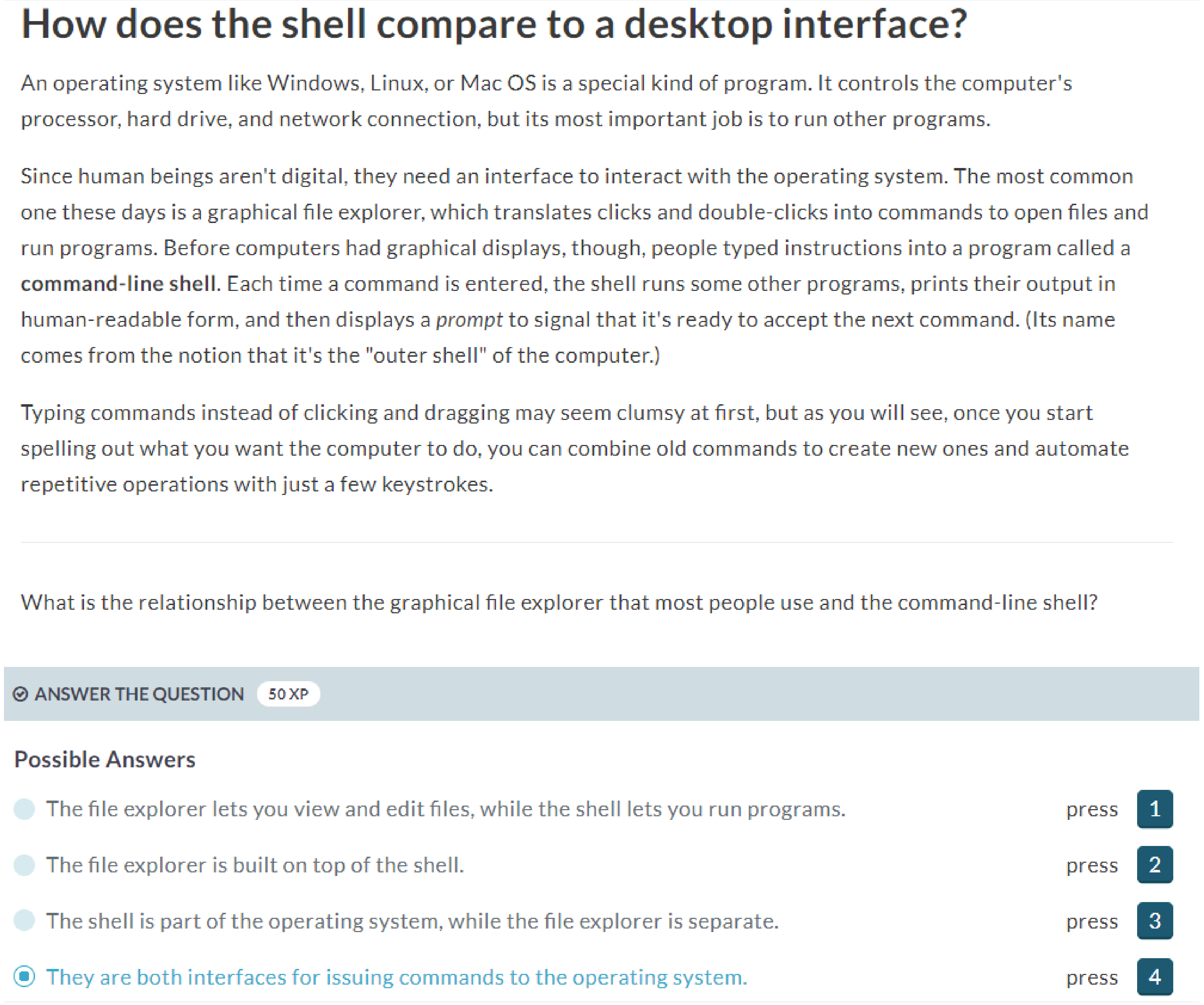
INTRODUCTION TO SHELL FOR DATA SCIENCE

研究助理 陳躍中 製作 20180803



How does the shell compare to a desktop interface?

An operating system like Windows, Linux, or Mac OS is a special kind of program. It controls the computer's processor, hard drive, and network connection, but its most important job is to run other programs.

Since human beings aren't digital, they need an interface to interact with the operating system. The most common one these days is a graphical file explorer, which translates clicks and double-clicks into commands to open files and run programs. Before computers had graphical displays, though, people typed instructions into a program called a **command-line shell**. Each time a command is entered, the shell runs some other programs, prints their output in human-readable form, and then displays a *prompt* to signal that it's ready to accept the next command. (Its name comes from the notion that it's the "outer shell" of the computer.)

Typing commands instead of clicking and dragging may seem clumsy at first, but as you will see, once you start spelling out what you want the computer to do, you can combine old commands to create new ones and automate repetitive operations with just a few keystrokes.

What is the relationship between the graphical file explorer that most people use and the command-line shell?

ANSWER THE QUESTION 50 XP

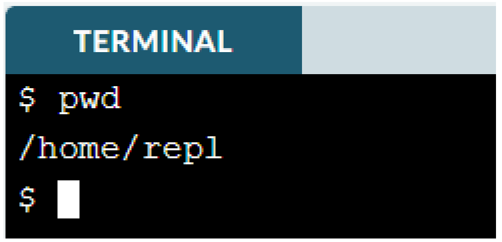
Possible Answers

- The file explorer lets you view and edit files, while the shell lets you run programs. press 1
- The file explorer is built on top of the shell. press 2
- The shell is part of the operating system, while the file explorer is separate. press 3
- [✓ They are both interfaces for issuing commands to the operating system.] press 4

Where am I?

The **filesystem** manages files and directories (or folders). Each is identified by an **absolute path** that shows how to reach it from the filesystem's **root directory**: `/home/repl` is the directory `repl` in the directory `home`, while `/home/repl/course.txt` is a file `course.txt` in that directory, and `/` on its own is the root directory.

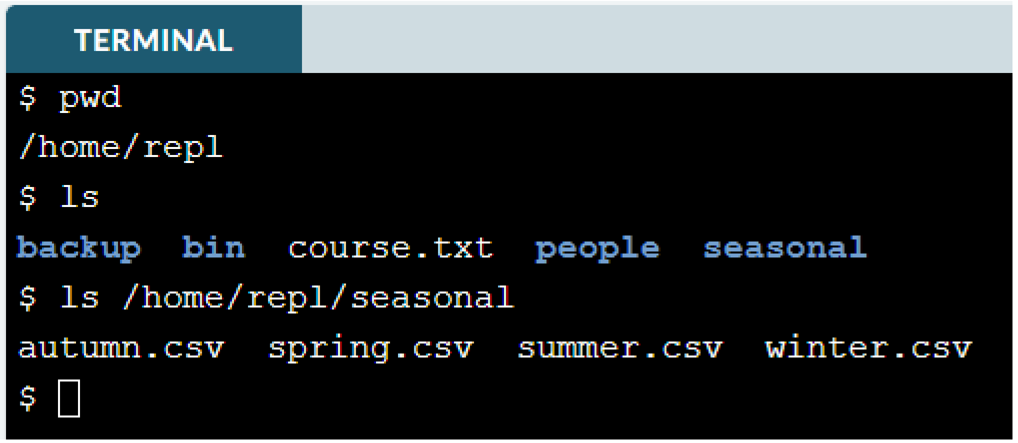
To find out where you are in the filesystem, run the command `pwd` (short for "print working directory"). This prints the absolute path of your **current working directory**, which is where the shell runs commands and looks for files by default.



```
$ pwd
/home/repl
$ 
```

How can I identify files and directories?

`pwd` tells you where you are. To find out what's there, type `ls` (which is short for "listing") and press the enter key. On its own, `ls` lists the contents of your current directory (the one displayed by `pwd`). If you add the names of some files, `ls` will list them, and if you add the names of directories, it will list their contents. For example, `ls /home/repl` shows you what's in your starting directory (usually called your **home directory**).

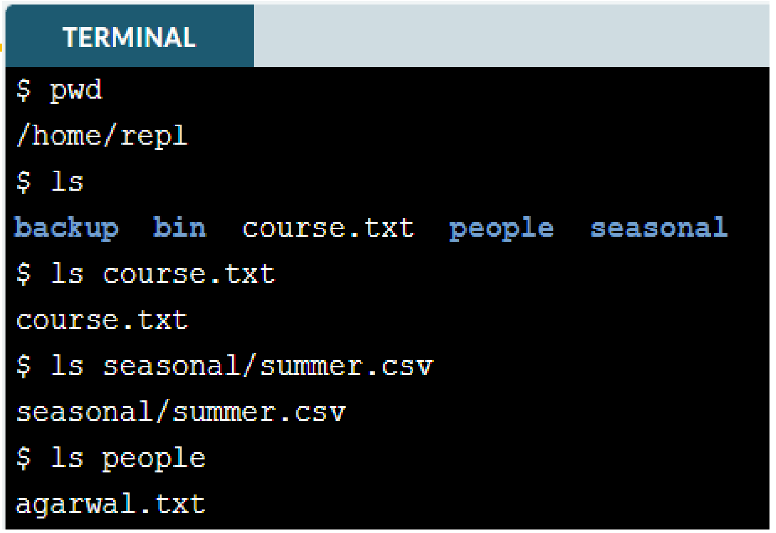


```
$ pwd
/home/repl
$ ls
backup  bin  course.txt  people  seasonal
$ ls /home/repl/seasonal
autumn.csv  spring.csv  summer.csv  winter.csv
$
```

How else can I identify files and directories?

An absolute path is like a latitude and longitude: it specifies the same thing no matter where you are. A **relative path**, on the other hand, specifies a location starting from where you are: it's like saying "20 kilometers north".

For example, if you are in the directory `/home/repl`, the relative path `seasonal` specifies the same directory as `/home/repl/seasonal`, while `seasonal/winter.csv` specifies the same file as `/home/repl/seasonal/winter.csv`. The shell decides if a path is absolute or relative by looking at its first character: if it begins with `/`, it is absolute, and if it doesn't, it is relative.

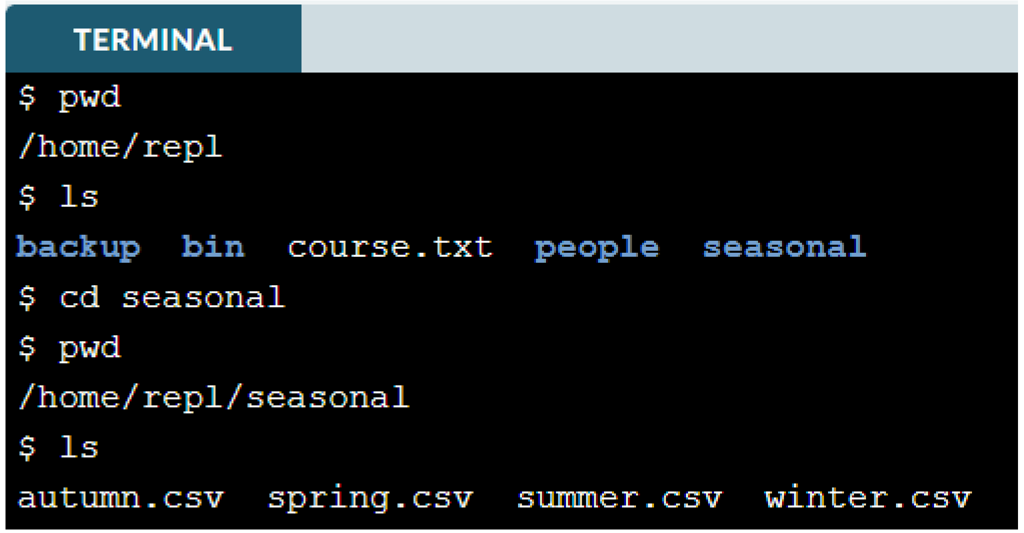

TERMINAL

```
$ pwd
/home/repl
$ ls
backup  bin  course.txt  people  seasonal
$ ls course.txt
course.txt
$ ls seasonal/summer.csv
seasonal/summer.csv
$ ls people
agarwal.txt
```

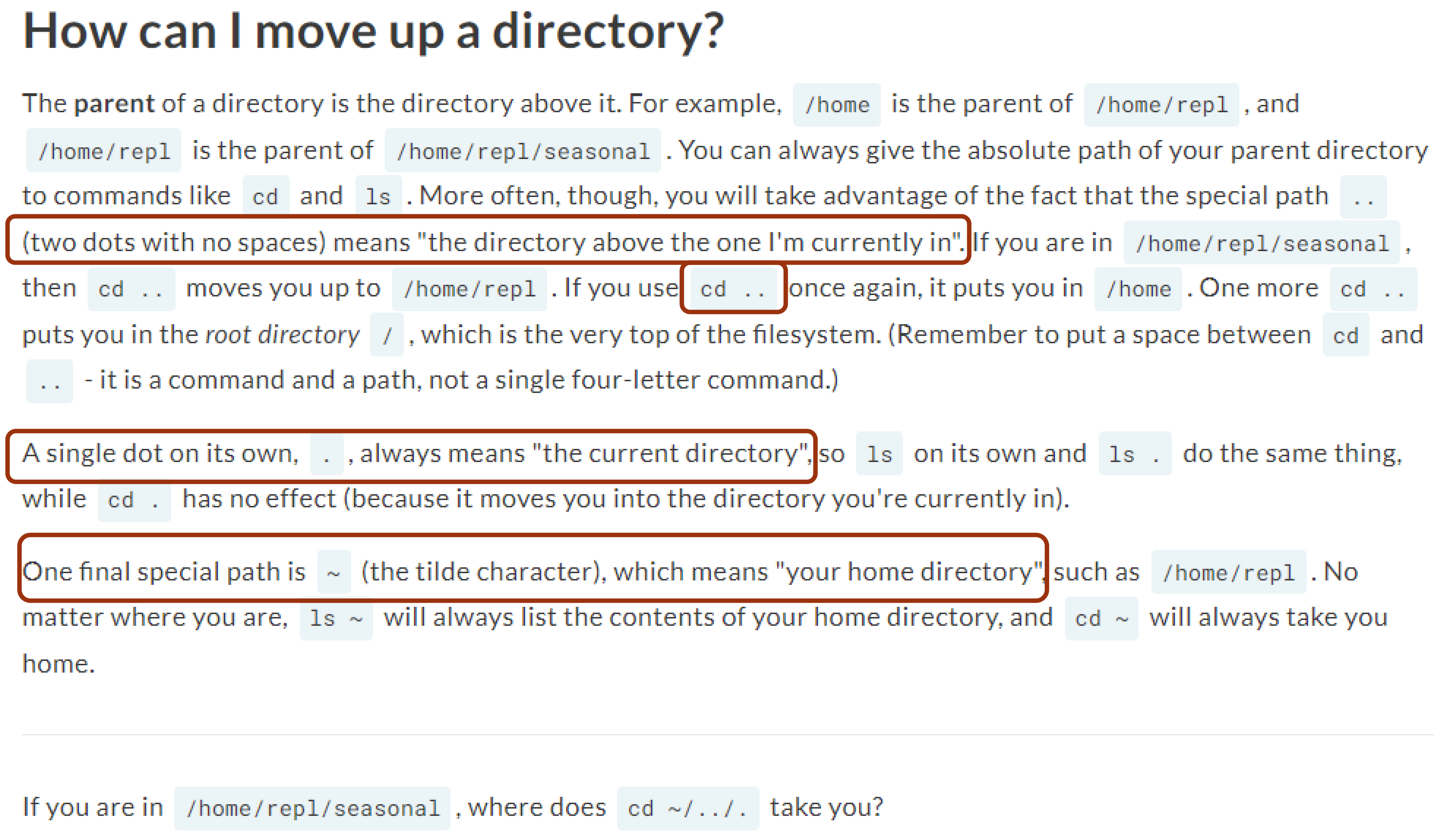
How can I move to another directory?

Just as you can move around in a file browser by double-clicking on folders, you can move around in the filesystem using the command `cd` (which stands for "change directory").

If you type `cd seasonal` and then type `pwd`, the shell will tell you that you are now in `/home/repl/seasonal`. If you then run `ls` on its own, it shows you the contents of `/home/repl/seasonal`, because that's where you are. If you want to get back to your home directory `/home/repl`, you can use the command `cd /home/repl`.



```
$ pwd
/home/repl
$ ls
backup  bin  course.txt  people  seasonal
$ cd seasonal
$ pwd
/home/repl/seasonal
$ ls
autumn.csv  spring.csv  summer.csv  winter.csv
```



How can I move up a directory?

The **parent** of a directory is the directory above it. For example, `/home` is the parent of `/home/repl`, and `/home/repl` is the parent of `/home/repl/seasonal`. You can always give the absolute path of your parent directory to commands like `cd` and `ls`. More often, though, you will take advantage of the fact that the special path `..` (two dots with no spaces) means "the directory above the one I'm currently in". If you are in `/home/repl/seasonal`, then `cd ..` moves you up to `/home/repl`. If you use `cd ..` once again, it puts you in `/home`. One more `cd ..` puts you in the *root directory* `/`, which is the very top of the filesystem. (Remember to put a space between `cd` and `..` - it is a command and a path, not a single four-letter command.)

A single dot on its own, `.`, always means "the current directory", so `ls` on its own and `ls .` do the same thing, while `cd .` has no effect (because it moves you into the directory you're currently in).

One final special path is `~` (the tilde character), which means "your home directory", such as `/home/repl`. No matter where you are, `ls ~` will always list the contents of your home directory, and `cd ~` will always take you home.

If you are in `/home/repl/seasonal`, where does `cd ~/../.` take you?

How can I copy files?

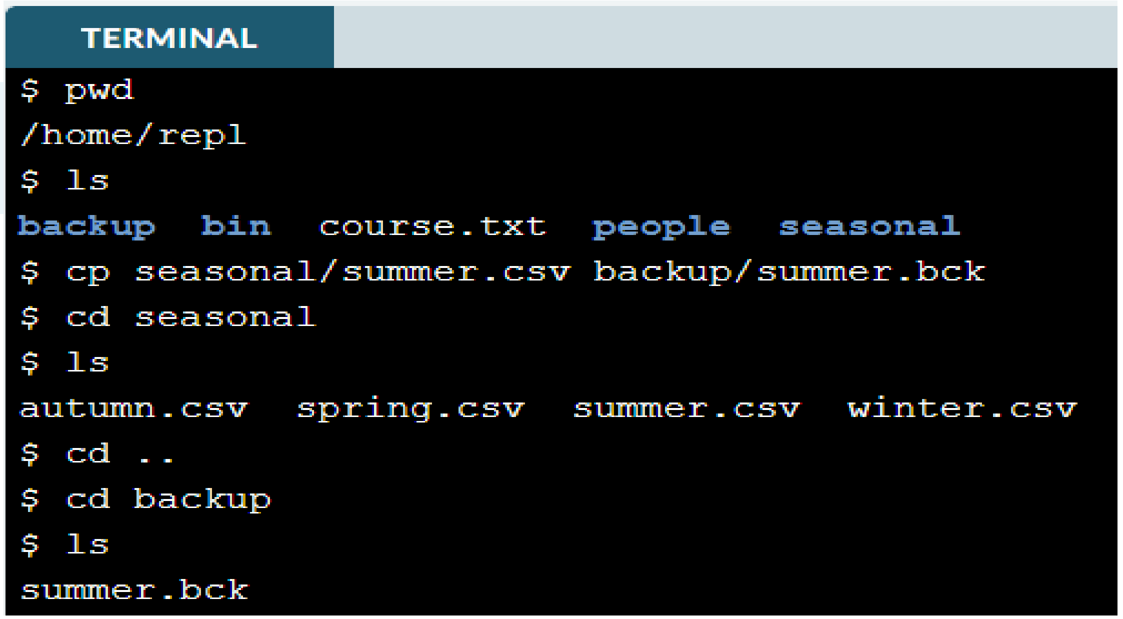
You will often want to copy files, move them into other directories to organize them, or rename them. One command to do this is `cp`, which is short for "copy". If `original.txt` is an existing file, then:

```
cp original.txt duplicate.txt
```

creates a copy of `original.txt` called `duplicate.txt`. If there already was a file called `duplicate.txt`, it is overwritten. If the last parameter to `cp` is an existing directory, then a command like:

```
cp seasonal/autumn.csv seasonal/winter.csv backup
```

copies *all* of the files into that directory.


TERMINAL

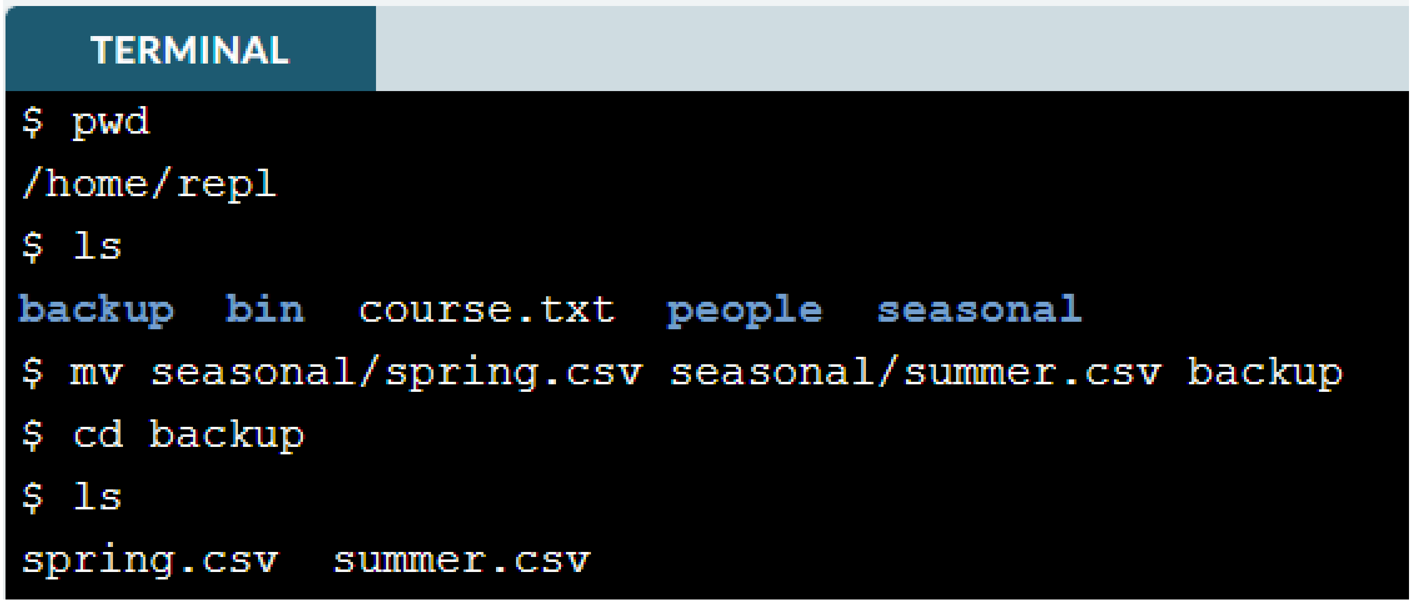
```
$ pwd
/home/repl
$ ls
backup  bin  course.txt  people  seasonal
$ cp seasonal/summer.csv backup/summer.bck
$ cd seasonal
$ ls
autumn.csv  spring.csv  summer.csv  winter.csv
$ cd ..
$ cd backup
$ ls
summer.bck
```

How can I move a file?

