Data Engineer Technical Documentation

Table of Contents

[Introduction to Shell 3](#_Toc172818381)

[Manipulating files and directories 3](#_Toc172818382)

[How does the shell compare to a desktop interface? 4](#_Toc172818383)

[Where am I? 5](#_Toc172818384)

[How can I identify files and directories? 6](#_Toc172818385)

[How else can I identify files and directories? 7](#_Toc172818386)

[How can I move to another directory? 8](#_Toc172818387)

[How can I move up a directory? 9](#_Toc172818388)

[How can I copy files? 10](#_Toc172818389)

[How can I rename files? 11](#_Toc172818390)

[How can I delete files? 12](#_Toc172818391)

[How can I create and delete directories? 14](#_Toc172818392)

[Wrapping up 15](#_Toc172818393)

[Manipulating data 17](#_Toc172818394)

[How can I view a file's contents? 17](#_Toc172818395)

[How can I view a file's contents piece by piece? 18](#_Toc172818396)

[How can I look at the start of a file? 22](#_Toc172818397)

[How can I type less? 23](#_Toc172818398)

[How can I control what commands do? 25](#_Toc172818399)

[How can I list everything below a directory? 26](#_Toc172818400)

[How can I get help for a command? 27](#_Toc172818401)

[How can I select columns from a file? 29](#_Toc172818402)

[What can't cut do? 31](#_Toc172818403)

[How can I repeat commands? 32](#_Toc172818404)

[How can I select lines containing specific values? 35](#_Toc172818405)

[Why isn't it always safe to treat data as text? 37](#_Toc172818406)

[Combining tools 38](#_Toc172818407)

[How can I store a command's output in a file? 38](#_Toc172818408)

[How can I use a command's output as an input? 39](#_Toc172818409)

[What's a better way to combine commands? 40](#_Toc172818410)

[How can I combine many commands? 41](#_Toc172818411)

[How can I count the records in a file? 42](#_Toc172818412)

[How can I specify many files at once? 43](#_Toc172818413)

[What other wildcards can I use? 44](#_Toc172818414)

[How can I sort lines of text? 44](#_Toc172818415)

[How can I remove duplicate lines? 46](#_Toc172818416)

[How can I save the output of a pipe? 47](#_Toc172818417)

[How can I stop a running program? 48](#_Toc172818418)

[Wrapping up 48](#_Toc172818419)

[Batch processing 50](#_Toc172818420)

[How does the shell store information? 50](#_Toc172818421)

[How can I print a variable's value? 51](#_Toc172818422)

[How else does the shell store information? 52](#_Toc172818423)

[How can I repeat a command many times? 53](#_Toc172818424)

[How can I repeat a command once for each file? 54](#_Toc172818425)

[How can I record the names of a set of files? 55](#_Toc172818426)

[A variable's name versus its value 56](#_Toc172818427)

[How can I run many commands in a single loop? 57](#_Toc172818428)

[Why shouldn't I use spaces in filenames? 58](#_Toc172818429)

[How can I do many things in a single loop? 59](#_Toc172818430)

[Creating new tools 60](#_Toc172818431)

[How can I edit a file? 60](#_Toc172818432)

[How can I record what I just did? 61](#_Toc172818433)

[How can I save commands to re-run later? 62](#_Toc172818434)

[How can I re-use pipes? 64](#_Toc172818435)

[How can I pass filenames to scripts? 65](#_Toc172818436)

[How can I process a single argument? 67](#_Toc172818437)

[How can one shell script do many things? 67](#_Toc172818438)

[How can I write loops in a shell script? 69](#_Toc172818439)

[What happens when I don't provide filenames? 72](#_Toc172818440)

# Introduction to Shell

Description

The Unix command line has survived and thrived for almost 50 years because it lets people do complex things with just a few keystrokes. Sometimes called "the universal glue 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 Introduction will introduce the key elements and show how to use them efficiently.

## Manipulating files and directories

This is a brief introduction to the Unix shell. You'll learn why it is still in use after almost 50 years, how it compares to the graphical tools you may be more familiar with, how to move around in the shell, and how to create, modify, and delete files and folders.

