

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”.

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

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.

## 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://software-carpentry.org/v4/regexp/index.html).

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.

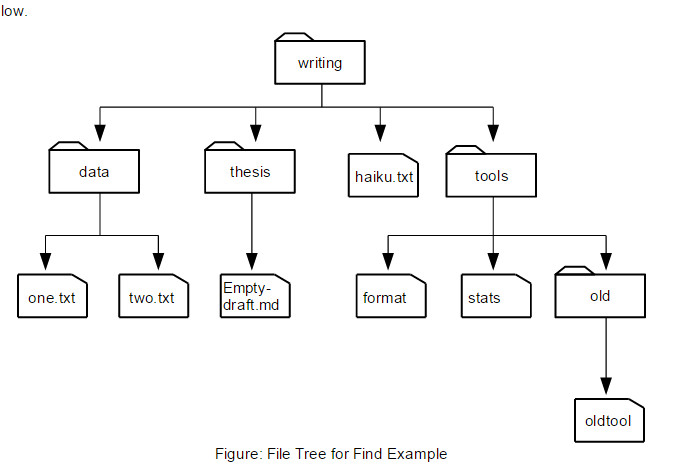
The ‘.’ matches a single character (just like ‘?’ in the shell),

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 filesone.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

(where) (what)

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.

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.”