While `cp` copies a file, `mv` moves it from one directory to another, just as if you had dragged it in a graphical file browser. It handles its parameters the same way as `cp`, so the command:

```
mv autumn.csv winter.csv ..
```

moves the files `autumn.csv` and `winter.csv` from the current working directory up one level to its parent directory (because `..` always refers to the directory above your current location).



```
$ pwd
/home/repl
$ ls
backup  bin  course.txt  people  seasonal
$ mv seasonal/spring.csv seasonal/summer.csv backup
$ cd backup
$ ls
spring.csv  summer.csv
```

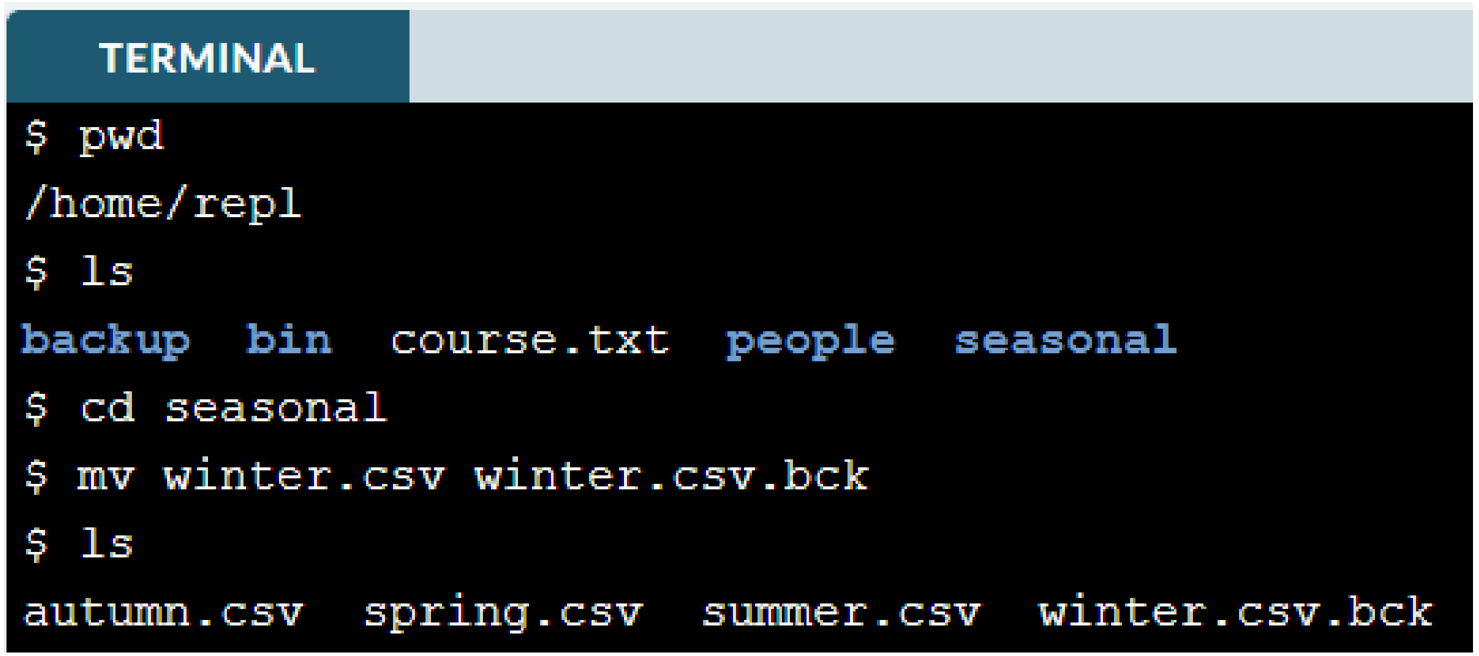
How can I rename files?

`mv` can also be used to rename files. If you run:

```
mv course.txt old-course.txt
```

then the file `course.txt` in the current working directory is "moved" to the file `old-course.txt`. This is different from the way file browsers work, but is often handy.

One warning: just like `cp`, `mv` will overwrite existing files. If, for example, you already have a file called `old-course.txt`, then the command shown above will replace it with whatever is in `course.txt`.


TERMINAL

```
$ pwd
/home/repl
$ ls
backup  bin  course.txt  people  seasonal
$ cd seasonal
$ mv winter.csv winter.csv.bck
$ ls
autumn.csv  spring.csv  summer.csv  winter.csv.bck
```

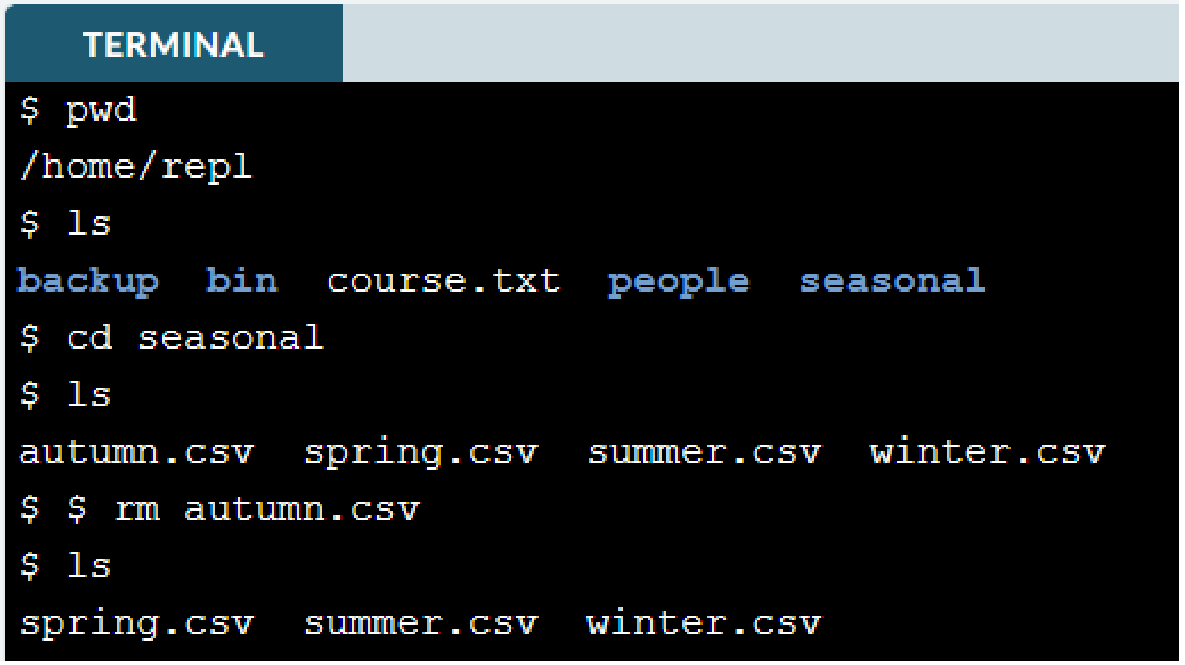
How can I delete files?

We can copy files and move them around; to delete them, we use `rm`, which stands for "remove". As with `cp` and `mv`, you can give `rm` the names of as many files as you'd like, so:

```
rm thesis.txt backup/thesis-2017-08.txt
```

removes both `thesis.txt` and `backup/thesis-2017-08.txt`

`rm` does exactly what its name says, and it does it right away: unlike graphical file browsers, the shell doesn't have a trash can, so when you type the command above, your thesis is gone for good.

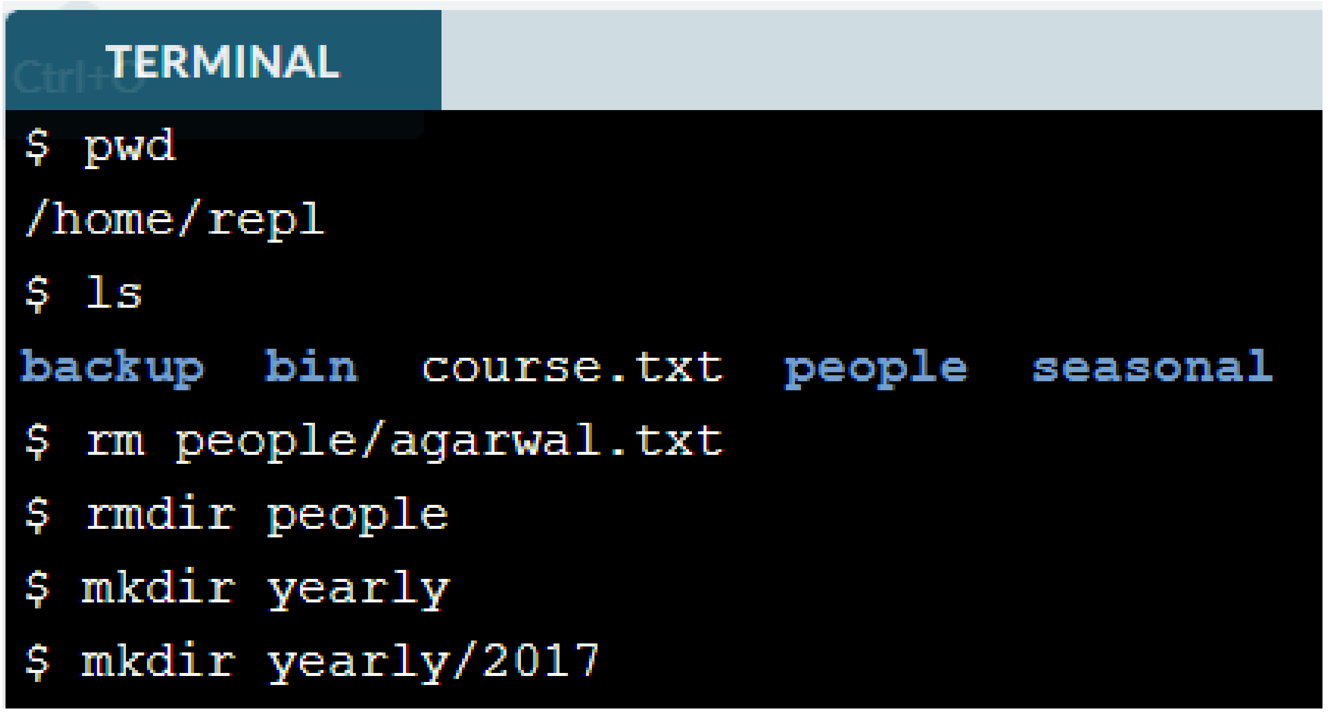


```
$ pwd
/home/repl
$ ls
backup  bin  course.txt  people  seasonal
$ cd seasonal
$ ls
autumn.csv  spring.csv  summer.csv  winter.csv
$ $ rm autumn.csv
$ ls
spring.csv  summer.csv  winter.csv
```

How can I create and delete directories?

`mv` treats directories the same way it treats files: if you are in your home directory and run `mv seasonal by-season`, for example, `mv` changes the name of the `seasonal` directory to `by-season`. However, `rm` works differently.

If you try to `rm` a directory, the shell prints an error message telling you it can't do that, primarily to stop you from accidentally deleting an entire directory full of work. Instead, you can use a separate command called `rmdir`. For added safety, it only works when the directory is empty, so you must delete the files in a directory *before* you delete the directory. (Experienced users can use the `-r` option to `rm` to get the same effect; we will discuss command options in the next chapter.)

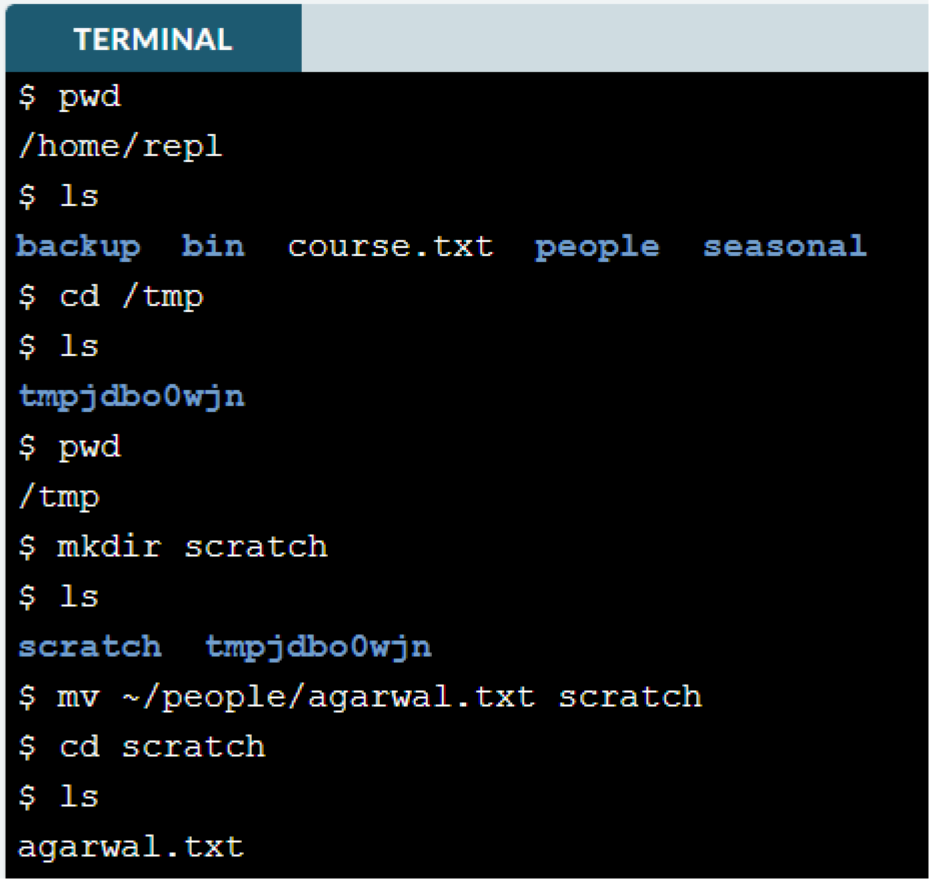


TERMINAL

```
$ pwd
/home/repl
$ ls
backup  bin  course.txt  people  seasonal
$ rm people/agarwal.txt
$ rmdir people
$ mkdir yearly
$ mkdir yearly/2017
```

Wrapping up

You will often create intermediate files when analyzing data. Rather than storing them in your home directory, you can put them in `/tmp`, which is where people and programs often keep files they only need briefly. (Note that `/tmp` is immediately below the root directory `/`, *not* below your home directory.) This wrap-up exercise will show you how to do that.



TERMINAL

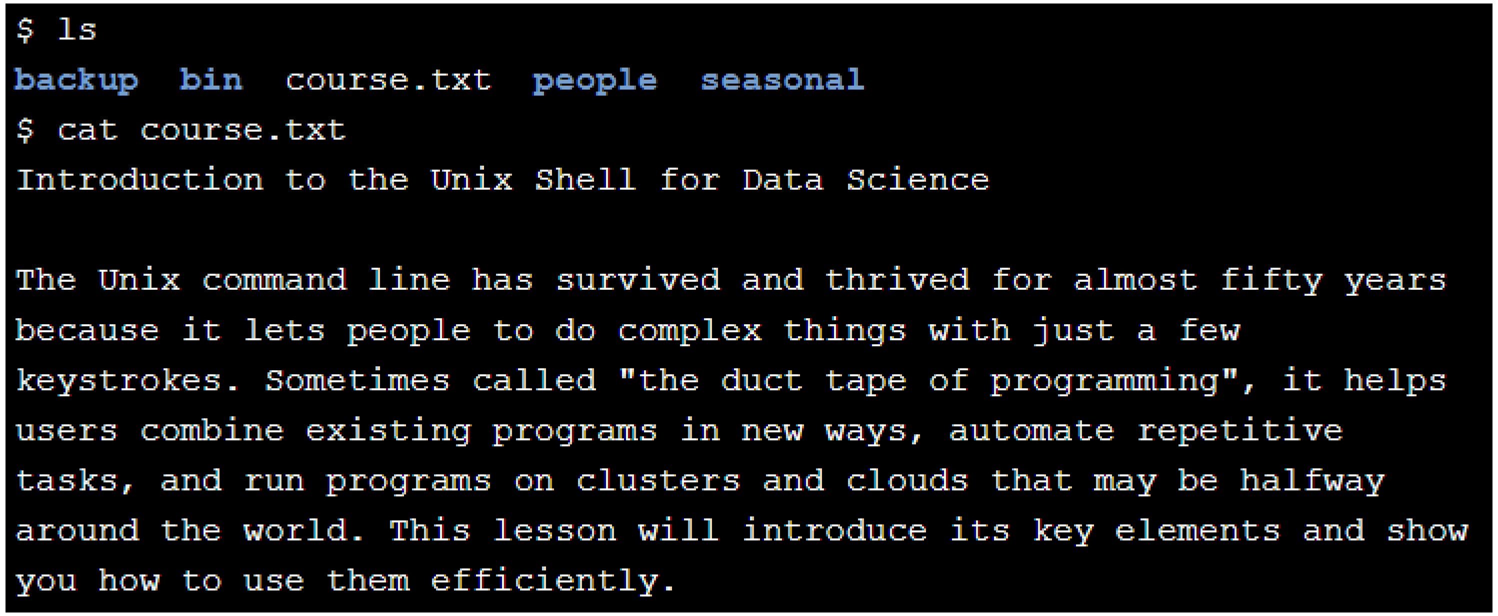
```
$ pwd
/home/repl
$ ls
backup  bin  course.txt  people  seasonal
$ cd /tmp
$ ls
tmpjdbo0wjn
$ pwd
/tmp
$ mkdir scratch
$ ls
scratch  tmpjdbo0wjn
$ mv ~/people/agarwal.txt scratch
$ cd scratch
$ ls
agarwal.txt
```

How can I view a file's contents?

Before you rename or delete files, you may want to have a look at their contents. The simplest way to do this is with `cat`, which just prints the contents of files onto the screen. (Its name is short for "concatenate", since it will print all the files whose names you give it).

```
cat agarwal.txt
```

```
name: Agarwal, Jasmine
position: RCT2
start: 2017-04-01
benefits: full
```



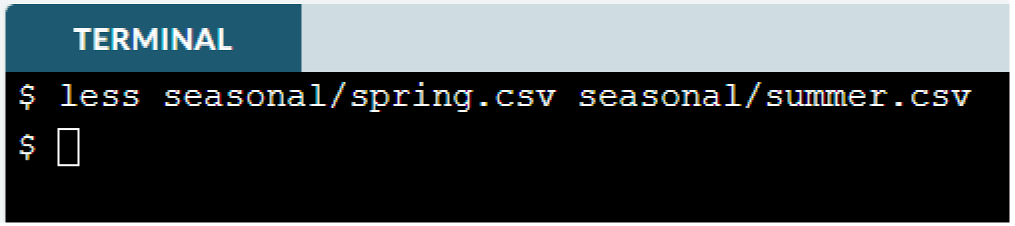
```
$ ls
backup  bin  course.txt  people  seasonal
$ cat course.txt
Introduction to the Unix Shell for Data Science

The Unix command line has survived and thrived for almost fifty years
because it lets people to do complex things with just a few
keystrokes. Sometimes called "the duct tape of programming", it helps
users combine existing programs in new ways, automate repetitive
tasks, and run programs on clusters and clouds that may be halfway
around the world. This lesson will introduce its key elements and show
you how to use them efficiently.
```

How can I view a file's contents piece by piece?

You can use `cat` to print large files and then scroll through the output, but it is usually more convenient to **page** the output. The original command for doing this was called `more`, but it has been superseded by a more powerful command called `less`. (This kind of naming is what passes for humor in the Unix world.) When you `less` a file, one page is displayed at a time; you can press spacebar to page down or type `q` to quit.

If you give `less` the names of several files, you can type `:n` (colon and a lower-case 'n') to move to the next file, `:p` to go back to the previous one, or `:q` to quit.

Note: If you view solutions to exercises that use `less`, you will see an extra command at the end that turns paging *off* so that we can test your solutions efficiently.



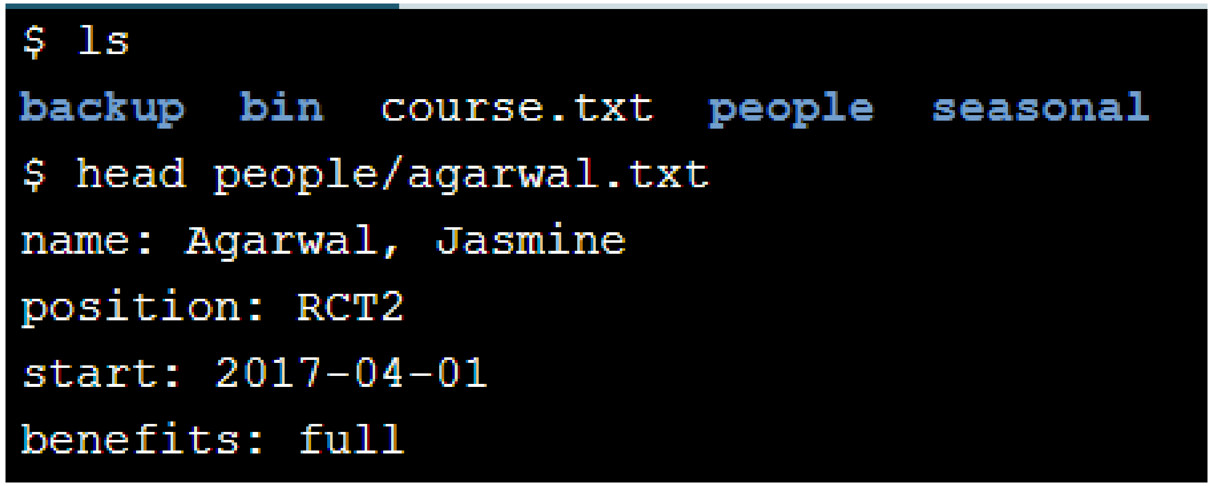
```
$ less seasonal/spring.csv seasonal/summer.csv
$ 
```

How can I look at the start of a file?