### 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.

|  |
| --- |
| **Question:** What is the relationship between the graphical file explorer that most people use and the command-line shell?  **Answer:** They are both interfaces for issuing commands to the operating system.  **Note:** Both take the user's commands (whether typed or clicked) and send them to the operating system.  **Wrong answer:**   * The shell is part of the operating system, while the file explorer is separate. * The file explorer is built on top of the shell. * The file explorer lets you view and edit files, while the shell lets you run programs. |

### 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.

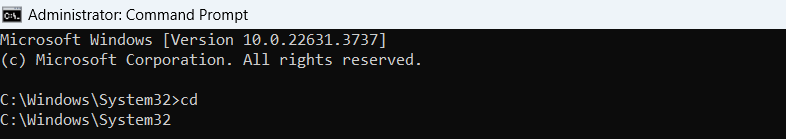
To find out where you are in the filesystem, run the command pwd (short for "**p**rint **w**orking **d**irectory"). This prints the absolute path of your **current working directory**, which is where the shell runs commands and looks for files by default.

Run pwd. Where are you right now?

**Note:** pwd is a linux command, won't work in windows. In windows just run cd without passing any arguments

**Linux:**

**Windows:**



### 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**).

**Question:** Use ls with an appropriate argument to list the files in the directory /home/repl/seasonal (which holds information on dental surgeries by date, broken down by season). Which of these files is not in that directory?

**Possible answers:**

autumn.csv

fall.csv - that file is not in the seasonal directory.

spring.csv

winter.csv

**Note:** Use the command dir to list all the directories and files in Windows; ls is a unix command.

### How else can I identify files and directories?

An absolute path is like a latitude and longitude: it has the same value 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".

As examples:

* If you are in the directory /home/repl, the **relative** path seasonal specifies the same directory as the **absolute** path /home/repl/seasonal.
* If you are in the directory /home/repl/seasonal, the **relative** path winter.csv specifies the same file as the **absolute** path /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. If it *does not* begin with /, it is relative.

**Question:** You are in /home/repl. Use ls with a **relative path** to list the file that has an absolute path of /home/repl/course.txt (and only that file).

**Answer:**



**Question:** You are in /home/repl. Use ls with a **relative** path to list the file /home/repl/seasonal/summer.csv (and only that file).

**Answer:**



**Question:** You are in /home/repl. Use ls with a **relative** path to list the contents of the directory /home/repl/people.

**Answer:**



### 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.

**Question:** You are in /home/repl/. Change directory to /home/repl/seasonal using a relative path.

**Answer:**

**Question:** Use pwd to check that you're there.

**Answer:**

**Question:** Use ls without any paths to see what's in that directory.

**Answer:** ****

### 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.

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

**Answer:** /home

**Note:** The path means 'home directory', 'up a level', 'here'.

**Wrong answer:**

* /home/repl
* /home/repl/seasonal
* / (the root directory)

### 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:

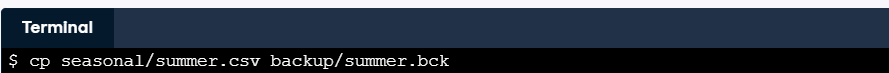
cp original.txt duplicate.txt

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

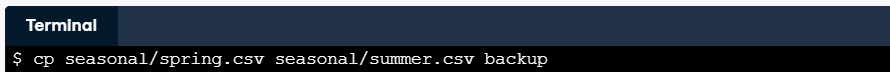
copies all of the files into that directory.

**Question:** Make a copy of seasonal/summer.csv in the backup directory (which is also in /home/repl), calling the new file summer.bck.

**Answer:**

**Question:** Copy spring.csv and summer.csv from the seasonal directory into the backup directory without changing your current working directory (/home/repl).

**Answer:**



**Note:** **By default**, there's no cp in Windows Command Prompt (cmd.exe). The equivalent cmd.exe command is copy. cp is a Unix command.

**Example:** copy D:\test1\test1.txt test2

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).

**Question:** You are in /home/repl, which has sub-directories seasonal and backup. Using a single command, move spring.csv and summer.csv from seasonal to backup.