The first thing most data scientists do when given a new dataset to analyze is figure out what fields it contains and what values those fields have. If the dataset has been exported from a database or spreadsheet, it will often be stored as **comma-separated values** (CSV). A quick way to figure out what it contains is to look at the first few rows.

We can do this in the shell using a command called `head`. As its name suggests, it prints the first few lines of a file (where "a few" means 10), so the command:

```
head seasonal/summer.csv
```

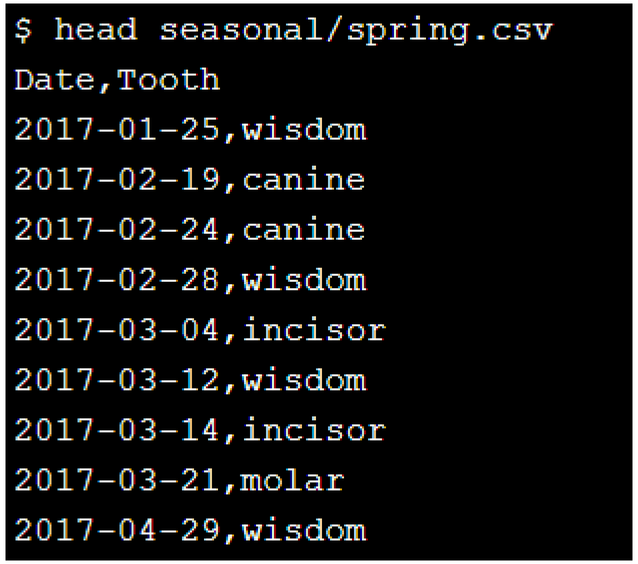


```
$ ls
backup  bin  course.txt  people  seasonal
$ head people/agarwal.txt
name: Agarwal, Jasmine
position: RCT2
start: 2017-04-01
benefits: full
```

How can I type less?

One of the shell's power tools is **tab completion.** If you start typing the name of a file and then press the tab key, the shell will do its best to auto-complete the path. For example, if you type `sea` and press tab, it will fill in the directory name `seasonal/` (with a trailing slash). If you then type `a` and tab, it will complete the path as `seasonal/autumn.csv` .

If the path is ambiguous, such as `seasonal/s` , pressing tab a second time will display a list of possibilities. Typing another character or two to make your path more specific and then pressing tab will fill in the rest of the name.



```
$ head seasonal/spring.csv
Date,Tooth
2017-01-25,wisdom
2017-02-19,canine
2017-02-24,canine
2017-02-28,wisdom
2017-03-04,incisor
2017-03-12,wisdom
2017-03-14,incisor
2017-03-21,molar
2017-04-29,wisdom
```

How can I control what commands do?

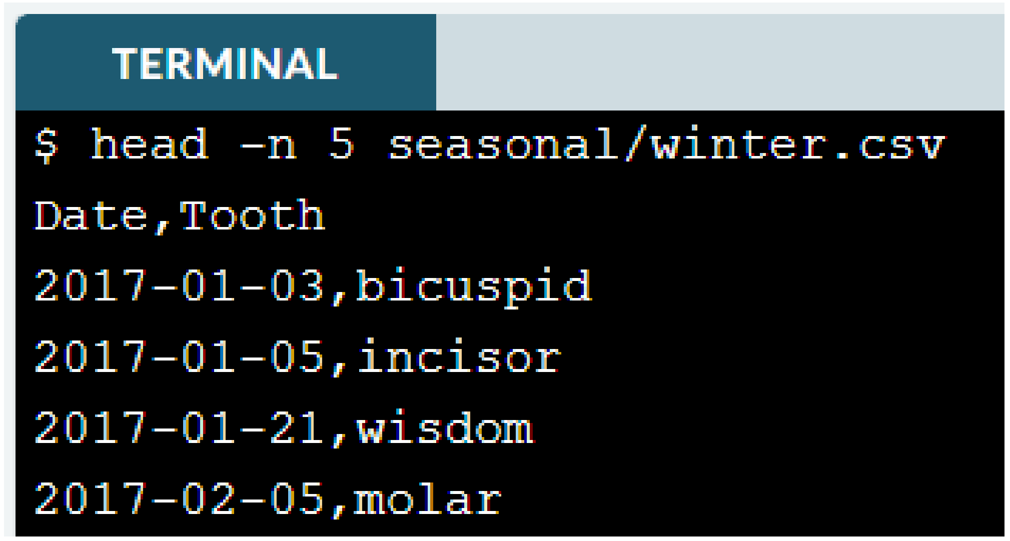
You won't always want to look at the first 10 lines of a file, so the shell lets you change `head` 's behavior by giving it a **command-line flag** (or just "flag" for short). If you run the command:

```
head -n 3 seasonal/summer.csv
```

`head` will only display the first three lines of the file. If you run `head -n 100` , it will display the first 100 (assuming there are that many), and so on.

A flag's name usually indicates its purpose (for example, `-n` is meant to signal "**n**umber of lines"). Command flags don't have to be a `-` followed by a single letter, but it's a widely-used convention.

Note: it's considered good style to put all flags *before* any filenames, so in this course, we only accept answers that do that.



```
$ head -n 5 seasonal/winter.csv
Date,Tooth
2017-01-03,bicuspid
2017-01-05,incisor
2017-01-21,wisdom
2017-02-05,molar
```

How can I list everything below a directory?

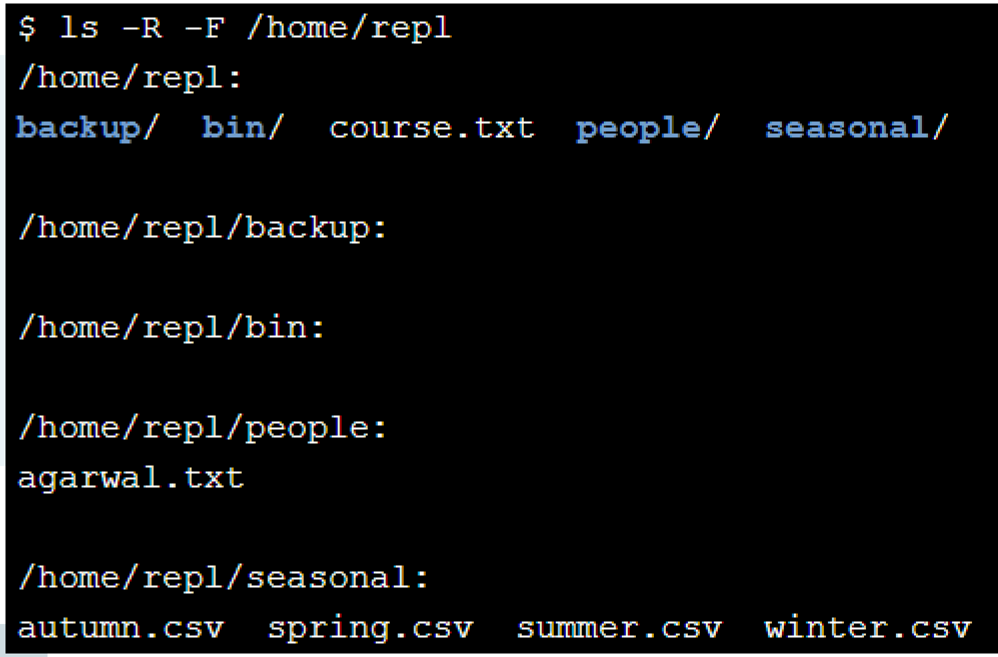
In order to see everything underneath a directory, no matter how deeply nested it is, you can give `ls` the flag `-R` (which means "recursive"). If you use `ls -R` in your home directory, you will see something like this:

```
backup          course.txt      people          seasonal

./backup:

./people:
agarwal.txt

./seasonal:
autumn.csv      spring.csv      summer.csv      winter.csv
```

This shows every file and directory in the current level, then everything in each sub-directory, and so on.

INSTRUCTIONS 100 XP

To help you know what is what, `ls` has another flag `-F` that prints a `/` after the name of every directory and a `*` after the name of every runnable program. Run `ls` with the two flags, `-R` and `-F`, and the absolute path to your home directory to see everything it contains. (The order of the flags doesn't matter, but the directory name must come last.)



```
$ ls -R -F /home/repl
/home/repl:
backup/  bin/  course.txt  people/  seasonal/

/home/repl/backup:

/home/repl/bin:

/home/repl/people:
agarwal.txt

/home/repl/seasonal:
autumn.csv  spring.csv  summer.csv  winter.csv
```

How can I get help for a command?

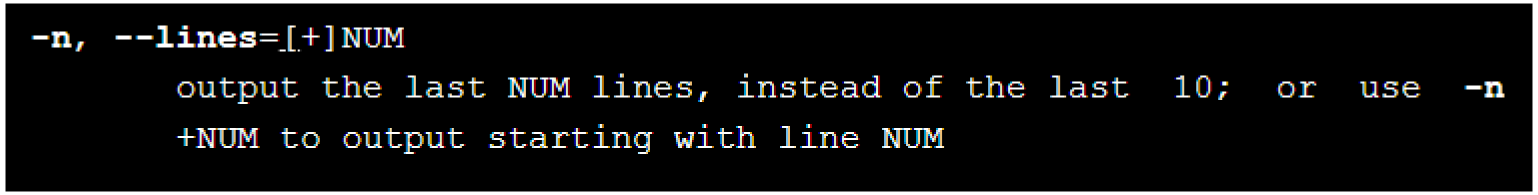
To find out what commands do, people used to use the `man` command (short for "manual"). For example, the command `man head` brings up this information:

```
HEAD(1)                BSD General Commands Manual               HEAD(1)

NAME
     head -- display first lines of a file

SYNOPSIS
     head [-n count | -c bytes] [file ...]

DESCRIPTION
     This filter displays the first count lines or bytes of each of
     the specified files, or of the standard input if no files are
     specified.  If count is omitted it defaults to 10.

     If more than a single file is specified, each file is preceded by
     a header consisting of the string ``==> XXX <=='' where ``XXX''
     is the name of the file.

SEE ALSO
     tail(1)
```

`man` automatically invokes `less`, so you may need to press spacebar to page through the information and `:q` to quit.

The one-line description under `NAME` tells you briefly what the command does, and the summary under `SYNOPSIS` lists all the flags it understands. Anything that is optional is shown in square brackets `[...]`, either/or alternatives are separated by `|`, and things that can be repeated are shown by `...`, so `head`'s manual page is telling you that you can *either* give a line count with `-n` or a byte count with `-c`, and that you can give it any number of filenames.

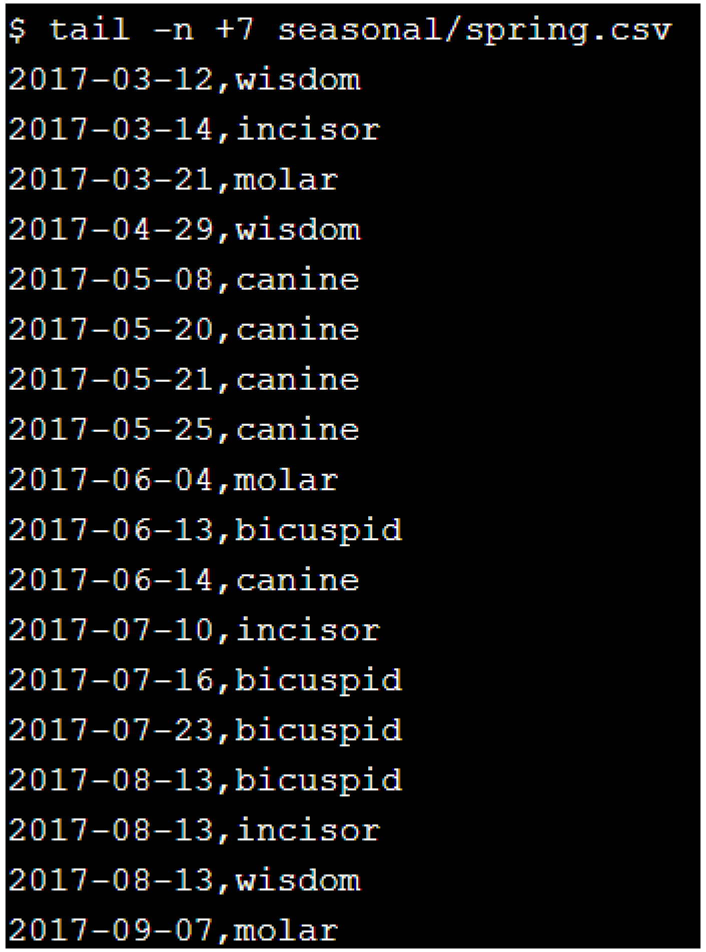
The problem with the Unix manual is that you have to know what you're looking for. If you don't, you can search Stack Overflow, ask a question on DataCamp's Slack channels, or look at the `SEE ALSO` sections of the commands you already know.



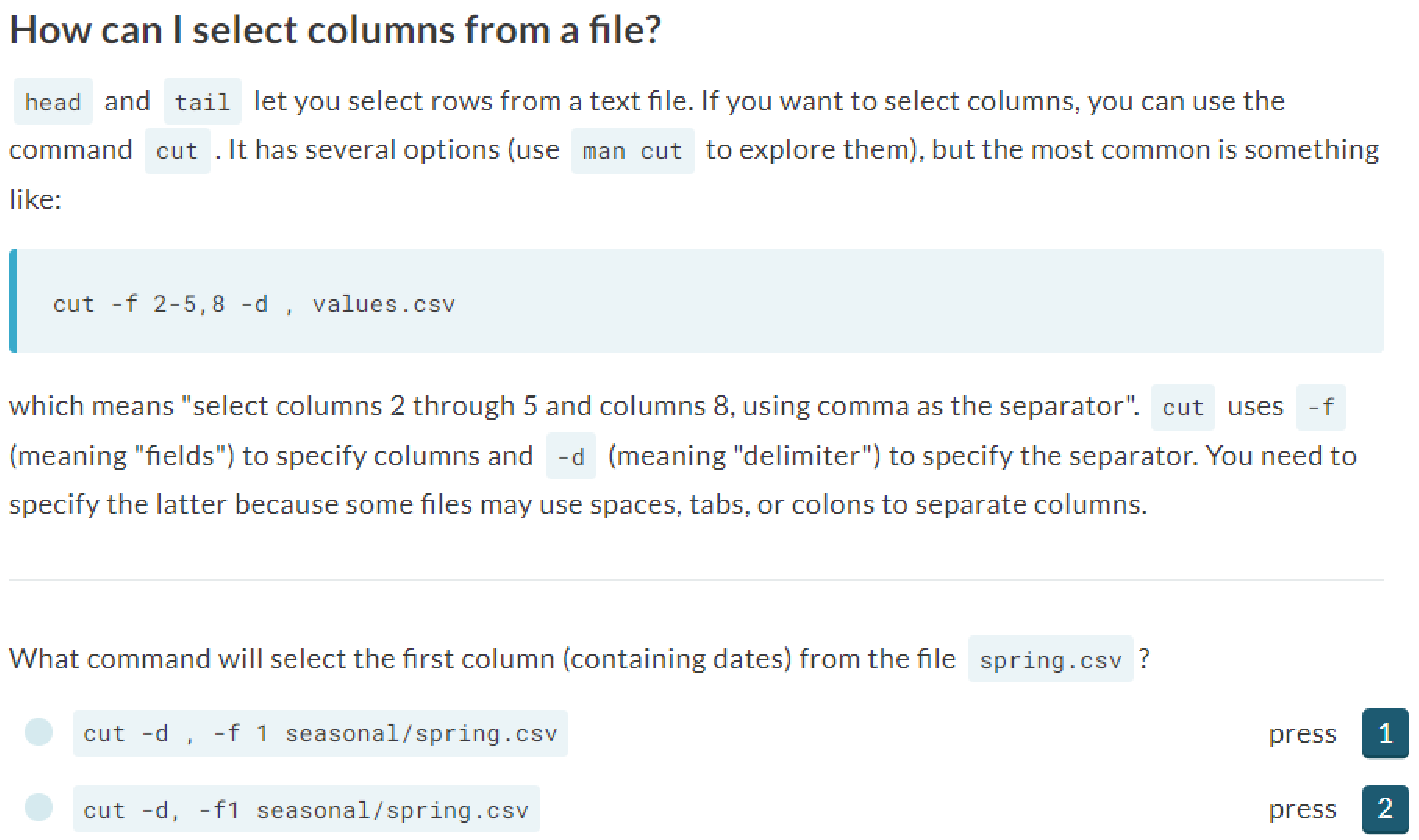
```
-n, --lines=[+]NUM
       output the last NUM lines, instead of the last  10;  or  use  -n
       +NUM to output starting with line NUM
```

✓ Read the manual page for the `tail` command to find out what putting a `+` sign in front of the number used with the `-n` flag does. (Remember to press spacebar to page down and/or type `q` to quit.)

✓ Use `tail` with `-n` and a number with a leading `+` to display all *but* the first six lines of `seasonal/spring.csv`.



```
$ tail -n +7 seasonal/spring.csv
2017-03-12,wisdom
2017-03-14,incisor
2017-03-21,molar
2017-04-29,wisdom
2017-05-08,canine
2017-05-20,canine
2017-05-21,canine
2017-05-25,canine
2017-06-04,molar
2017-06-13,bicuspid
2017-06-14,canine
2017-07-10,incisor
2017-07-16,bicuspid
2017-07-23,bicuspid
2017-08-13,bicuspid
2017-08-13,incisor
2017-08-13,wisdom
2017-09-07,molar
```



How can I select columns from a file?

`head` and `tail` let you select rows from a text file. If you want to select columns, you can use the command `cut`. It has several options (use `man cut` to explore them), but the most common is something like:

```
cut -f 2-5,8 -d , values.csv
```

which means "select columns 2 through 5 and columns 8, using comma as the separator". `cut` uses `-f` (meaning "fields") to specify columns and `-d` (meaning "delimiter") to specify the separator. You need to specify the latter because some files may use spaces, tabs, or colons to separate columns.

What command will select the first column (containing dates) from the file `spring.csv`?

- `cut -d , -f 1 seasonal/spring.csv` press 1
- `cut -d, -f1 seasonal/spring.csv` press 2

Both is correct, order doesn't matter

What can't cut do?

`cut` is a simple-minded command. In particular, it doesn't understand quoted strings. If, for example, your file is:

```
Name,Age
"Johel,Ranjit",28
"Sharma,Rupinder",26
```

then:

```
cut -f 2 -d , everyone.csv
```

will produce:

```
Age
Ranjit"
Rupinder"
```

rather than everyone's age, because it will think the comma between last and first names is a column separator.

What is the output of `cut -d : -f 2-4` on the line:

[✓ `second:third:`]

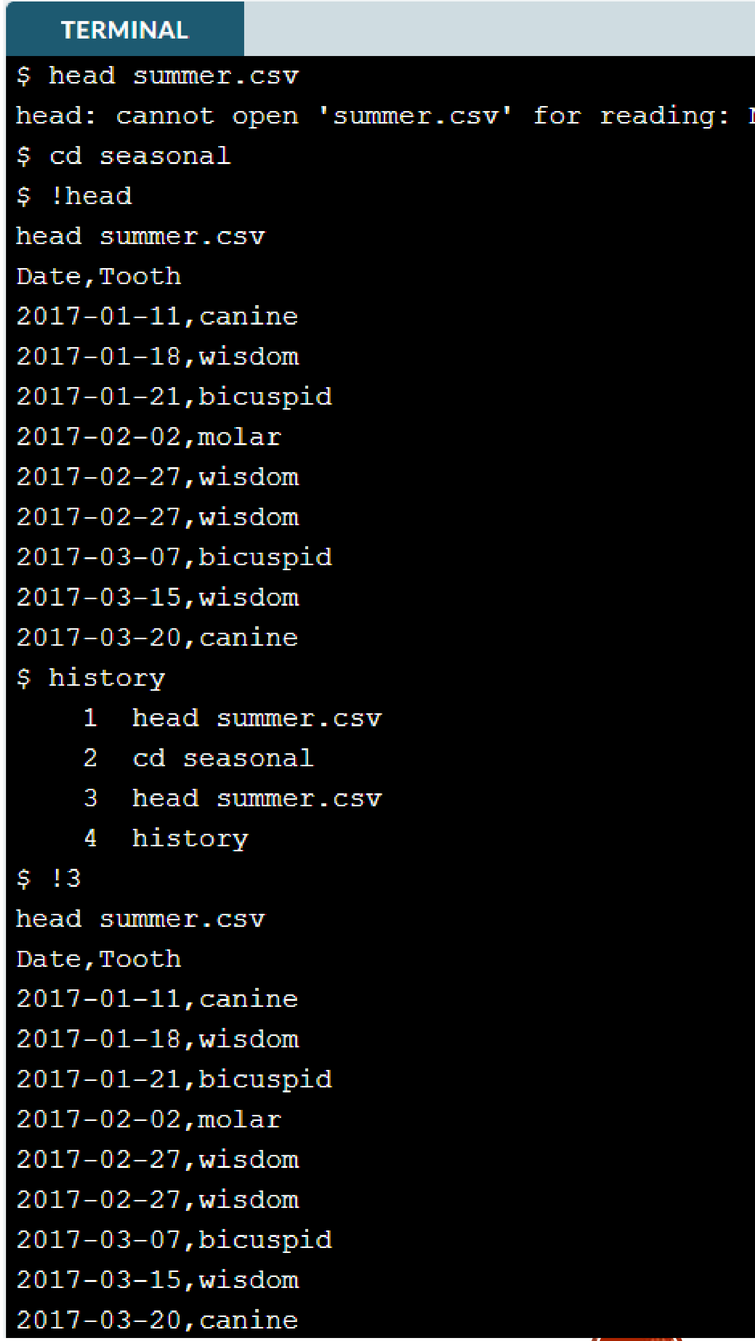
```
first:second:third:
```

(Note the trailing colon.)