**Answer:**



**Note:** **By default**, there's no mv in Windows Command Prompt (cmd.exe). The equivalent cmd.exe command is move. mv is a Unix command.

**Example:** move D:\test1\test1.txt test2

### 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.

**Question:** Go into the seasonal directory.

**Answer:**



**Question:** Rename the file winter.csv to be winter.csv.bck.

**Answer:**



**Question:** Run ls to check that everything has worked.

**Answer:**



**Example** **In Windows OS (operating system)**: move test1.txt test1.txt.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.

**Question:** You are in /home/repl. Go into the seasonal directory.

**Answer:** ****

**Question:** Remove autumn.csv.

**Answer:** 

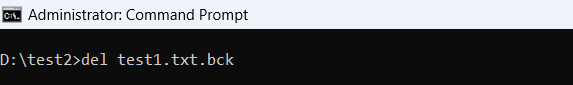
**Question:** Go back to your home directory.

**Answer:** ****

**Question:** Remove seasonal/summer.csv without changing directories again.

**Answer:** ****

**Note:** InWindows use del **Example:** del test1.txt.bck



### 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;)

**Question:** Without changing directories, delete the file agarwal.txt in the people directory.

**Answer:** ****

**Question:** Now that the people directory is empty, use a single command to delete it.

**Answer:** ****

**Note In Windows is the same:** rmdir test2



**Question:** Since a directory is not a file, you must use the command mkdir directory\_name to create a new (empty) directory. Use this command to create a new directory called yearly below your home directory.

**Answer:**



**Note In Windows is the same:** mkdir test2

**Question:** Now that yearly exists, create another directory called 2017 inside it without leaving your home directory.

**Answer:** ****

### 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.

**Question:** Use cd to go into /tmp.

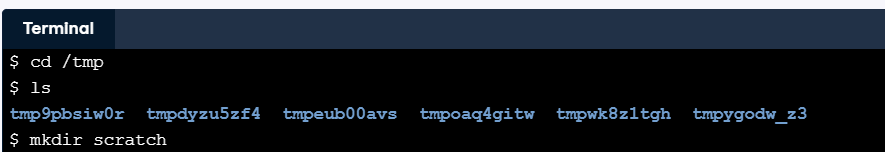
**Answer:** ****

**Question:** List the contents of /tmp without typing a directory name.

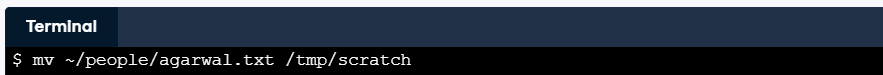
**Answer:** ****

**Question:** Make a new directory inside /tmp called scratch.

**Answer:**

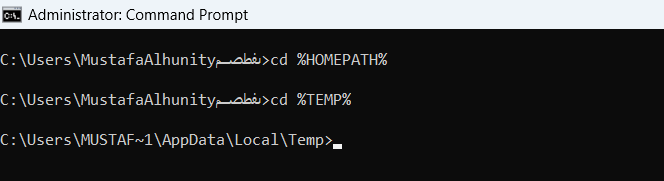


**Question:** Move /home/repl/people/agarwal.txt into /tmp/scratch. We suggest you use the ~ shortcut for your home directory and a relative path for the second rather than the absolute path.

**Answer:** ****

**Note In Windows OS is:** cd %TEMP%

**Note home directory in Windows OS is:** cd %HOMEPATH%



## Manipulating data

The commands you saw in the Manipulating files and directories allowed you to move things around in the filesystem. Here will show you how to work with the data in those files. The tools we’ll use are fairly simple, but are solid building blocks.