How can I repeat commands?

One of the biggest advantages of using the shell is that it makes it easy for you to do things over again. If you run some commands, you can then press the up-arrow key to cycle back through them. You can also use the left and right arrow keys and the delete key to edit them. Pressing return will then run the modified command.

Even better, `history` will print a list of commands you have run recently. Each one is preceded by a serial number to make it easy to re-run particular commands: just type `!55` to re-run the 55th command in your history (if you have that many). You can also re-run a command by typing an exclamation mark followed by the command's name, such as `!head` or `!cut`, which will re-run the most recent use of that command.



```
$ head summer.csv
head: cannot open 'summer.csv' for reading:
$ cd seasonal
$ !head
head summer.csv
Date,Tooth
2017-01-11,canine
2017-01-18,wisdom
2017-01-21,bicuspid
2017-02-02,molar
2017-02-27,wisdom
2017-02-27,wisdom
2017-03-07,bicuspid
2017-03-15,wisdom
2017-03-20,canine
$ history
    1  head summer.csv
    2  cd seasonal
    3  head summer.csv
    4  history
$ !3
head summer.csv
Date,Tooth
2017-01-11,canine
2017-01-18,wisdom
2017-01-21,bicuspid
2017-02-02,molar
2017-02-27,wisdom
2017-02-27,wisdom
2017-03-07,bicuspid
2017-03-15,wisdom
2017-03-20,canine
```

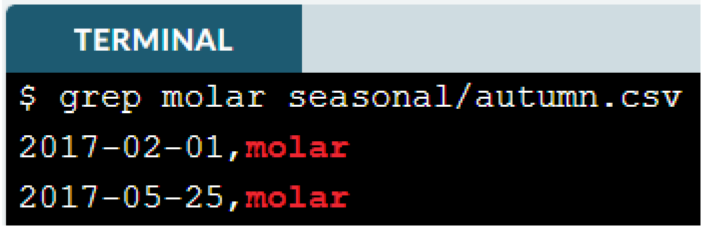
How can I select lines containing specific values?

`head` and `tail` select rows, `cut` selects columns, and `grep` selects lines according to what they contain. In its simplest form, `grep` takes a piece of text followed by one or more filenames and prints all of the lines in those files that contain that text. For example, `grep bicuspid seasonal/winter.csv` prints lines from `winter.csv` that contain "bicuspid".

`grep` can search for patterns as well; we will explore those in the next course. What's more important right now is some of `grep`'s more common flags:

- `-c` : print a count of matching lines rather than the lines themselves
- `-h` : do *not* print the names of files when searching multiple files
- `-i` : ignore case (e.g., treat "Regression" and "regression" as matches)
- `-l` : print the names of files that contain matches, not the matches
- `-n` : print line numbers for matching lines
- `-v` : invert the match, i.e., only show lines that *don't* match

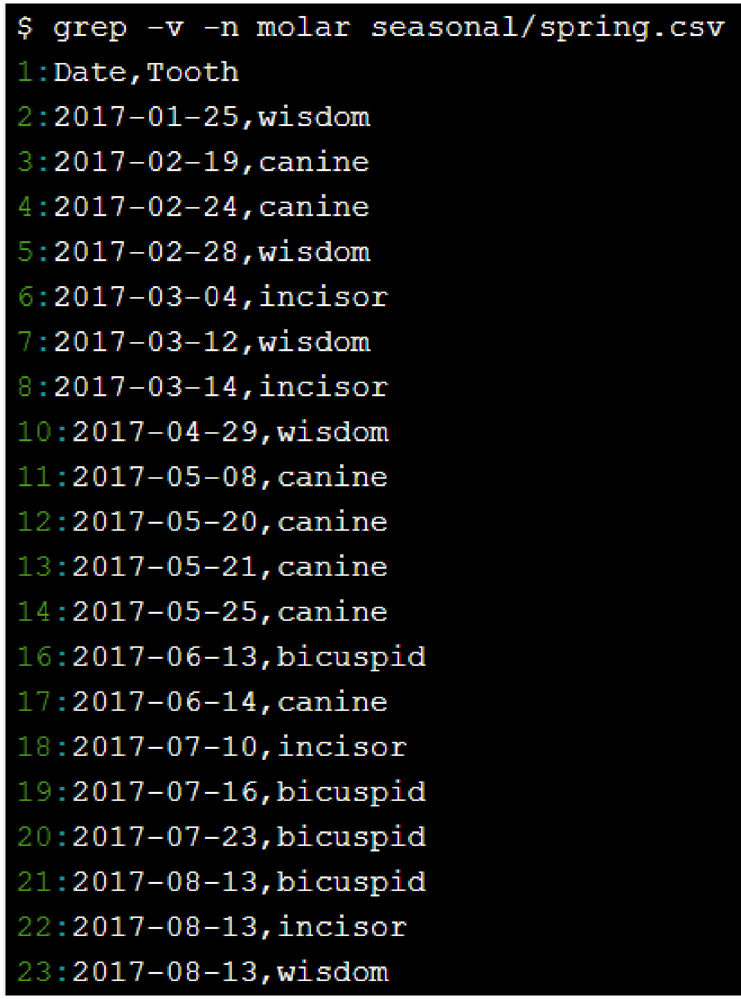
✓ Find all of the lines containing the word `molar` in `seasonal/autumn.csv` by running a single command while in your home directory. Again, it's considered good practice to put flags and arguments before filenames, so this course only accepts solutions that do that.



TERMINAL

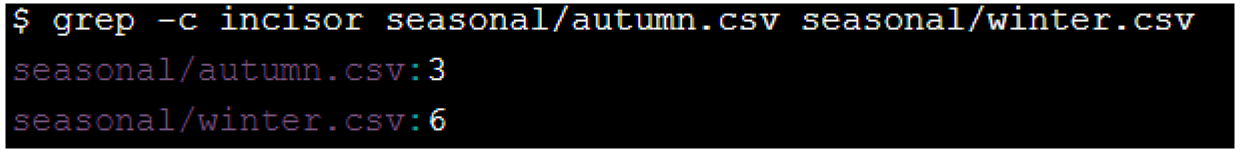
```
$ grep molar seasonal/autumn.csv
2017-02-01,molar
2017-05-25,molar
```

2 Find all of the lines that *don't* contain the word `molar` in `seasonal/spring.csv`, and show their line numbers. Remember, it's considered good style to put all of the flags *before* other values like filenames or the search term "molar", so in this course, we only accept answers that do that.



```
$ grep -v -n molar seasonal/spring.csv
1:Date,Tooth
2:2017-01-25,wisdom
3:2017-02-19,canine
4:2017-02-24,canine
5:2017-02-28,wisdom
6:2017-03-04,incisor
7:2017-03-12,wisdom
8:2017-03-14,incisor
10:2017-04-29,wisdom
11:2017-05-08,canine
12:2017-05-20,canine
13:2017-05-21,canine
14:2017-05-25,canine
16:2017-06-13,bicuspid
17:2017-06-14,canine
18:2017-07-10,incisor
19:2017-07-16,bicuspid
20:2017-07-23,bicuspid
21:2017-08-13,bicuspid
22:2017-08-13,incisor
23:2017-08-13,wisdom
```

✓ Count how many lines contain the word `incisor` in `autumn.csv` and `winter.csv` combined. (Again, run a single command from your home directory.)



```
$ grep -c incisor seasonal/autumn.csv seasonal/winter.csv
seasonal/autumn.csv:3
seasonal/winter.csv:6
```

How can I select lines containing specific values?

`head` and `tail` select rows, `cut` selects columns, and `grep` selects lines according to what they contain. In its simplest form, `grep` takes a piece of text followed by one or more filenames and prints all of the lines in those files that contain that text. For example, `grep bicuspid seasonal/winter.csv` prints lines from `winter.csv` that contain "bicuspid".

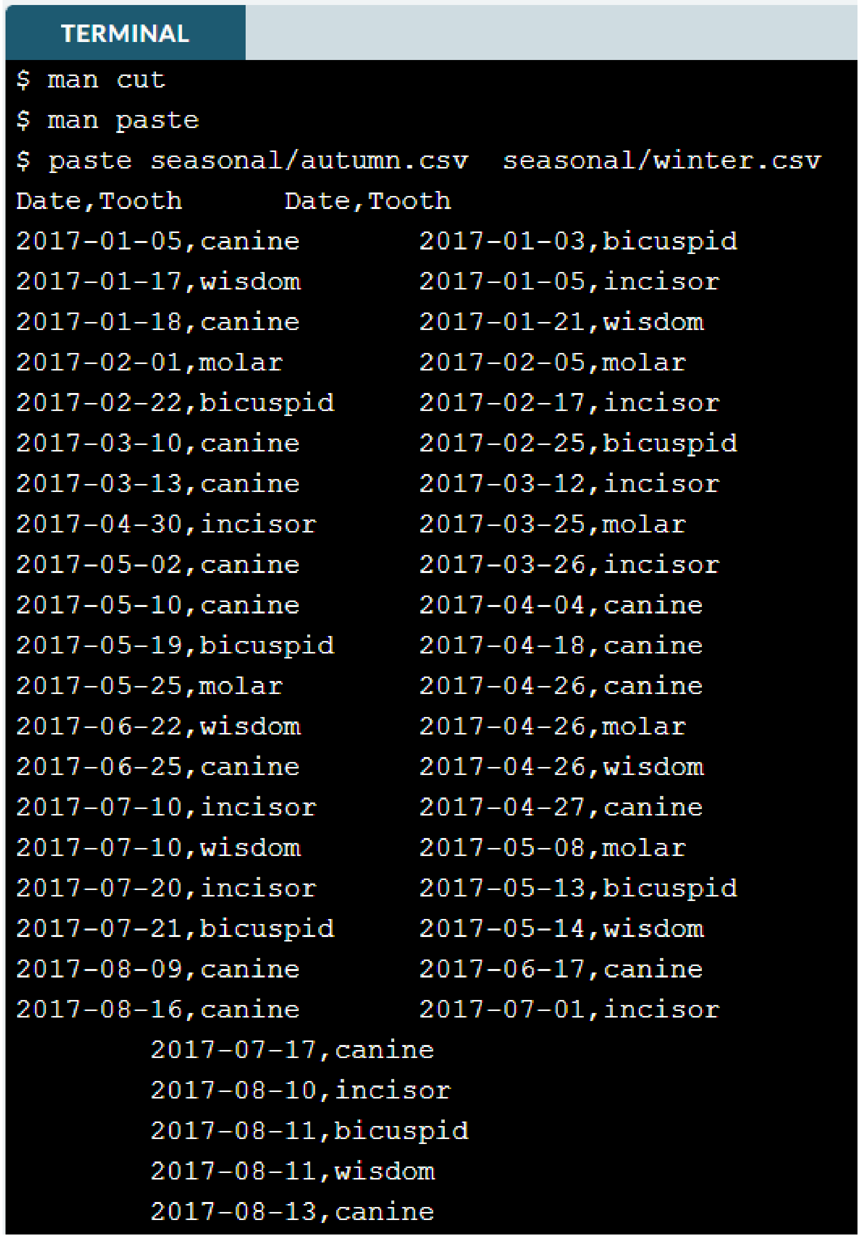
`grep` can search for patterns as well; we will explore those in the next course. What's more important right now is some of `grep`'s more common flags:

- `-c` : print a count of matching lines rather than the lines themselves
- `-h` : do *not* print the names of files when searching multiple files
- `-i` : ignore case (e.g., treat "Regression" and "regression" as matches)
- `-l` : print the names of files that contain matches, not the matches
- `-n` : print line numbers for matching lines
- `-v` : invert the match, i.e., only show lines that *don't* match

Why isn't it always safe to treat data as text?

The `SEE ALSO` section of the manual page for `cut` refers to a command called `paste` that can be used to combine data files instead of cutting them up.

Read the manual page for `paste`, and then run `paste` to combine the autumn and winter data files in a single table using a comma as a separator. What's wrong with the output from a data analysis point of view?



```
$ man cut
$ man paste
$ paste seasonal/autumn.csv  seasonal/winter.csv
Date,Tooth      Date,Tooth
2017-01-05,canine       2017-01-03,bicuspid
2017-01-17,wisdom       2017-01-05,incisor
2017-01-18,canine       2017-01-21,wisdom
2017-02-01,molar        2017-02-05,molar
2017-02-22,bicuspid     2017-02-17,incisor
2017-03-10,canine       2017-02-25,bicuspid
2017-03-13,canine       2017-03-12,incisor
2017-04-30,incisor      2017-03-25,molar
2017-05-02,canine       2017-03-26,incisor
2017-05-10,canine       2017-04-04,canine
2017-05-19,bicuspid     2017-04-18,canine
2017-05-25,molar        2017-04-26,canine
2017-06-22,wisdom       2017-04-26,molar
2017-06-25,canine       2017-04-26,wisdom
2017-07-10,incisor      2017-04-27,canine
2017-07-10,wisdom       2017-05-08,molar
2017-07-20,incisor      2017-05-13,bicuspid
2017-07-21,bicuspid     2017-05-14,wisdom
2017-08-09,canine       2017-06-17,canine
2017-08-16,canine       2017-07-01,incisor
        2017-07-17,canine
        2017-08-10,incisor
        2017-08-11,bicuspid
        2017-08-11,wisdom
        2017-08-13,canine
```

COMBINING TOOLS

- The real power of the Unix shell lies not in the individual commands, but in how easily they can be combined to do new things. This chapter will show you how to use this power to select the data you want, and introduce commands for sorting values and removing duplicates.

How can I store a command's output in a file?

All of the tools you have seen so far let you name input files. Most don't have an option for naming an output file because they don't need one. Instead, you can use **redirection** to save any command's output anywhere you want. If you run this command:

```
head -n 5 seasonal/summer.csv
```

it prints the first 5 lines of the summer data on the screen. If you run this command instead:

```
head -n 5 seasonal/summer.csv > top.csv
```

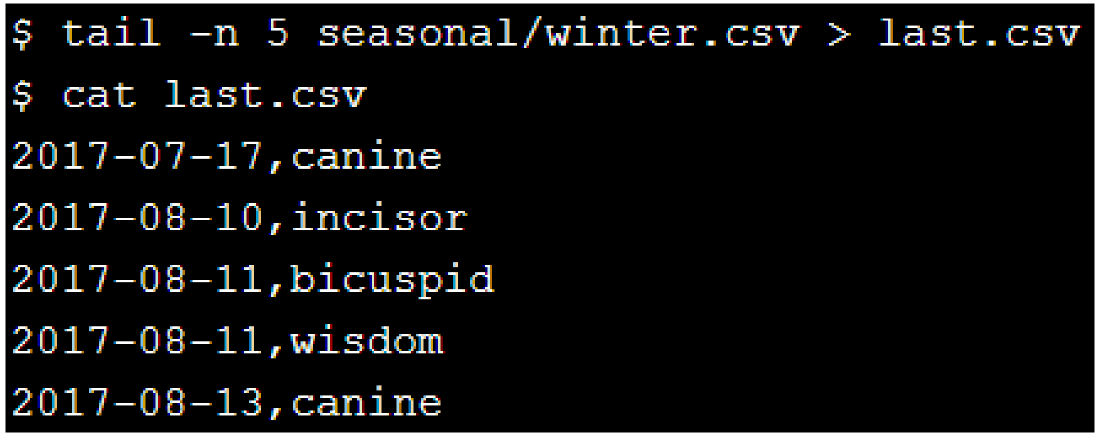
nothing appears on the screen. Instead, `head`'s output is put in a new file called `top.csv`. You can take a look at that file's contents using `cat`:

```
cat top.csv
```

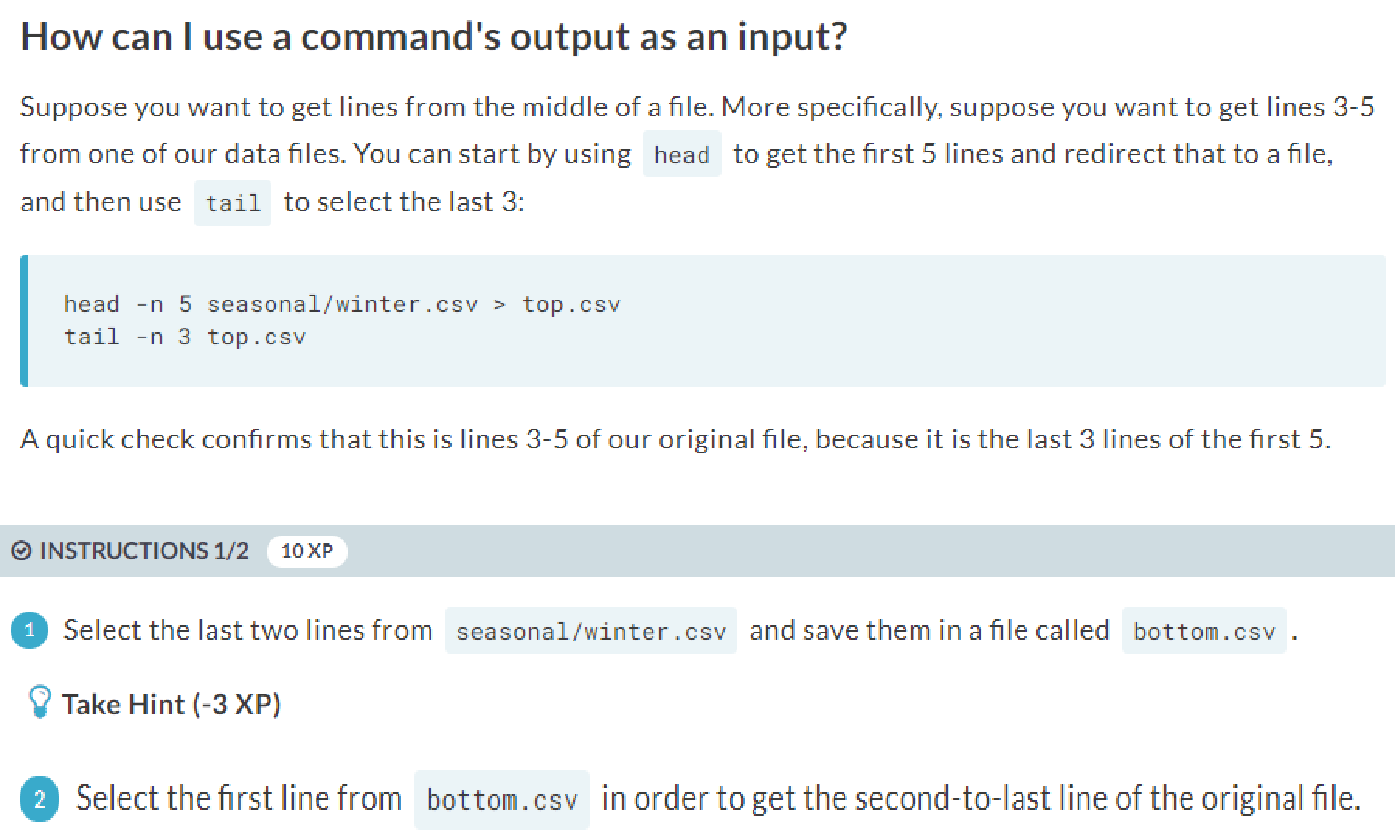
The greater-than sign `>` tells the shell to redirect `head`'s output to a file. It isn't part of the `head` command; instead, it works with every shell command that produces output.

INSTRUCTIONS 100 XP

Save the last 5 lines of `seasonal/winter.csv` in a file called `last.csv`. (Use `tail` to get the last 5 lines.)



```
$ tail -n 5 seasonal/winter.csv > last.csv
$ cat last.csv
2017-07-17,canine
2017-08-10,incisor
2017-08-11,bicuspid
2017-08-11,wisdom
2017-08-13,canine
```



How can I use a command's output as an input?

Suppose you want to get lines from the middle of a file. More specifically, suppose you want to get lines 3-5 from one of our data files. You can start by using `head` to get the first 5 lines and redirect that to a file, and then use `tail` to select the last 3:

```
head -n 5 seasonal/winter.csv > top.csv
tail -n 3 top.csv
```

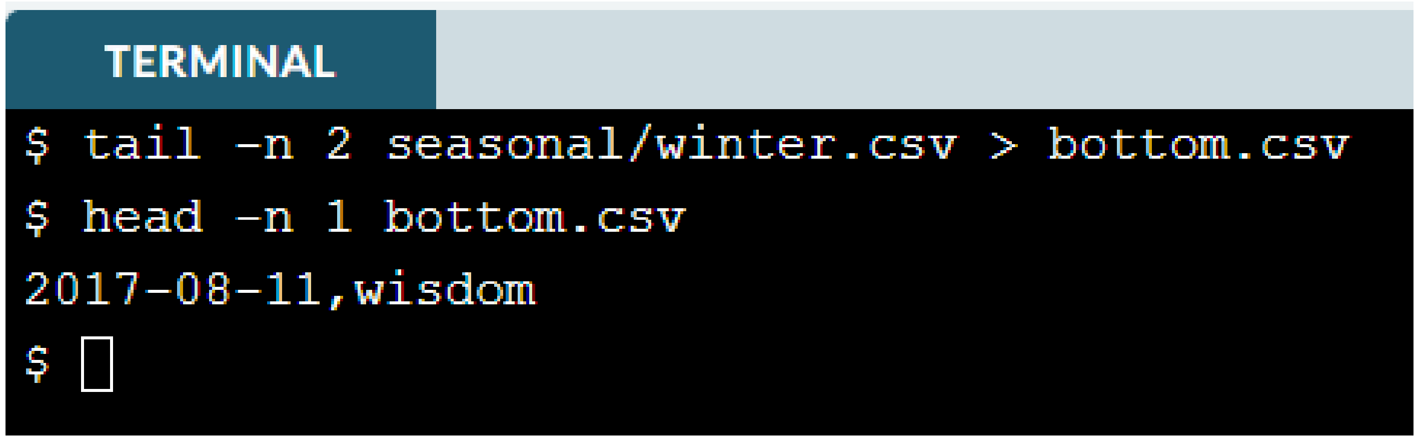
A quick check confirms that this is lines 3-5 of our original file, because it is the last 3 lines of the first 5.