### 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", meaning "to link things together", since it will print all the files whose names you give it, one after the other.)

cat agarwal.txt

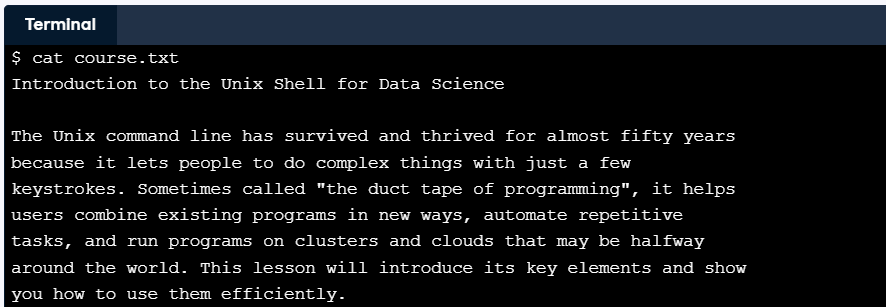
name: Agarwal, Jasmine

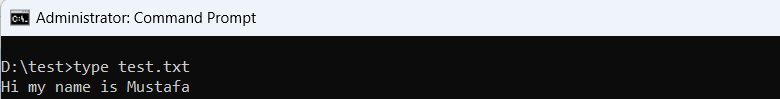
position: RCT2

start: 2017-04-01

benefits: full

**Question:** Print the contents of course.txt to the screen.

**Answer:** ****

**the Windows equivalent of the Unix command cat is type:**

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

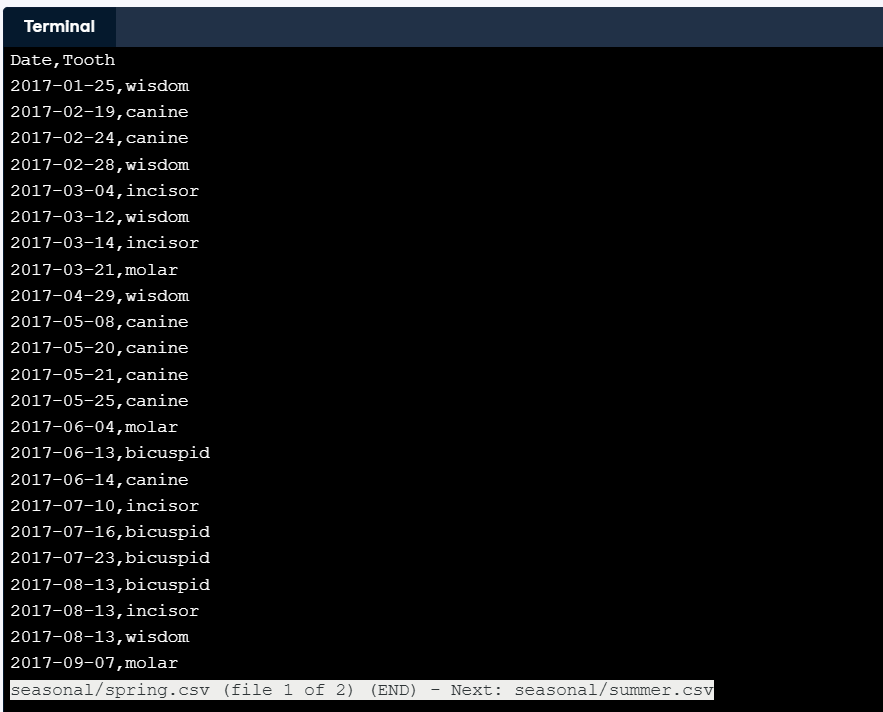
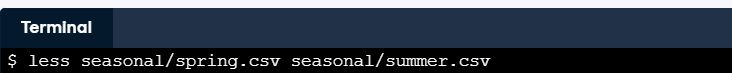
You can use cat to print large files and then scroll through the output, but it is usually more convenient to **page** the output. The original command for doing this was called more, but it has been superseded by a more powerful command called less. (This kind of naming is what passes for humor in the 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.

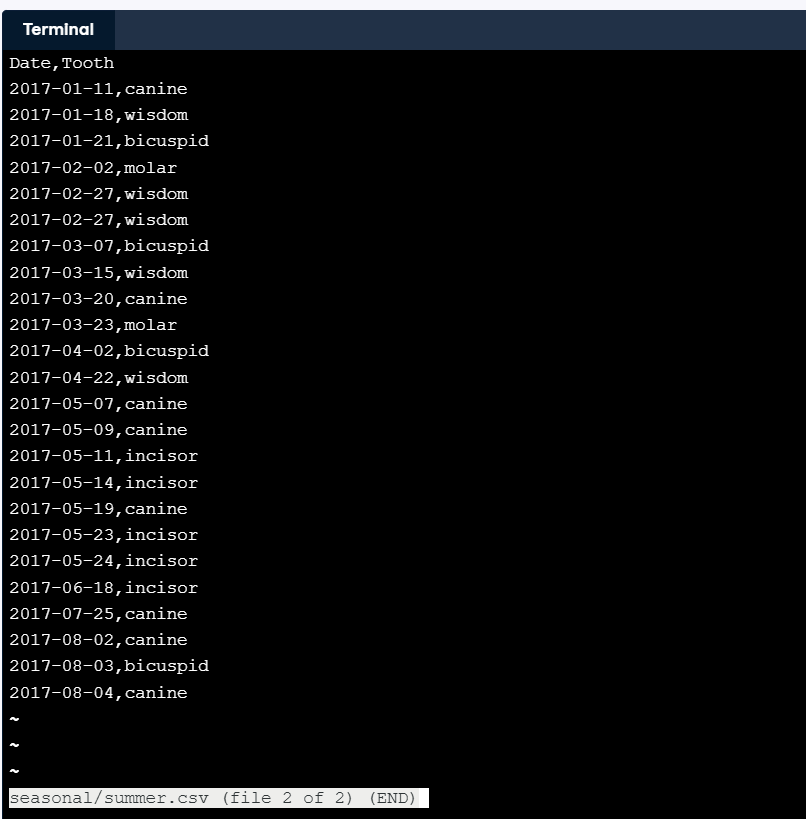
If you give less the names of several files, you can type :n (colon and a lower-case 'n') to move to the next file, :p to go back to the previous one, or :q to quit.

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.

**Question:** Use less seasonal/spring.csv seasonal/summer.csv to view those two files in that order. Press spacebar to page down, :n to go to the second file, and :q to quit.

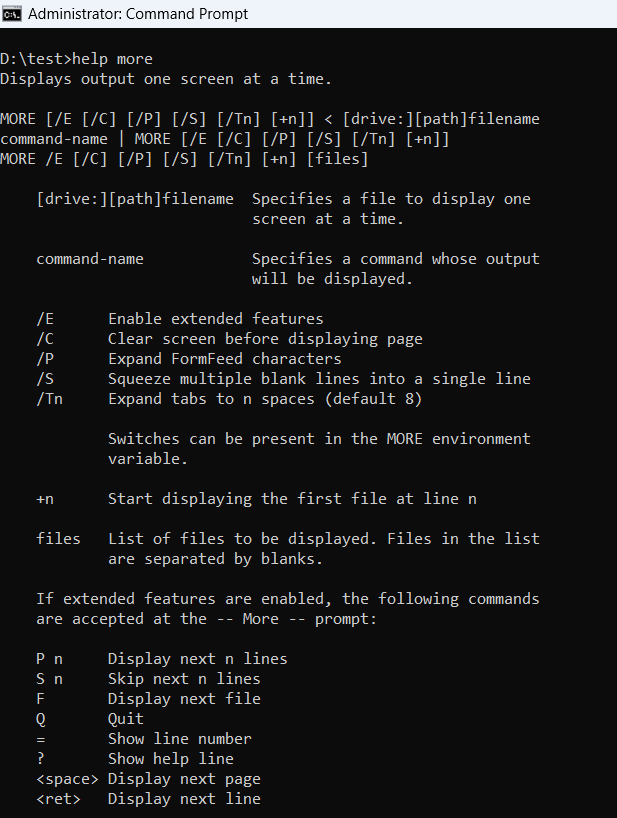
**Answer:**





**the Windows equivalent:**

**more** command



### 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

displays:

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

**Question:** What does head do if there aren't 10 lines in the file? (To find out, use it to look at the top of people/agarwal.txt.)

**Possible answer:**

Print an error message because the file is too short.