INSTRUCTIONS 1/2 10 XP

1. Select the last two lines from `seasonal/winter.csv` and save them in a file called `bottom.csv`.

Take Hint (-3 XP)

2. Select the first line from `bottom.csv` in order to get the second-to-last line of the original file.



TERMINAL

```
$ tail -n 2 seasonal/winter.csv > bottom.csv
$ head -n 1 bottom.csv
2017-08-11,wisdom
$ 
```

What's a better way to combine commands?

Using redirection to combine commands has two drawbacks:

1. It leaves a lot of intermediate files lying around (like `top.csv`).
2. The commands to produce your final result are scattered across several lines of history.

The shell provides another tool that solves both of these problems at once called a **pipe**. Once again, start by running `head`:

```
head -n 5 seasonal/summer.csv
```

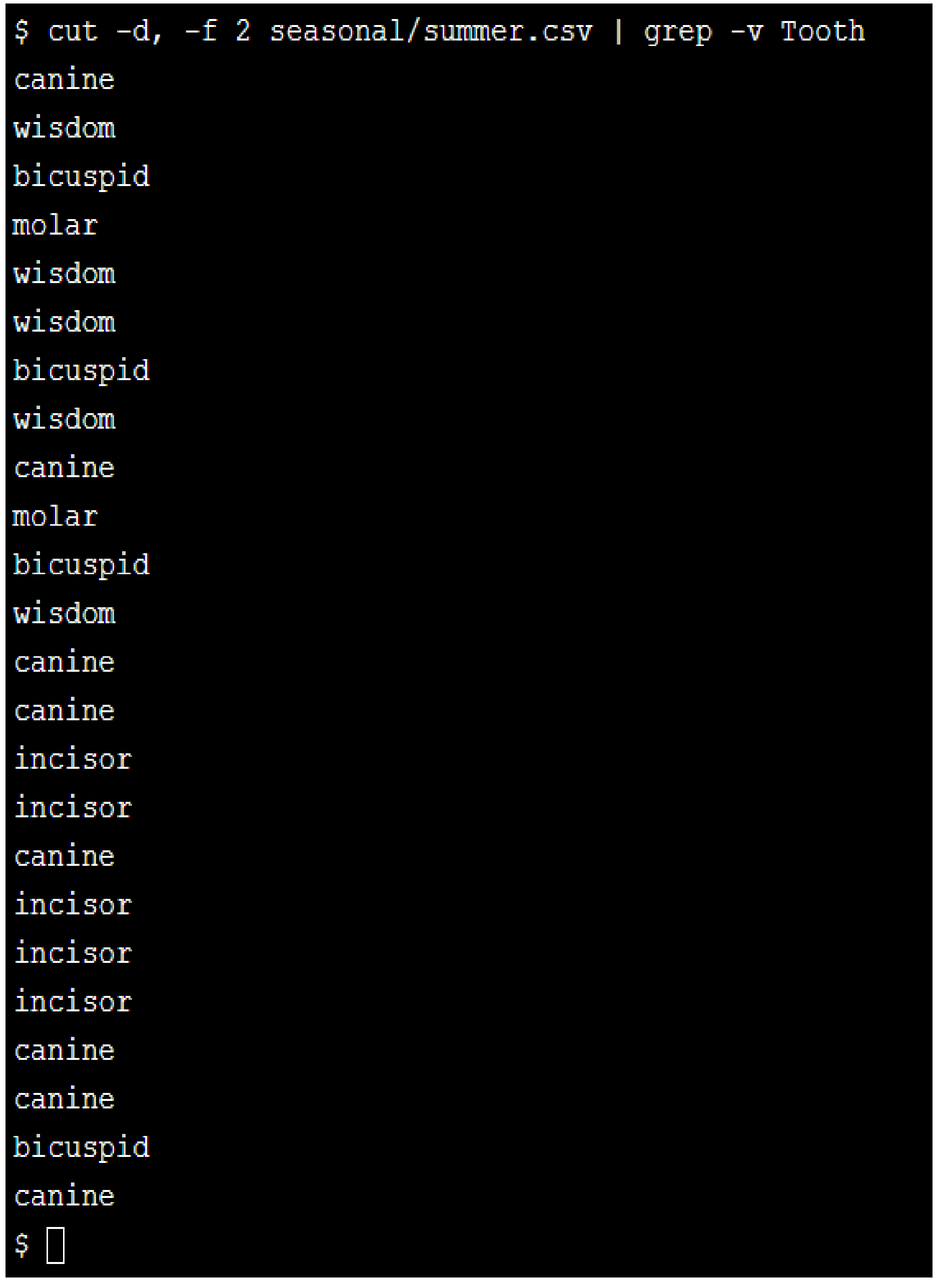
Instead of sending `head`'s output to a file, add a vertical bar and the `tail` command *without* a filename:

```
head -n 5 seasonal/summer.csv | tail -n 3
```

The pipe symbol tells the shell to use the output of the command on the left as the input to the command on the right.

INSTRUCTIONS 100 XP

Write a pipeline that uses `cut` to select all of the tooth names from column 2 of `seasonal/summer.csv` and then `grep -v` to exclude the header line containing the word "Tooth".



```
$ cut -d, -f 2 seasonal/summer.csv | grep -v Tooth
canine
wisdom
bicuspid
molar
wisdom
wisdom
bicuspid
wisdom
canine
molar
bicuspid
wisdom
canine
canine
incisor
incisor
canine
incisor
incisor
incisor
canine
canine
bicuspid
canine
$
```

How can I combine many commands?

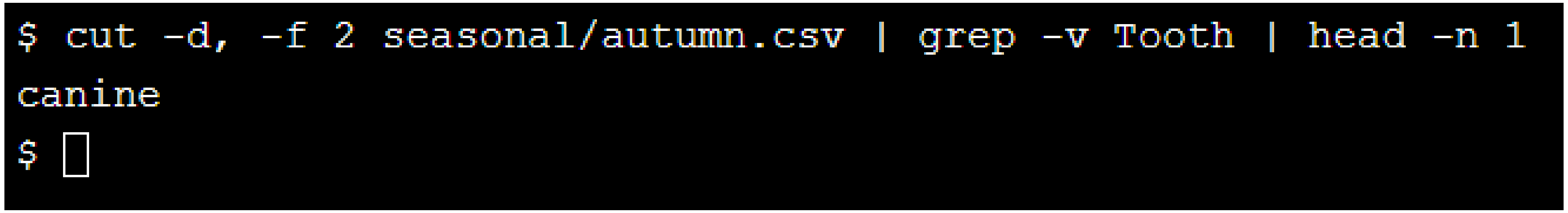
You can chain any number of commands together. For example, this command:

```
cut -d , -f 1 seasonal/spring.csv | grep -v Date | head -n 10
```

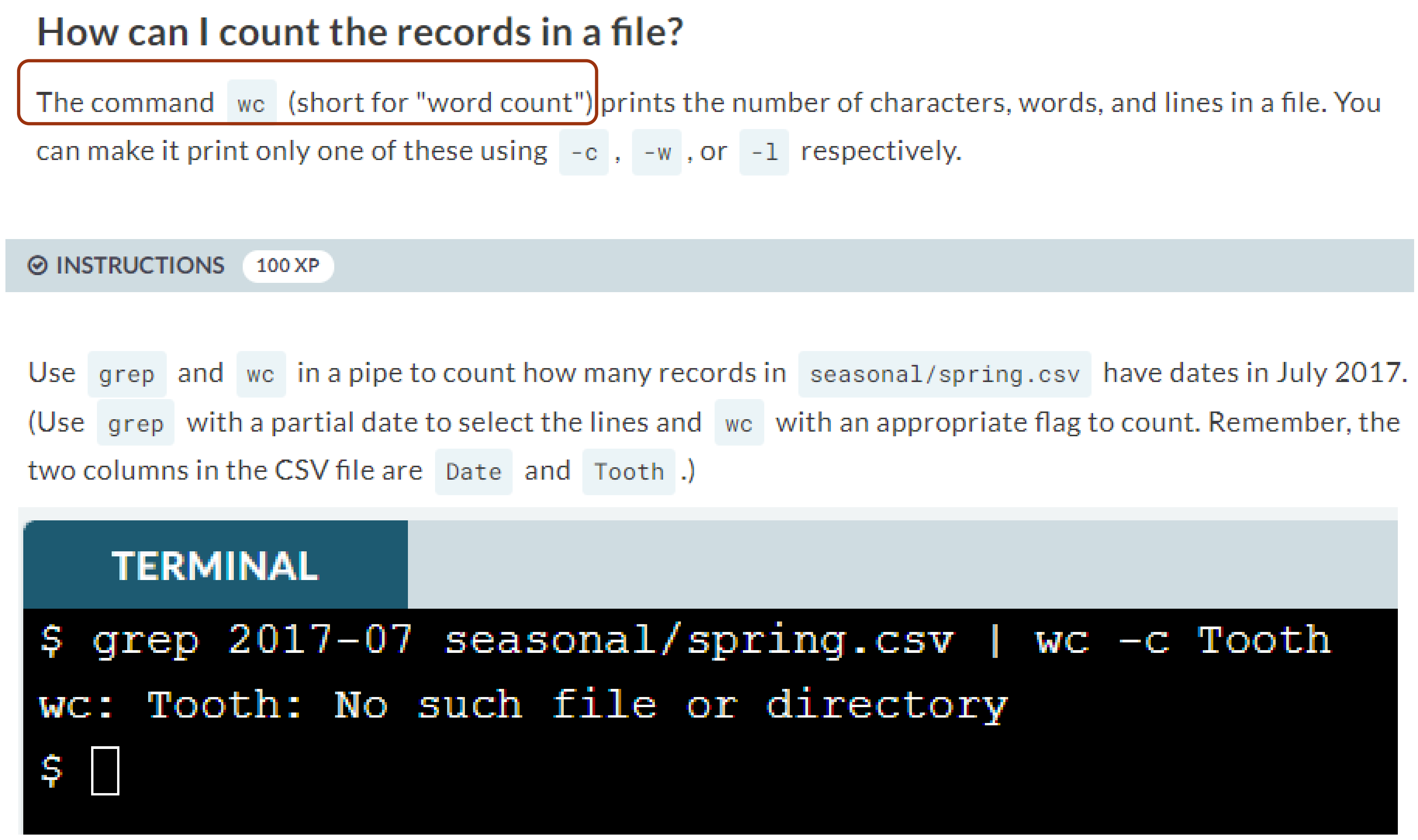
will:

1. select the first column from the spring data;
2. remove the header line containing the word "Date"; and
3. select the first 10 lines of actual data.

Write a pipeline that uses `cut`, `grep`, and `head` in that order to select the first value in column 2 of `seasonal/autumn.csv` *after* the header "Tooth".



```
$ cut -d, -f 2 seasonal/autumn.csv | grep -v Tooth | head -n 1
canine
$ 
```



How can I count the records in a file?

The command `wc` (short for "word count") prints the number of characters, words, and lines in a file. You can make it print only one of these using `-c` , `-w` , or `-l` respectively.

INSTRUCTIONS 100 XP

Use `grep` and `wc` in a pipe to count how many records in `seasonal/spring.csv` have dates in July 2017. (Use `grep` with a partial date to select the lines and `wc` with an appropriate flag to count. Remember, the two columns in the CSV file are `Date` and `Tooth` .)

TERMINAL

```
$ grep 2017-07 seasonal/spring.csv | wc -c Tooth
wc: Tooth: No such file or directory
$ 
```

How can I specify many files at once?

Most shell commands will work on multiple files if you give them multiple filenames. For example, you can get the first column from all of the seasonal data files at once like this:

```
cut -d , -f 1 seasonal/winter.csv seasonal/spring.csv seasonal/summer.csv seasonal/autumn.cs
```

But typing the names of many files over and over is a bad idea: it wastes time, and sooner or later you will either leave a file out or repeat a file's name. To make your life better, the shell allows you to use wildcards to specify a list of files with a single expression. The most common wildcard is `*`, which means "match zero or more characters". Using it, we can shorten the `cut` command above to this:

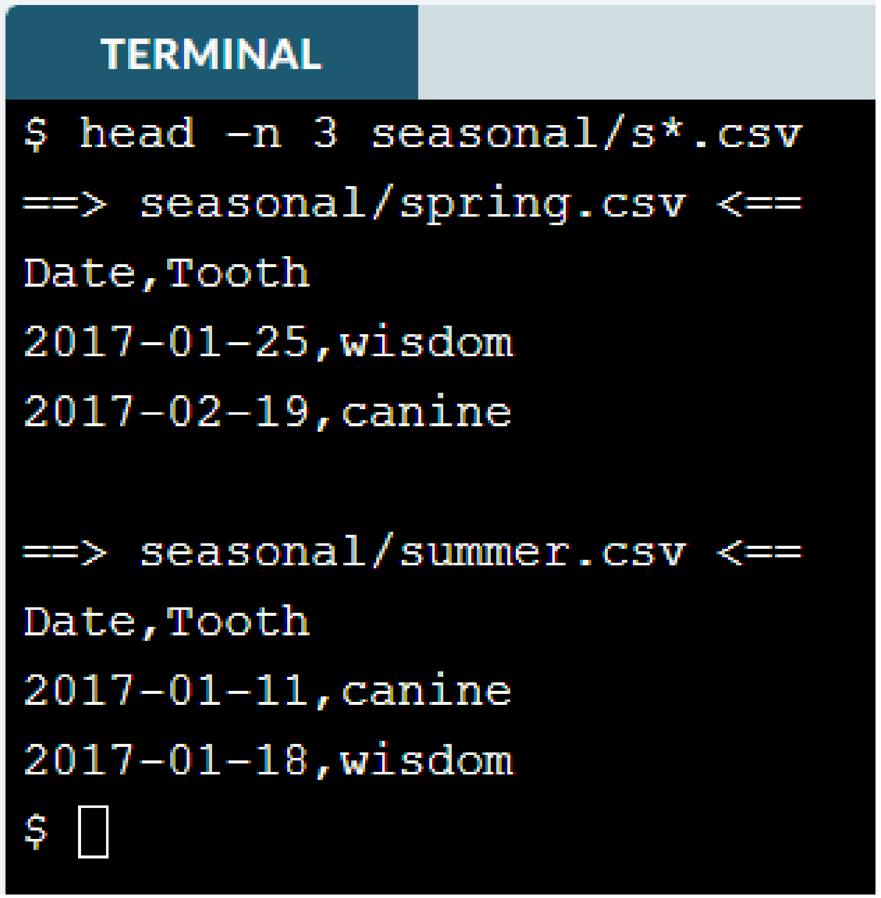
```
cut -d , -f 1 seasonal/*
```

or:

```
cut -d , -f 1 seasonal/*.csv
```

INSTRUCTIONS 100 XP

Write a single command using `head` to get the first three lines from both `seasonal/spring.csv` and `seasonal/summer.csv` (a total of six lines of data) but *not* from the autumn or winter data files. Use a wildcard instead of spelling out the files' names in full.



```
$ head -n 3 seasonal/s*.csv
==> seasonal/spring.csv <==
Date,Tooth
2017-01-25,wisdom
2017-02-19,canine

==> seasonal/summer.csv <==
Date,Tooth
2017-01-11,canine
2017-01-18,wisdom
$ 
```

What other wildcards can I use?

The shell has other wildcards as well, though they are less commonly used:

- `?` matches a single character, so `201?.txt` will match `2017.txt` or `2018.txt`, but not `2017-01.txt`.
- `[...]` matches any one of the characters inside the square brackets, so `201[78].txt` matches `2017.txt` or `2018.txt`, but not `2016.txt`.
- `{...}` matches any of the comma-separated patterns inside the curly brackets, so `{*.txt, *.csv}` matches any file whose name ends with `.txt` or `.csv`, but not files whose names end with `.pdf`.

Which expression would match `singh.pdf` and `johel.txt` but *not* `sandhu.pdf` or `sandhu.txt`?

Possible Answers

- `[sj]*.{.pdf, .txt}` press 1
- `{s*.pdf, j*.txt}` press 2
- `[singh,johel]{*.pdf, *.txt}` press 3
- [✓ `{singh.pdf, j*.txt}`] press 4

Submit Answer

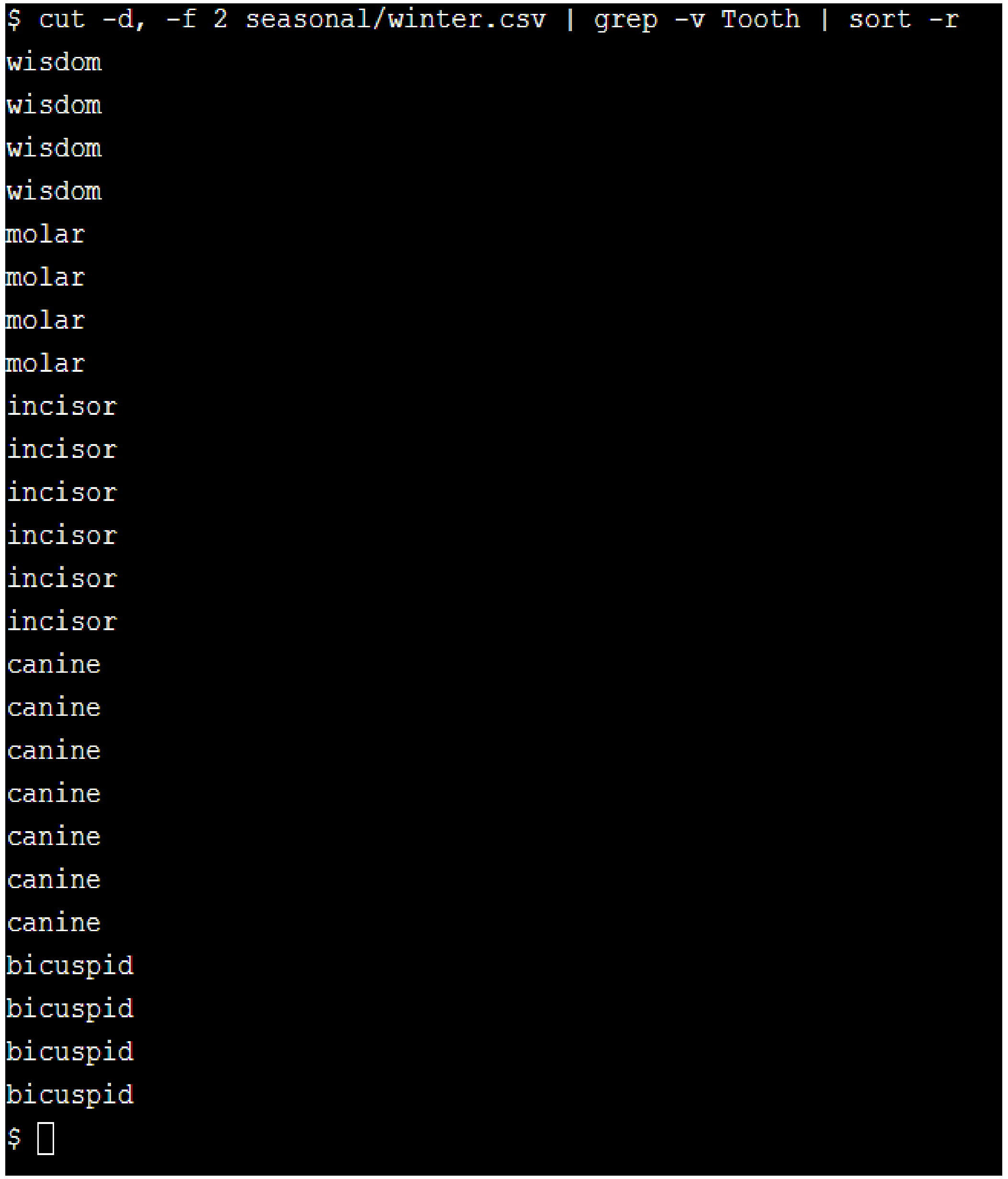
Take Hint (-15 XP)

How can I sort lines of text?

As its name suggests, `sort` puts data in order. By default it does this in ascending alphabetical order, but the flags `-n` and `-r` can be used to sort numerically and reverse the order of its output, while `-b` tells it to ignore leading blanks and `-f` tells it to fold case (i.e., be case-insensitive). Pipelines often use `grep` to get rid of unwanted records and then `sort` to put the remaining records in order.

INSTRUCTIONS 100 XP

Write a pipeline to sort the names of the teeth in `seasonal/winter.csv` in descending alphabetical order *without* including the header "Tooth". Use `cut`, `grep`, and `sort` in that order, and remember that the names are in column 2. (Please use tab completion to fill in the complete filename rather than using wildcards.)



```
$ cut -d, -f 2 seasonal/winter.csv | grep -v Tooth | sort -r
wisdom
wisdom
wisdom
wisdom
molar
molar
molar
molar
incisor
incisor
incisor
incisor
incisor
incisor
canine
canine
canine
canine
canine
canine
canine
bicuspid
bicuspid
bicuspid
bicuspid
$ 
```

How can I remove duplicate lines?

Another command that is often used with `sort` is `uniq`, whose job is to remove duplicated lines. More specifically, it removes *adjacent* duplicated lines. If a file contains:

```
2017-07-03
2017-07-03
2017-08-03
2017-08-03
```

then `uniq` will produce:

```
2017-07-03
2017-08-03
```

but if it contains:

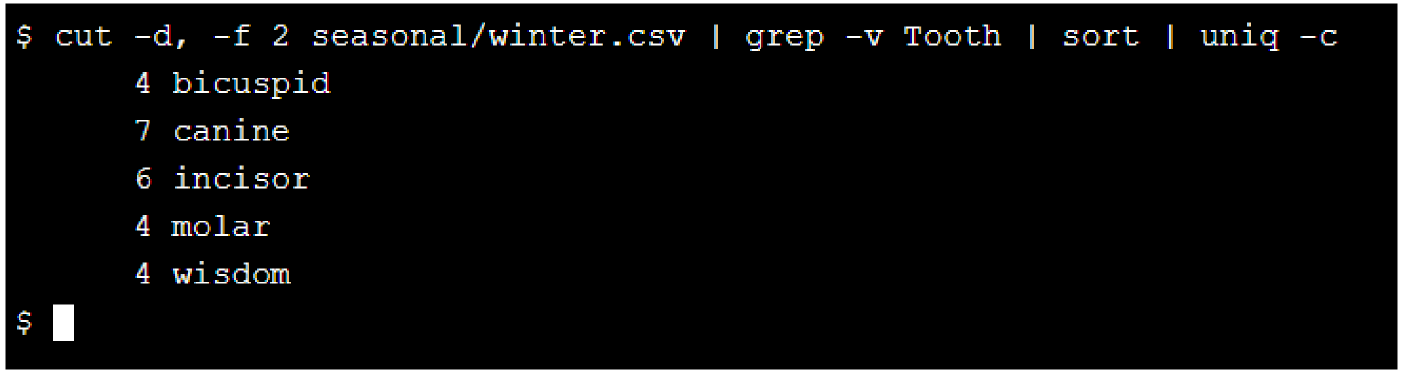
```
2017-07-03
2017-08-03
2017-07-03
2017-08-03
```

then `uniq` will print all four lines. The reason is that `uniq` is built to work with very large files. In order to remove non-adjacent lines from a file, it would have to keep the whole file in memory (or at least, all the unique lines seen so far). By only removing adjacent duplicates, it only has to keep the most recent unique line in memory.

Write a pipeline to:

- get the second column from `seasonal/winter.csv`,
- remove the word "Tooth" from the output so that only tooth names are displayed,
- sort the output so that all occurrences of a particular tooth name are adjacent; and
- display each tooth name once along with a count of how often it occurs.

Use `uniq -c` to display unique lines with a count of how often each occurs rather than using `uniq` and `wc`.



```
$ cut -d, -f 2 seasonal/winter.csv | grep -v Tooth | sort | uniq -c
      4 bicuspid
      7 canine
      6 incisor
      4 molar
      4 wisdom
$ 
```

How can I save the output of a pipe?

The shell lets us redirect the output of a sequence of piped commands:

```
cut -d , -f 2 seasonal/*.csv | grep -v Tooth > teeth-only.txt
```

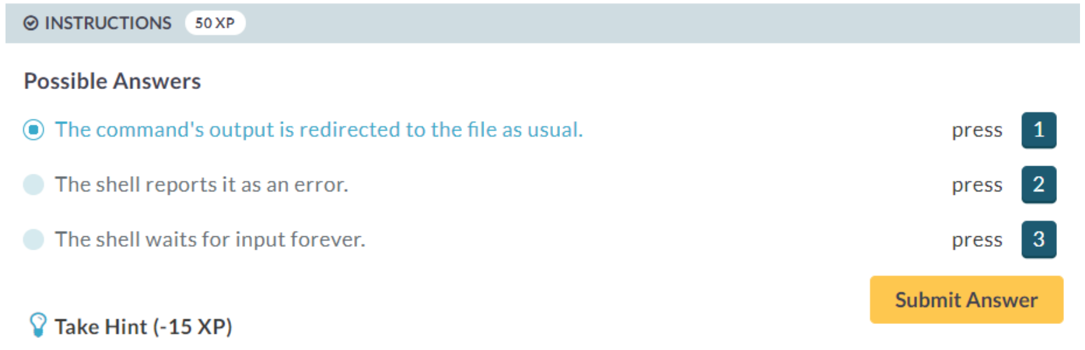
However, `>` must appear at the end of the pipeline: if we try to use it in the middle, like this:

```
cut -d , -f 2 seasonal/*.csv > teeth-only.txt | grep -v Tooth
```

then all of the output from `cut` is written to `teeth-only.txt`, so there is nothing left for `grep` and it waits forever for some input.

What happens if we put redirection at the front of a pipeline as in:

```
> result.txt head -n 3 seasonal/winter.csv
```



INSTRUCTIONS 50 XP

Possible Answers

- [✓ The command's output is redirected to the file as usual.] press 1
- The shell reports it as an error. press 2
- The shell waits for input forever. press 3

Submit Answer

Take Hint (-15 XP)

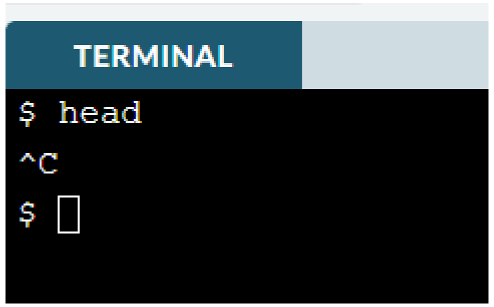
How can I stop a running program?

The commands and scripts that you have run so far have all executed quickly, but some tasks will take minutes, hours, or even days to complete. You may also mistakenly put redirection in the middle of a pipeline, causing it to hang up. If you decide that you don't want a program to keep running, you can type Ctrl-C to end it. This is often written `^C` in Unix documentation; note that the 'c' can be lower-case.

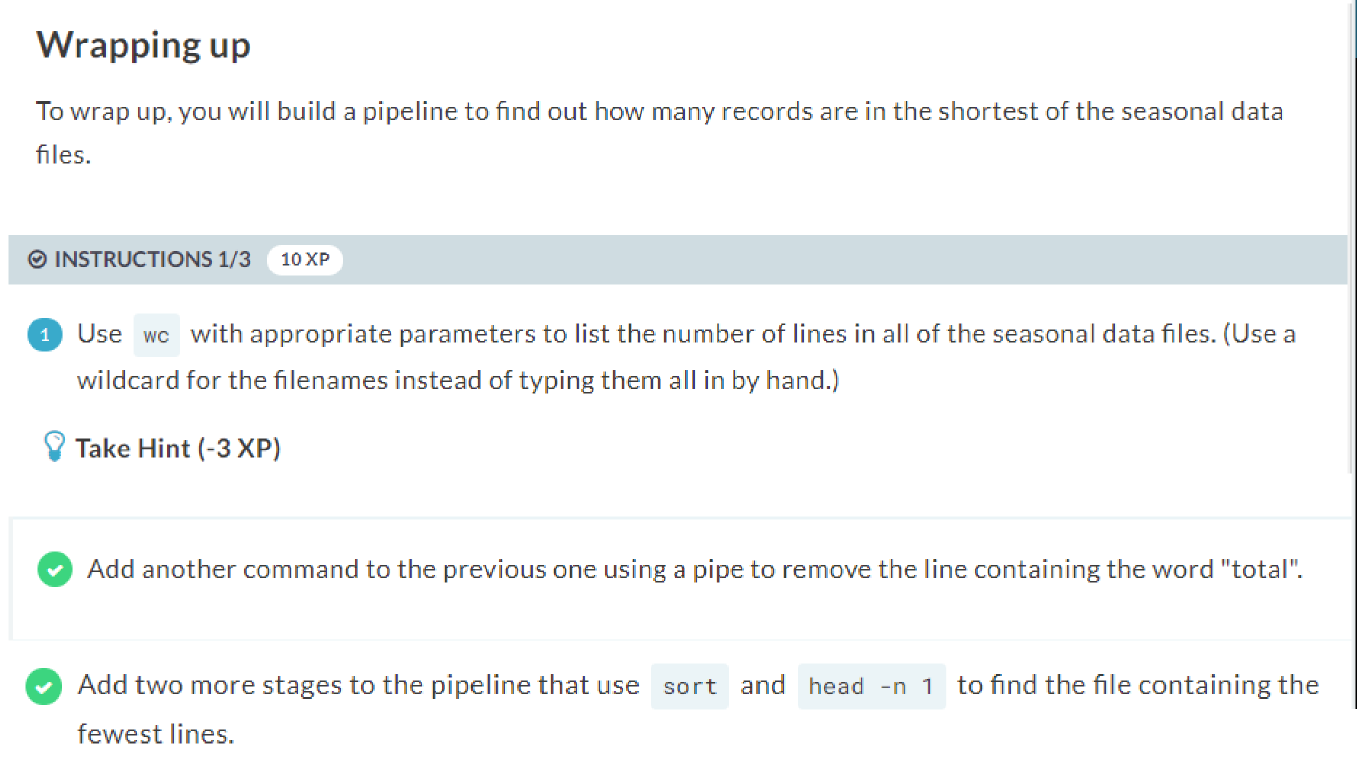
Run the command:

```
head
```

with no arguments (so that it waits for input that will never come) and then stop it by typing Ctrl-C.



```
$ head
^C
$ 
```



Wrapping up

To wrap up, you will build a pipeline to find out how many records are in the shortest of the seasonal data files.

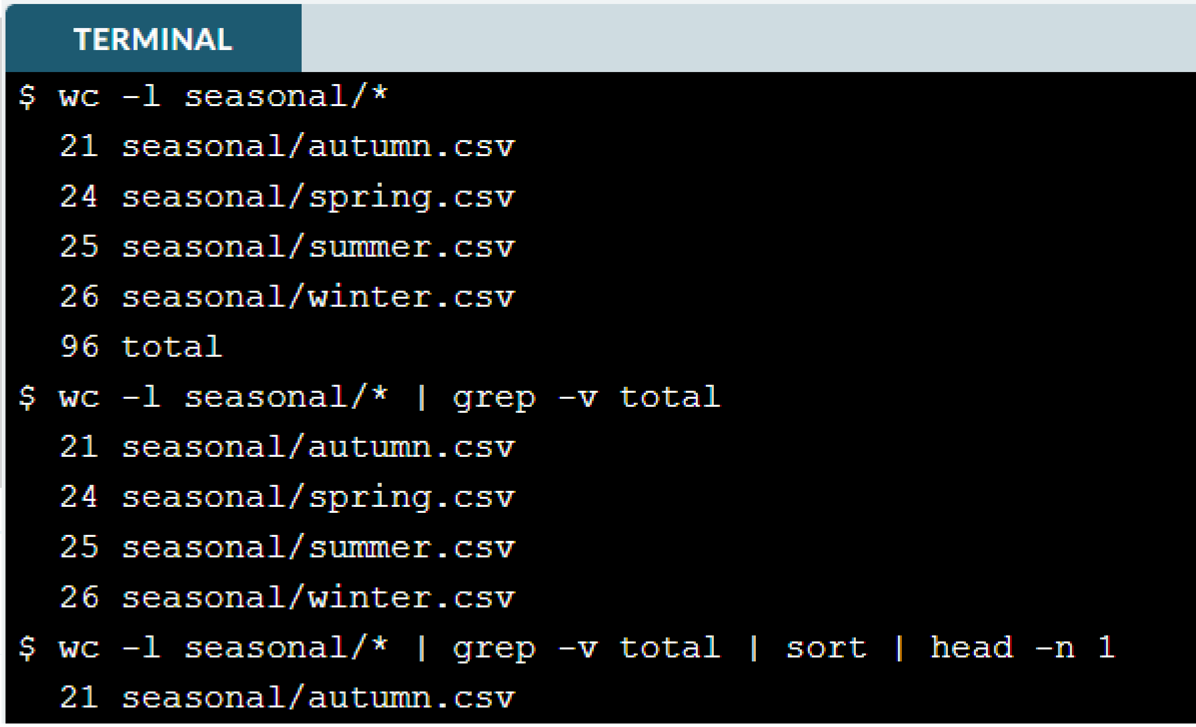
INSTRUCTIONS 1/3 10 XP

1. Use `wc` with appropriate parameters to list the number of lines in all of the seasonal data files. (Use a wildcard for the filenames instead of typing them all in by hand.)

Take Hint (-3 XP)

✓ Add another command to the previous one using a pipe to remove the line containing the word "total".

✓ Add two more stages to the pipeline that use `sort` and `head -n 1` to find the file containing the fewest lines.



TERMINAL

```
$ wc -l seasonal/*
  21 seasonal/autumn.csv
  24 seasonal/spring.csv
  25 seasonal/summer.csv
  26 seasonal/winter.csv
  96 total
$ wc -l seasonal/* | grep -v total
  21 seasonal/autumn.csv
  24 seasonal/spring.csv
  25 seasonal/summer.csv
  26 seasonal/winter.csv
$ wc -l seasonal/* | grep -v total | sort | head -n 1
  21 seasonal/autumn.csv
```

BATCH PROCESSING

- Most shell commands will process many files at once. This chapter will show you how to make your own pipelines do that. Along the way, you will see how the shell uses variables to store information.

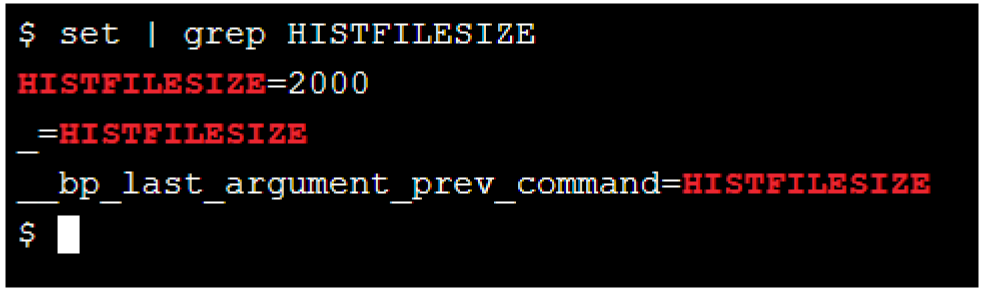
How does the shell store information?

Like other programs, the shell stores information in variables. Some of these, called **environment variables**, are available all the time. Environment variables' names are conventionally written in upper case, and a few of the more commonly-used ones are shown below.

Variable	Purpose	Value
`HOME`	User's home directory	`/home/repl`
`PWD`	Present working directory	Same as `pwd` command
`SHELL`	Which shell program is being used	`/bin/bash`
`USER`	User's ID	`repl`

To get a complete list (which is quite long), you can type `set` in the shell.

Use `set` and `grep` with a pipe to display the value of `HISTFILESIZE`, which determines how many old commands are stored in your command history. What is its value?



```
$ set | grep HISTFILESIZE
HISTFILESIZE=2000
_=HISTFILESIZE
__bp_last_argument_prev_command=HISTFILESIZE
$ 
```

How can I print a variable's value?

A simpler way to find a variable's value is to use a command called `echo`, which prints its arguments:

```
echo hello DataCamp!
```

```
hello DataCamp!
```

If you try to use it to print a variable's value like this:

```
echo USER
```

```
USER
```

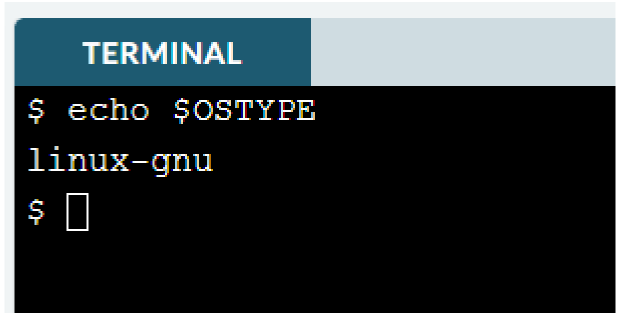
it will print the variable's name. To get the variable's value, you must put a dollar sign `$` in front of it:

```
echo $USER
```

```
repl
```

This is true everywhere: to get the value of a variable called `X`, you must write `$X`. (This is so that the shell can tell whether you mean "a file named X" or "the value of a variable named X".)

The variable `OSTYPE` holds the name of the kind of operating system you are using. Display its value using `echo`.



```
$ echo $OSTYPE
linux-gnu
$ 
```

How else does the shell store information?

The other kind of variable is called a **shell variable**, which is like a local variable in a programming language.

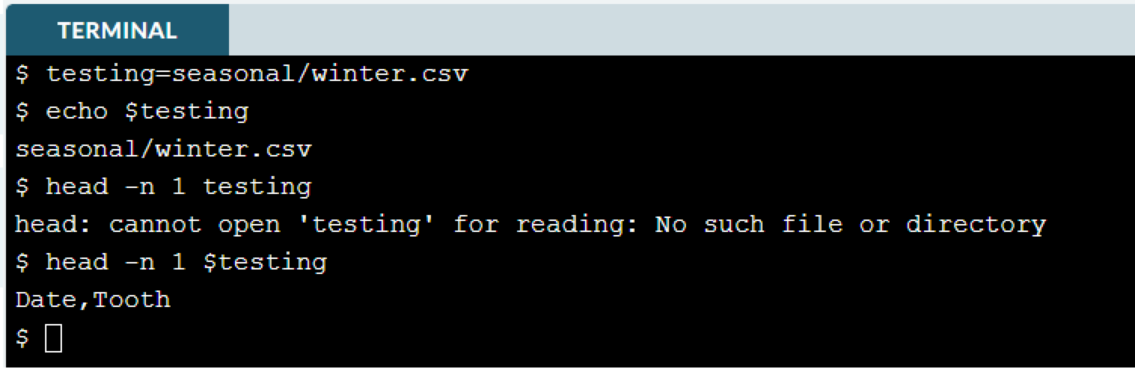
To create a shell variable, you simply assign a value to a name:

```
training=seasonal/summer.csv
```

without any spaces before or after the `=` sign. Once you have done this, you can check the variable's value with:

```
echo $training
```

```
seasonal/summer.csv
```


TERMINAL

```
$ testing=seasonal/winter.csv
$ echo $testing
seasonal/winter.csv
$ head -n 1 testing
head: cannot open 'testing' for reading: No such file or directory
$ head -n 1 $testing
Date,Tooth
$ 
```

1. Define a variable called `testing` with the value `seasonal/winter.csv`.
2. Use `head -n 1 SOMETHING` to get the first line from `seasonal/winter.csv` using the value of the variable `testing` instead of the name of the file.

How can I repeat a command many times?

Shell variables are also used in **loops**, which repeat commands many times. If we run this command:

```
for suffix in gif jpg png; do echo $suffix; done
```

it produces:

```
gif
jpg
png
```

The loop's parts are:

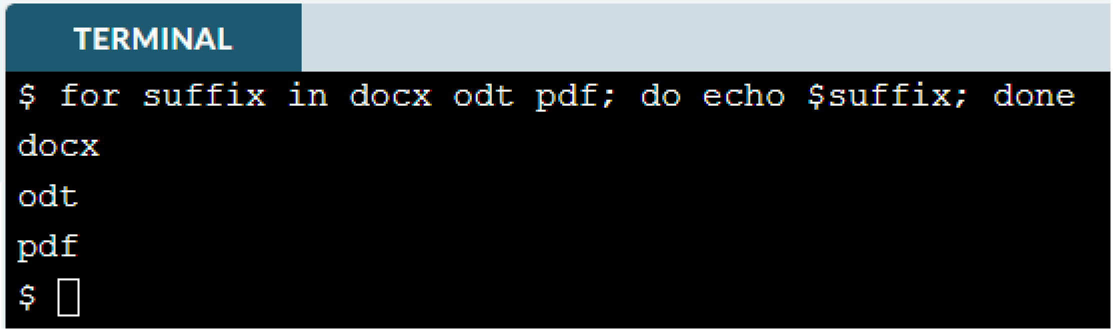
1. The skeleton `for` ...variable... `in` ...list... `; do` ...body... `; done`
2. The list of things the loop is to process (in our case, the words `gif` , `jpg` , and `png`).
3. The variable that keeps track of which thing the loop is currently processing (in our case, `suffix`).
4. The body of the loop that does the processing (in our case, `echo $suffix`).

Notice that the body uses `$suffix` to get the variable's value instead of just `suffix` , just like it does with any other shell variable. Also notice where the semi-colons go: the first one comes between the list and the keyword `do` , and the second comes between the body and the keyword `done` .

Modify the loop so that it prints:

```
docx
odt
pdf
```

Please use `suffix` as the name of the loop variable.



```
$ for suffix in docx odt pdf; do echo $suffix; done
docx
odt
pdf
$ 
```

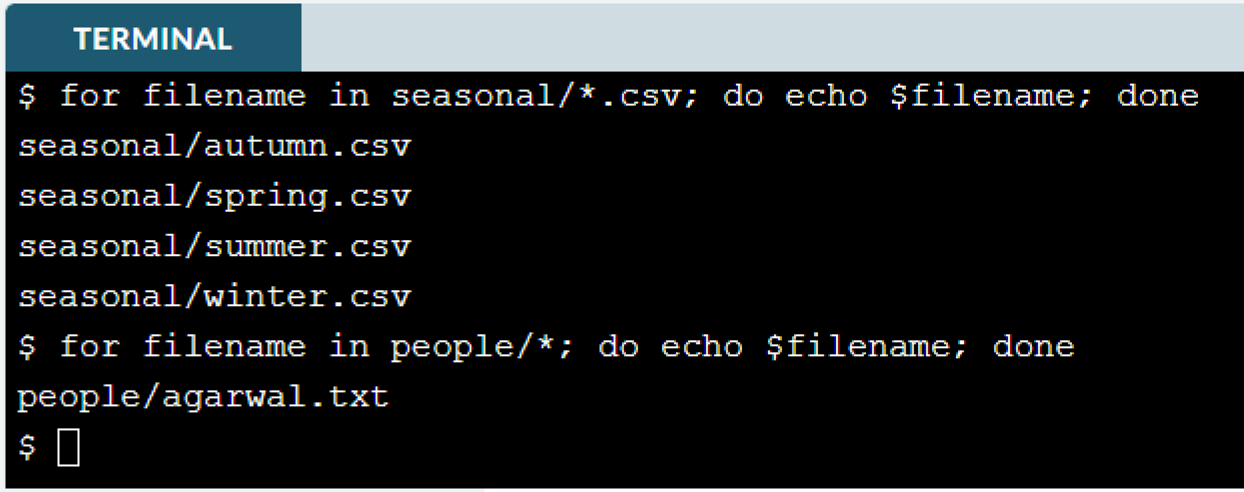
How can I repeat a command once for each file?