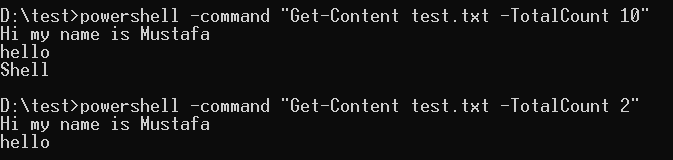
Display as many lines as there are.

Display enough blank lines to bring the total to 10.

****

**The Windows equivalent:**

powershell -command "Get-Content test.txt -TotalCount 10"

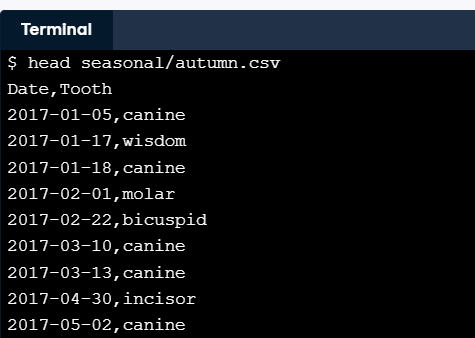


### How can I type less?

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.

**Question:** Run head seasonal/autumn.csv without typing the full filename.

**Answer:**

**Question:** Run head seasonal/spring.csv without typing the full filename.

**Answer:**

### 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

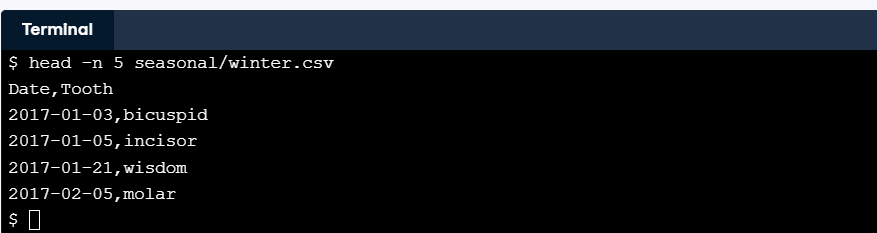
head will only display the first three lines of the file. If you run head -n 100, it will display the first 100 (assuming there are that many), and so on.

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 document, we only accept answers that do that.

**Question:** Display the first 5 lines of winter.csv in the seasonal directory.

**Answer:**



**Note:** With this technique, you can avoid your shell from blowing up if you want to have a look at larger text files.

### 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:

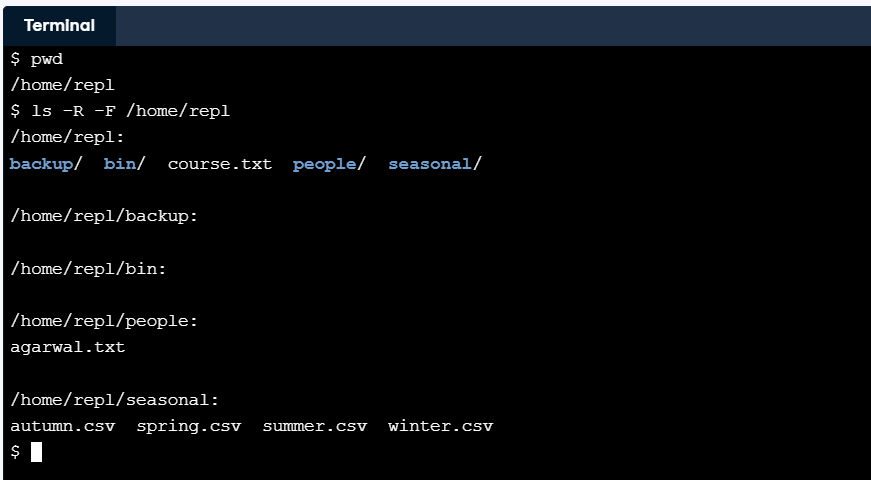
agarwal.txt

./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.