You can always type in the names of the files you want to process when writing the loop, but it's usually better to use wildcards. Try running this loop in the console:

```
for filename in seasonal/*.csv; do echo $filename; done
```

It prints:

```
seasonal/autumn.csv
seasonal/spring.csv
seasonal/summer.csv
seasonal/winter.csv
```


TERMINAL

```
$ for filename in seasonal/*.csv; do echo $filename; done
seasonal/autumn.csv
seasonal/spring.csv
seasonal/summer.csv
seasonal/winter.csv
$ for filename in people/*; do echo $filename; done
people/agarwal.txt
$ 
```

because the shell expands `seasonal/*.csv` to be a list of four filenames before it runs the loop.

INSTRUCTIONS 100 XP

Modify the wildcard expression to `people/*` so that the loop prints the names of the files in the `people` directory regardless of what suffix they do or don't have. Please use `filename` as the name of your loop variable.

How can I record the names of a set of files?

People often set a variable using a wildcard expression to record a list of filenames. For example, if you define `datasets` like this:

```
datasets=seasonal/*.csv
```

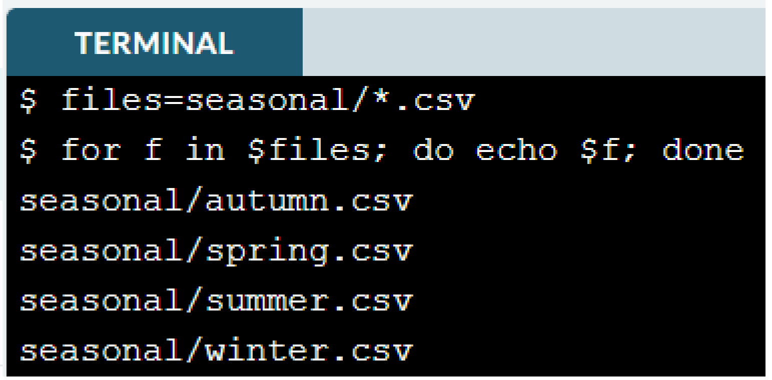
you can display the files' names later using:

```
for filename in $datasets; do echo $filename; done
```

This saves typing and makes errors less likely.

If you run these two commands in your home directory, how many lines of output will they print?

```
files=seasonal/*.csv
for f in $files; do echo $f; done
```



```
$ files=seasonal/*.csv
$ for f in $files; do echo $f; done
seasonal/autumn.csv
seasonal/spring.csv
seasonal/summer.csv
seasonal/winter.csv
```

How can I run many commands in a single loop?

Printing filenames is useful for debugging, but the real purpose of loops is to do things with multiple files. This loop prints the second line of each data file:

```
for file in seasonal/*.csv; do head -n 2 $file | tail -n 1; done
```

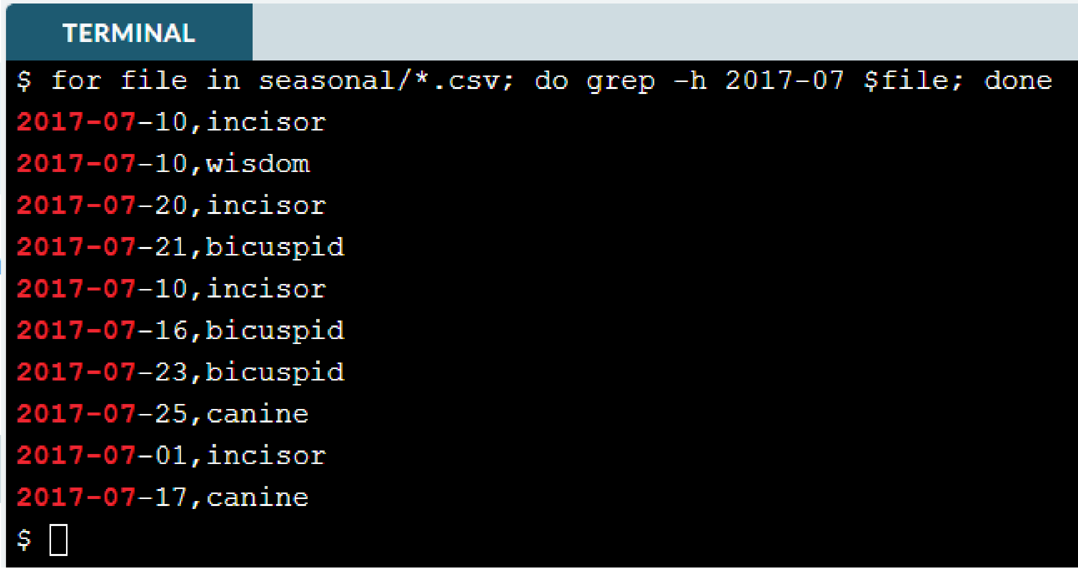
It has the same structure as the other loops you have already seen: all that's different is that its body is a pipeline of two commands instead of a single command.

INSTRUCTIONS 100 XP

Write a loop that produces the same output as

```
grep -h 2017-07 seasonal/*.csv
```

but uses a loop to process each file separately. Please use `file` as the name of the loop variable, and remember that the `-h` flag used above tells `grep` *not* to print filenames in the output.



TERMINAL

```
$ for file in seasonal/*.csv; do grep -h 2017-07 $file; done
2017-07-10,incisor
2017-07-10,wisdom
2017-07-20,incisor
2017-07-21,bicuspid
2017-07-10,incisor
2017-07-16,bicuspid
2017-07-23,bicuspid
2017-07-25,canine
2017-07-01,incisor
2017-07-17,canine
$ 
```

Why shouldn't I use spaces in filenames?

It's easy and sensible to give files multi-word names like `July 2017.csv` when you are using a graphical file explorer. However, this causes problems when you are working in the shell. For example, suppose you wanted to rename `July 2017.csv` to be `2017 July data.csv`. You cannot type:

```
mv July 2017.csv 2017 July data.csv
```

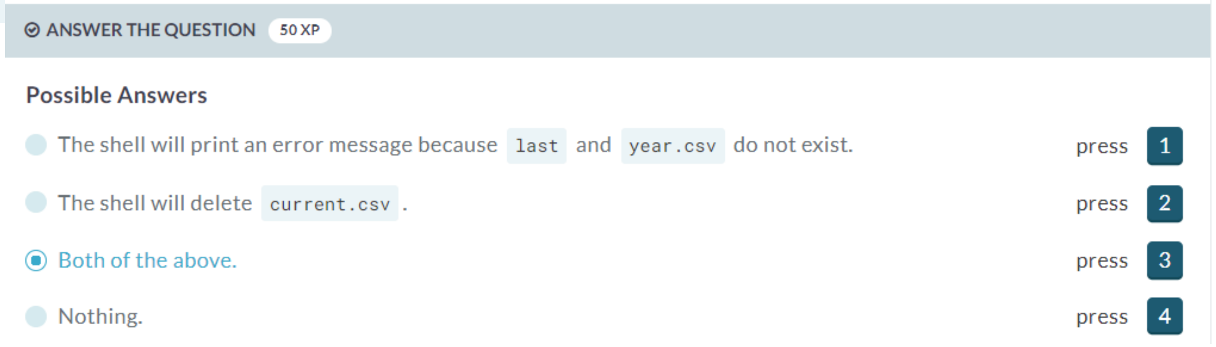
because it looks to the shell as though you are trying to move four files called `July`, `2017.csv`, `2017`, and `July` (again) into a directory called `data.csv`. Instead, you have to quote the files' names so that the shell treats each one as a single parameter:

```
mv 'July 2017.csv' '2017 July data.csv'
```

If you have two files called `current.csv` and `last year.csv` (with a space in its name) and you type:

```
rm current.csv last year.csv
```

what will happen:



ANSWER THE QUESTION 50 XP

Possible Answers

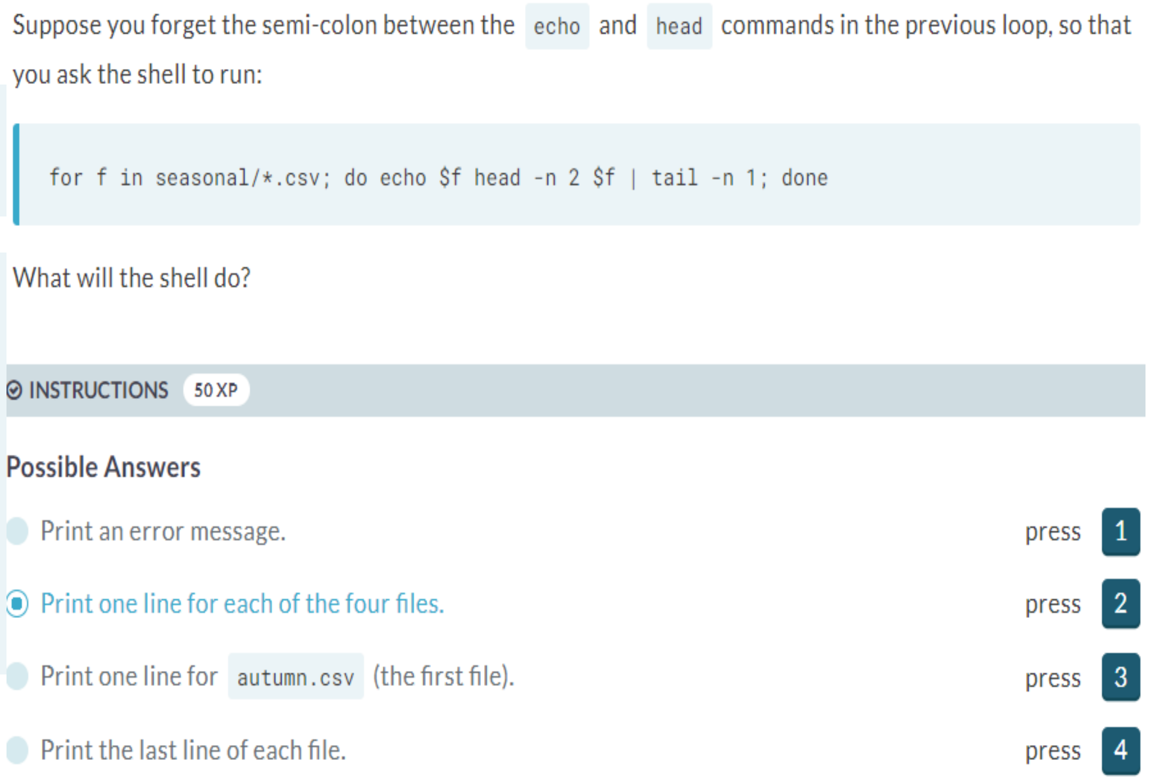
- The shell will print an error message because `last` and `year.csv` do not exist. press 1
- The shell will delete `current.csv`. press 2
- [✓ Both of the above.] press 3
- Nothing. press 4

How can I do many things in a single loop?

The loops you have seen so far all have a single command or pipeline in their body, but a loop can contain any number of commands. To tell the shell where one ends and the next begins, you must separate them with semi-colons:

```
for f in seasonal/*.csv; do echo $f; head -n 2 $f | tail -n 1; done
```

```
seasonal/autumn.csv
2017-01-05,canine
seasonal/spring.csv
2017-01-25,wisdom
seasonal/summer.csv
2017-01-11,canine
seasonal/winter.csv
2017-01-03,bicuspid
```



Suppose you forget the semi-colon between the `echo` and `head` commands in the previous loop, so that you ask the shell to run:

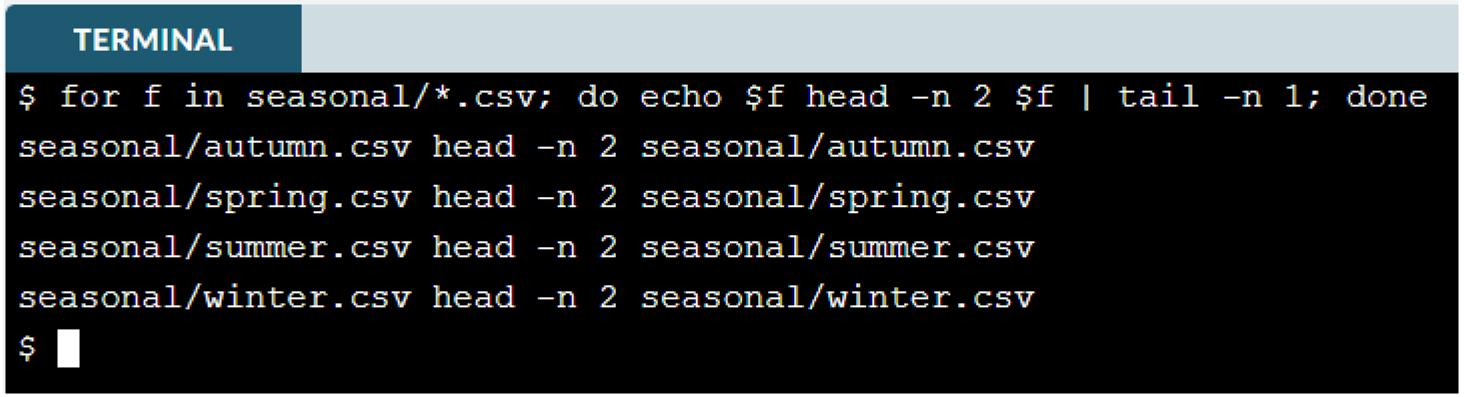
```
for f in seasonal/*.csv; do echo $f head -n 2 $f | tail -n 1; done
```

What will the shell do?

INSTRUCTIONS 50 XP

Possible Answers

- Print an error message. press 1
- [✓ Print one line for each of the four files.] press 2
- Print one line for `autumn.csv` (the first file). press 3
- Print the last line of each file. press 4



TERMINAL

```
$ for f in seasonal/*.csv; do echo $f head -n 2 $f | tail -n 1; done
seasonal/autumn.csv head -n 2 seasonal/autumn.csv
seasonal/spring.csv head -n 2 seasonal/spring.csv
seasonal/summer.csv head -n 2 seasonal/summer.csv
seasonal/winter.csv head -n 2 seasonal/winter.csv
$
```

CREATING NEW TOOLS

- History lets you repeat things with just a few keystrokes, and pipes let you combine existing commands to create new ones. In this chapter, you will see how to go one step further and create new commands of your own.

How can I edit a file?

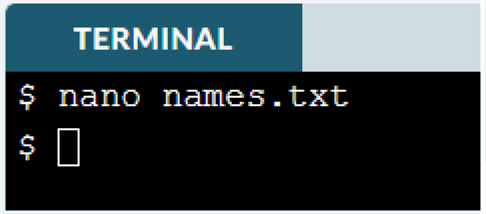
Unix has a bewildering variety of text editors. For this course, we will use a simple one called Nano. If you type `nano filename`, it will open `filename` for editing (or create it if it doesn't already exist). You can move around with the arrow keys, delete characters using backspace, and do other operations with control-key combinations:

- Ctrl-K: delete a line.
- Ctrl-U: un-delete a line.
- Ctrl-O: save the file ('O' stands for 'output').
- Ctrl-X: exit the editor.

INSTRUCTIONS 100 XP

Run `nano names.txt` to edit a new file in your home directory and enter the following four lines:

```
Lovelace
Hopper
Johnson
Wilson
```

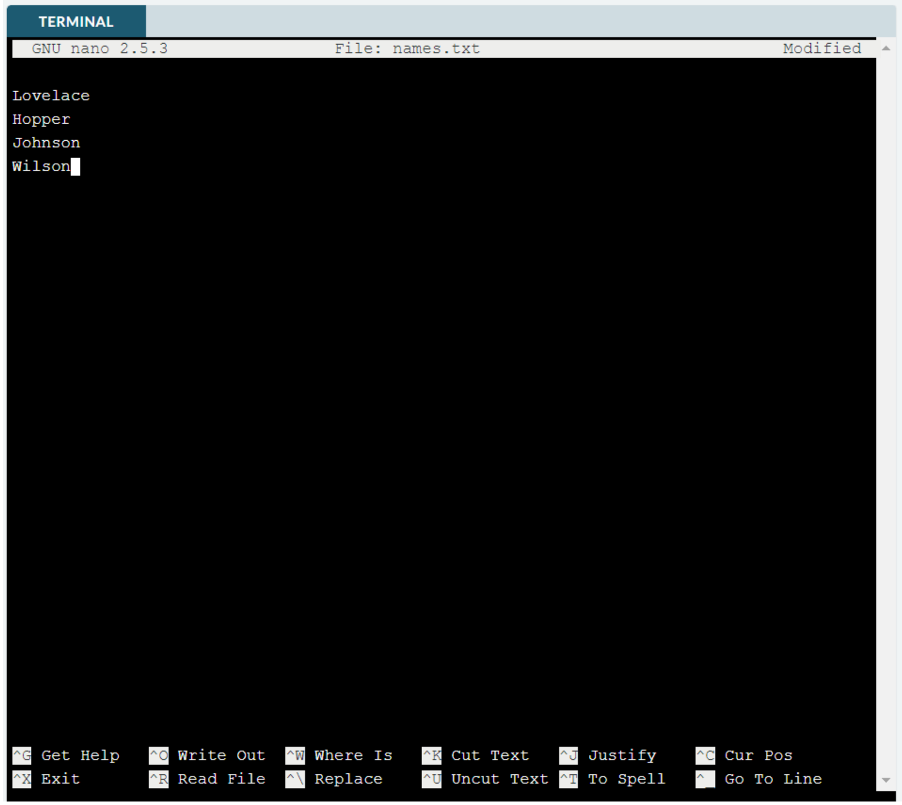


TERMINAL

```
$ nano names.txt
$ 
```

To save what you have written, type Ctrl-O to write the file out, then Enter to confirm the filename, then Ctrl-X and Enter to exit the editor.

Note: if you view our solution, it uses `cp` instead of `nano` for our automated back end to check, because the back end can't edit files interactively.



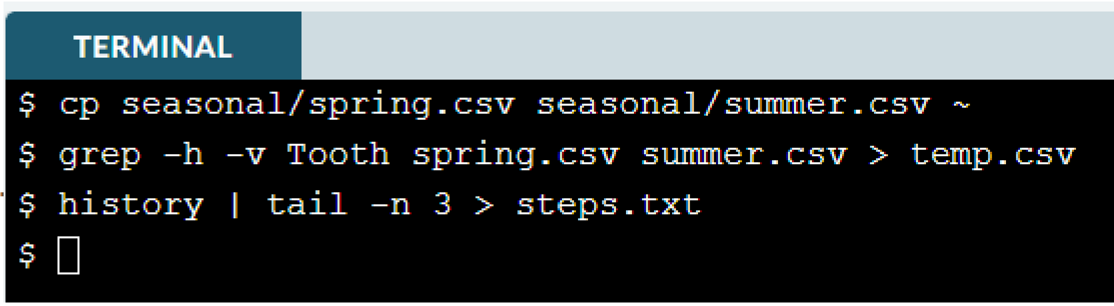
TERMINAL

```
  GNU nano 2.5.3               File: names.txt                        Modified

Lovelace
Hopper
Johnson
Wilson

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line
```

How can I record what I just did?

When you are doing a complex analysis, you will often want to keep a record of the commands you used. You can do this with the tools you have already seen:

1. Run `history`.
2. Pipe its output to `tail -n 10` (or however many recent steps you want to save).
3. Redirect that to a file called something like `figure-5.history`.


TERMINAL

```
$ cp seasonal/spring.csv seasonal/summer.csv ~
$ grep -h -v Tooth spring.csv summer.csv > temp.csv
$ history | tail -n 3 > steps.txt
$ 
```

This is better than writing things down in a lab notebook because it is guaranteed not to miss any steps. It also illustrates the central idea of the shell: simple tools that produce and consume lines of text can be combined in a wide variety of ways to solve a broad range of problems.

✓ Copy the files `seasonal/spring.csv` and `seasonal/summer.csv` to your home directory.

✓ Use `grep` with the `-h` flag (to stop it from printing filenames) and `-v Tooth` (to select lines that *don't* match the header line) to select the data records from `spring.csv` and `summer.csv` in that order and redirect the output to `temp.csv`.

3 Pipe `history` into `tail -n 3` and redirect the output to `steps.txt` to save the last three commands in a file. (You need to save three instead of just two because the `history` command itself will be in the list.)

How can I save commands to re-run later?

You have been using the shell interactively so far. But since the commands you type in are just text, you can store them in files for the shell to run over and over again. To start exploring this powerful capability, put the following command in a file called `headers.sh` :

```
head -n 1 seasonal/*.csv
```

This command selects the first row from each of the CSV files in the `seasonal` directory. Once you have created this file, you can run it by typing:

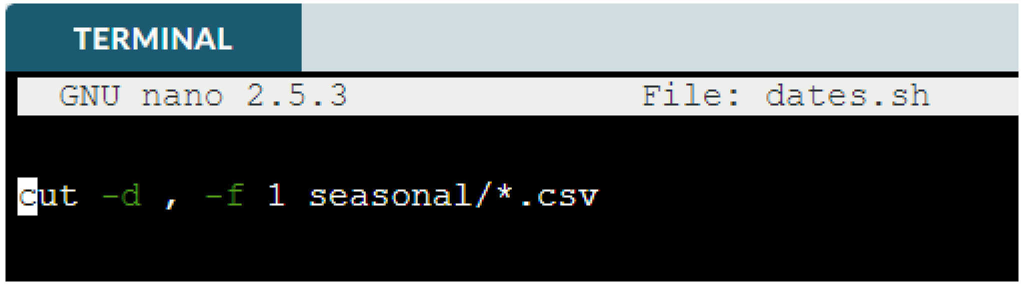
```
bash headers.sh
```

This tells the shell (which is just a program called `bash`) to run the commands contained in the file `headers.sh` , which produces the same output as running the commands directly.

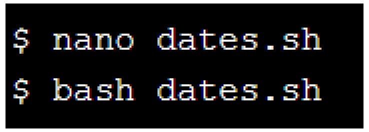
1. Use `nano dates.sh` to create a file called `dates.sh` that uses this command:

   ```
   cut -d , -f 1 seasonal/*.csv
   ```

   to extract the first column from all of the CSV files in `seasonal` .



```
  GNU nano 2.5.3                    File: dates.sh

cut -d , -f 1 seasonal/*.csv
```



```
$ nano dates.sh
$ bash dates.sh
```

How can I re-use pipes?

A file full of shell commands is called a shell script, or sometimes just a "script" for short. Scripts don't have to have names ending in `.sh`, but this lesson will use that convention to help you keep track of which files are scripts.

Scripts may contain pipes. For example, if `all-dates.sh` contains this line:

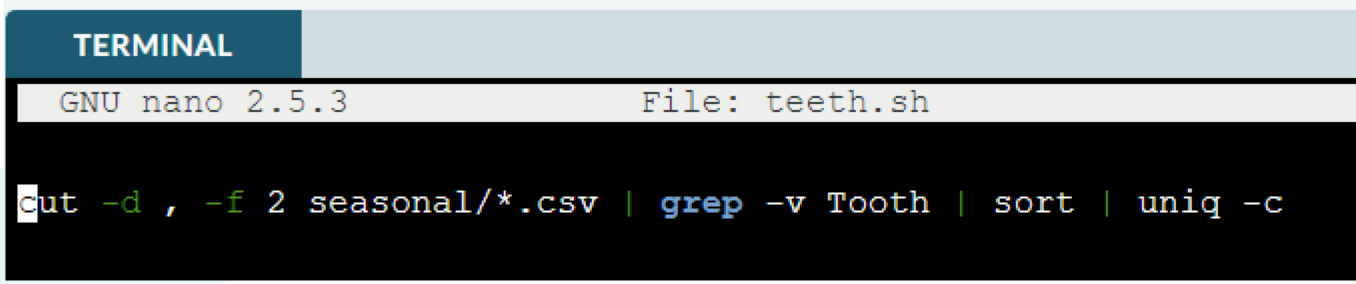
```
cut -d , -f 1 seasonal/*.csv | grep -v Date | sort | uniq
```

then:

```
bash all-dates.sh > dates.out
```

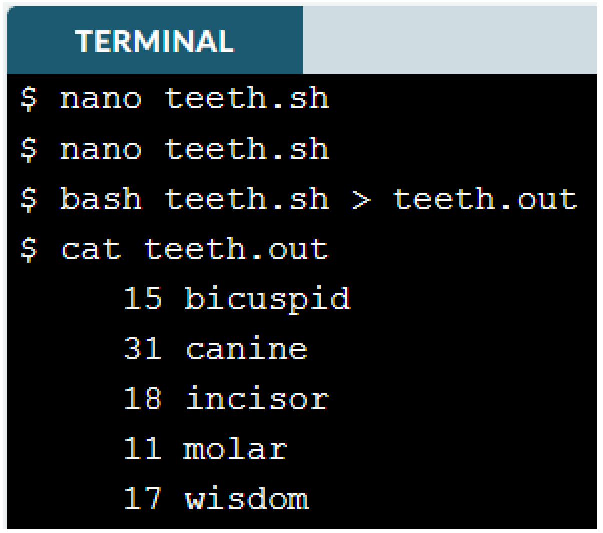
will extract the unique dates from the seasonal data files and save them in `dates.out`.

- ✓ Use Nano to edit the shell script `teeth.sh` and replace the two `____` placeholders with `seasonal/*.csv` and `-c` so that this script prints a count of the number of times each tooth name appears in the CSV files in the `seasonal` directory.
- ✓ Use `bash` to run `teeth.sh` and `>` to redirect its output to `teeth.out`.
- ✓ Run `cat teeth.out` to inspect your results.



TERMINAL

```
  GNU nano 2.5.3                    File: teeth.sh

cut -d , -f 2 seasonal/*.csv | grep -v Tooth | sort | uniq -c
```



TERMINAL

```
$ nano teeth.sh
$ nano teeth.sh
$ bash teeth.sh > teeth.out
$ cat teeth.out
     15 bicuspid
     31 canine
     18 incisor
     11 molar
     17 wisdom
```

How can I pass filenames to scripts?

A script that processes specific files is useful as a record of what you did, but one that allows you to process any files you want is more useful. To support this, you can use the special expression `$@` (dollar sign immediately followed by at-sign) to mean "all of the command-line parameters given to the script". For example, if `unique-lines.sh` contains this:

```
sort $@ | uniq
```

then when you run:

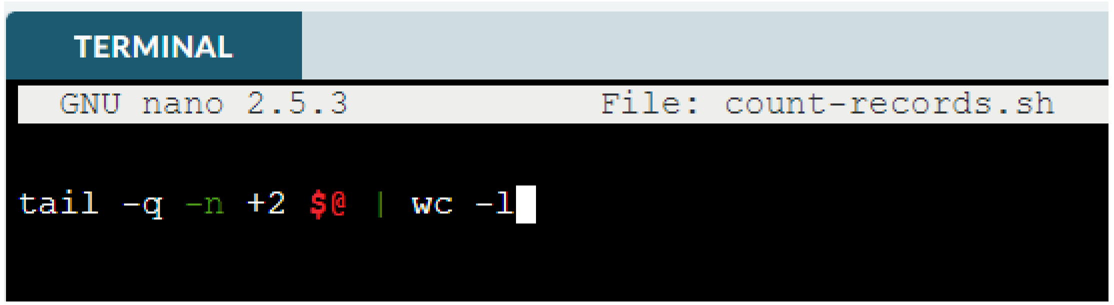
```
bash unique-lines.sh seasonal/summer.csv
```

the shell replaces `$@` with `seasonal/summer.csv` and processes one file. If you run this:

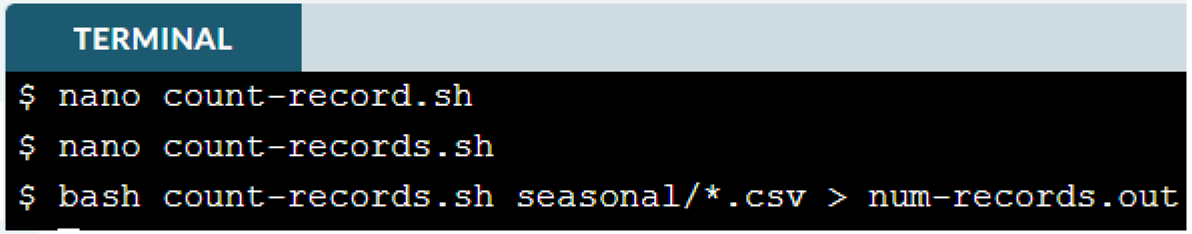
```
bash unique-lines.sh seasonal/summer.csv seasonal/autumn.csv
```

it processes two data files, and so on.

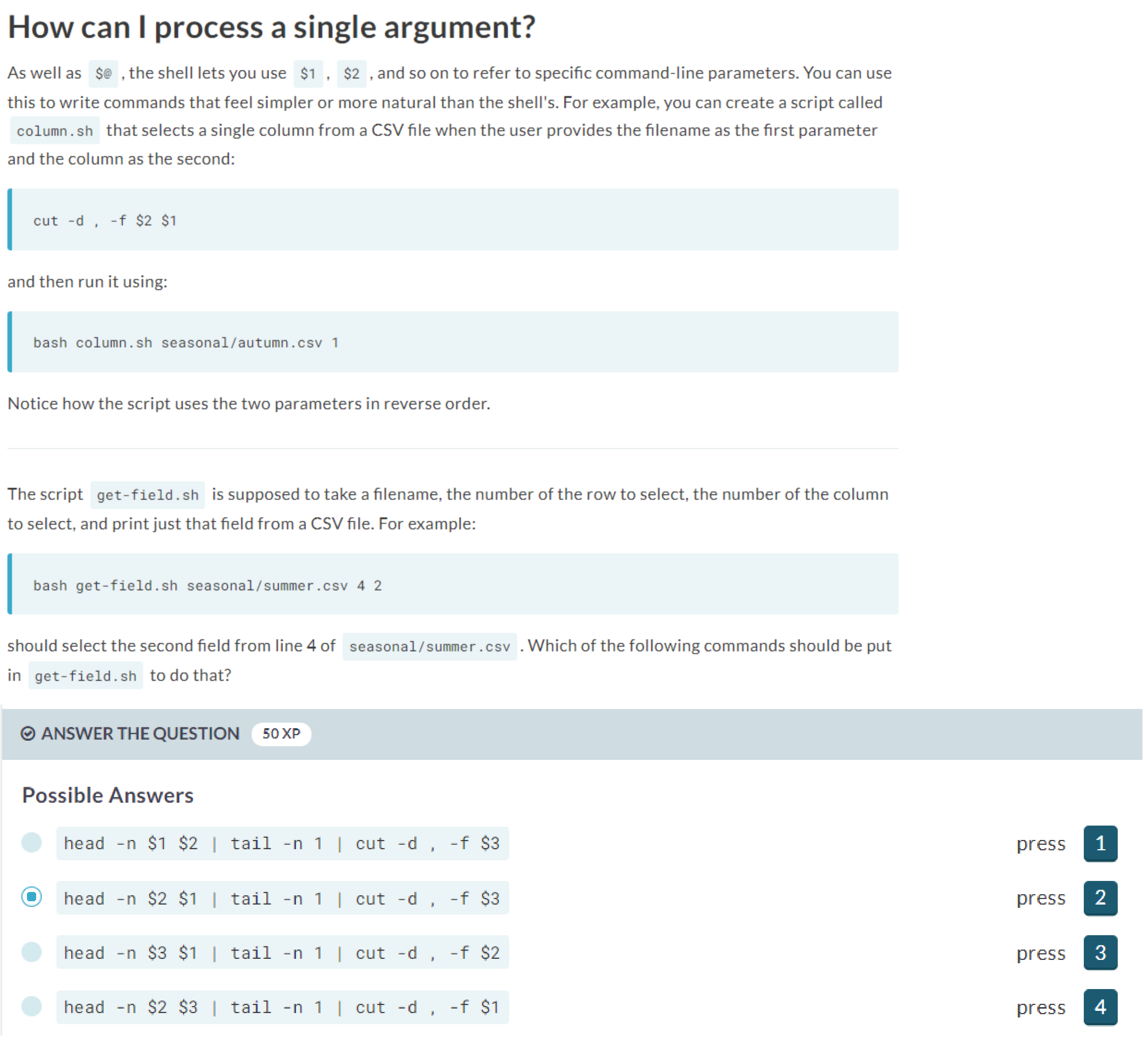
✓ Edit the script `count-records.sh` with Nano and fill in the two `____` placeholders with `$@` and `-l` respectively so that it counts the number of lines in one or more files, excluding the first line of each.



```
  GNU nano 2.5.3                  File: count-records.sh

tail -q -n +2 $@ | wc -l
```



```
$ nano count-record.sh
$ nano count-records.sh
$ bash count-records.sh seasonal/*.csv > num-records.out
```



How can I process a single argument?

As well as `$@`, the shell lets you use `$1`, `$2`, and so on to refer to specific command-line parameters. You can use this to write commands that feel simpler or more natural than the shell's. For example, you can create a script called `column.sh` that selects a single column from a CSV file when the user provides the filename as the first parameter and the column as the second:

```
cut -d , -f $2 $1
```

and then run it using:

```
bash column.sh seasonal/autumn.csv 1
```

Notice how the script uses the two parameters in reverse order.

The script `get-field.sh` is supposed to take a filename, the number of the row to select, the number of the column to select, and print just that field from a CSV file. For example:

```
bash get-field.sh seasonal/summer.csv 4 2
```

should select the second field from line 4 of `seasonal/summer.csv`. Which of the following commands should be put in `get-field.sh` to do that?

ANSWER THE QUESTION 50 XP

Possible Answers

- `head -n $1 $2 | tail -n 1 | cut -d , -f $3` press 1
- [✓ `head -n $2 $1 | tail -n 1 | cut -d , -f $3`] press 2
- `head -n $3 $1 | tail -n 1 | cut -d , -f $2` press 3
- `head -n $2 $3 | tail -n 1 | cut -d , -f $1` press 4

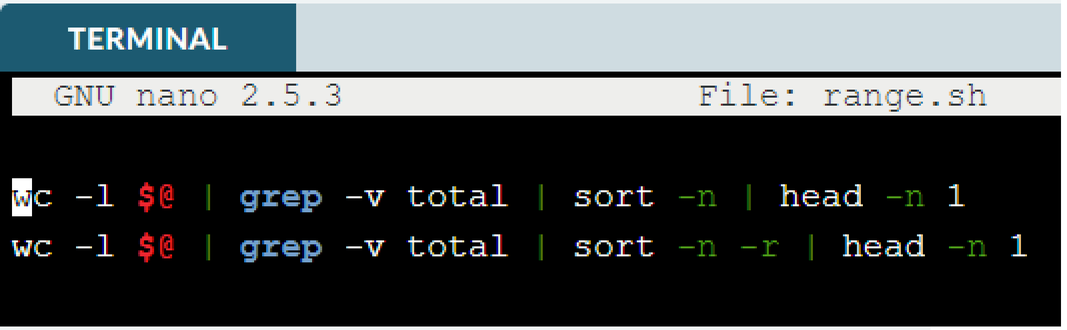
How can one shell script do many things?

Our shells scripts so far have had a single command or pipe, but a script can contain many lines of commands. For example, you can create one that tells you how many records are in the shortest and longest of your data files, i.e., the range of your datasets' lengths.

INSTRUCTIONS 1/4 10 XP

1 2 3 4

Use Nano to edit the script `range.sh` and replace the two `____` placeholders with `$@` and `-v` so that it lists the names and number of lines in all of the files given on the command line *without* showing the total number of lines in all files. (Do not try to subtract the column header lines from the files.)



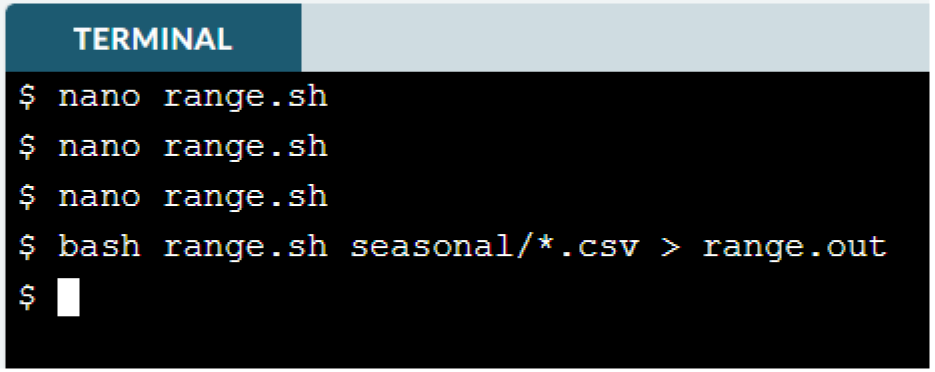
TERMINAL

```
  GNU nano 2.5.3                  File: range.sh

wc -l $@ | grep -v total | sort -n | head -n 1
wc -l $@ | grep -v total | sort -n -r | head -n 1
```

Add `sort -n` and `head -n 1` in that order to the pipeline in `range.sh` to display the count of the shortest file given to it.

Add a second line to `range.sh` to print the name and record count of the *longest* file in t *well as* the shortest. This line should be a duplicate of the one you have already written, b `sort -n -r` rather than `sort -n`.

Run the script on the files in the `seasonal` directory using `seasonal/*.csv` to match and redirect the output using `>` to a file called `range.out` in your home directory.



TERMINAL

```
$ nano range.sh
$ nano range.sh
$ nano range.sh
$ bash range.sh seasonal/*.csv > range.out
$ 
```

How can I write loops in a shell script?

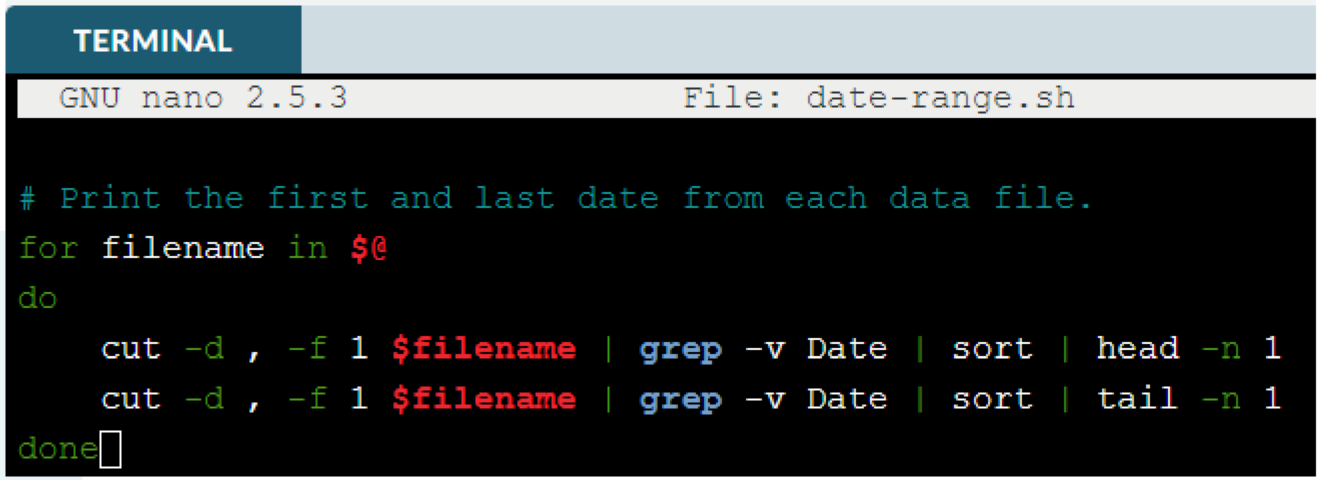
Shell scripts can also contain loops. You can write them using semi-colons, or split them across lines without semi-colons to make them more readable:

```
# Print the first and last data records of each file.
for filename in $@
do
    head -n 2 $filename | tail -n 1
    tail -n 1 $filename
done
```

(You don't have to indent the commands inside the loop, but doing so makes things clearer.)

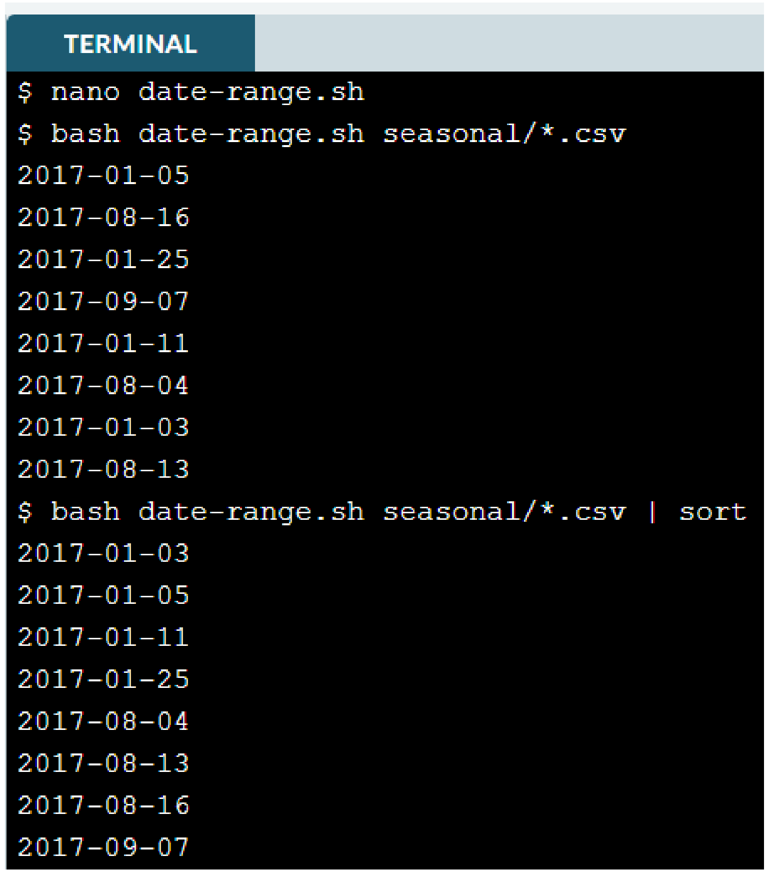
The first line of this script is a comment to tell readers what the script does. Comments start with the `#` character and run to the end of the line. Your future self will thank you for adding brief explanations like the one shown here to every script you write.

1. Fill in the placeholders in the script `date-range.sh` with `$filename` (twice), `head`, and `tail` so that it prints the first and last date from one or more files.
2. Run `date-range.sh` on all four of the seasonal data files using `seasonal/*.csv` to match their names.
3. ✓ Run `date-range.sh` on all four of the seasonal data files using `seasonal/*.csv` to match their names, and pipe its output to `sort` to see that your scripts can be used just like Unix's built-in commands.



TERMINAL

```
  GNU nano 2.5.3                File: date-range.sh

# Print the first and last date from each data file.
for filename in $@
do
    cut -d , -f 1 $filename | grep -v Date | sort | head -n 1
    cut -d , -f 1 $filename | grep -v Date | sort | tail -n 1
done
```



TERMINAL

```
$ nano date-range.sh
$ bash date-range.sh seasonal/*.csv
2017-01-05
2017-08-16
2017-01-25
2017-09-07
2017-01-11
2017-08-04
2017-01-03
2017-08-13
$ bash date-range.sh seasonal/*.csv | sort
2017-01-03
2017-01-05
2017-01-11
2017-01-25
2017-08-04
2017-08-13
2017-08-16
2017-09-07
```

What happens when I don't provide filenames?

A common mistake in shell scripts (and interactive commands) is to put filenames in the wrong place. If you type:

```
tail -n 3
```

then since `tail` hasn't been given any filenames, it waits to read input from your keyboard. This means that if you type:

```
head -n 5 | tail -n 3 somefile.txt
```

then `tail` goes ahead and prints the last three lines of `somefile.txt`, but `head` waits forever for keyboard input, since it wasn't given a filename and there isn't anything ahead of it in the pipeline.

Suppose you do accidentally type:

```
head -n 5 | tail -n 3 somefile.txt
```

What should you do next?

INSTRUCTIONS 50 XP

Possible Answers

- Wait 10 seconds for `head` to time out. press 1
- Type `somefile.txt` and press Enter to give `head` some input. press 2
- [✓ Use Ctrl-C to stop the running `head` program.] press 3