**Question:** 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.)

**Answer:**

### 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)

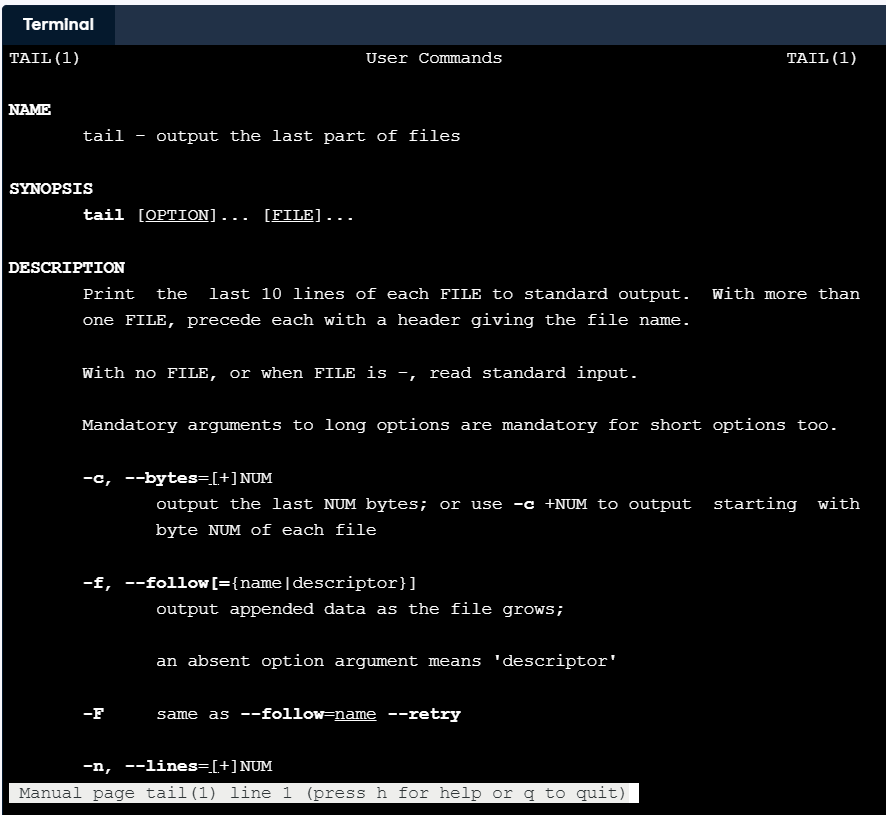
man automatically invokes less, so you may need to press spacebar to page through the information and :q to quit.

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**](https://stackoverflow.com/), ask a question on DataCamp's Slack channels, or look at the SEE ALSO sections of the commands you already know.

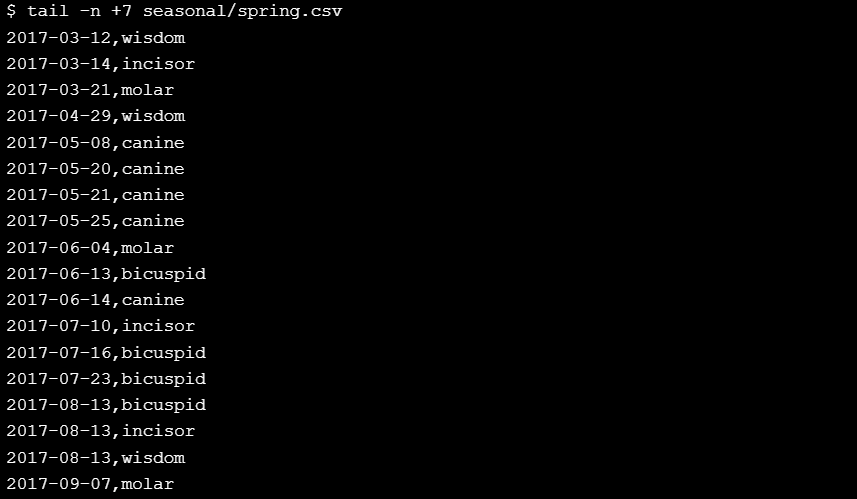
**Question:** Read the manual page for the tail command to find out what putting a + sign in front of the number used with the -n flag does. (Remember to press spacebar to page down and/or type q to quit.)

**Answer:**



**Question:** Use tail with the flag -n +7 to display all but the first six lines of seasonal/spring.csv.

**Answer:**



### 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.

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

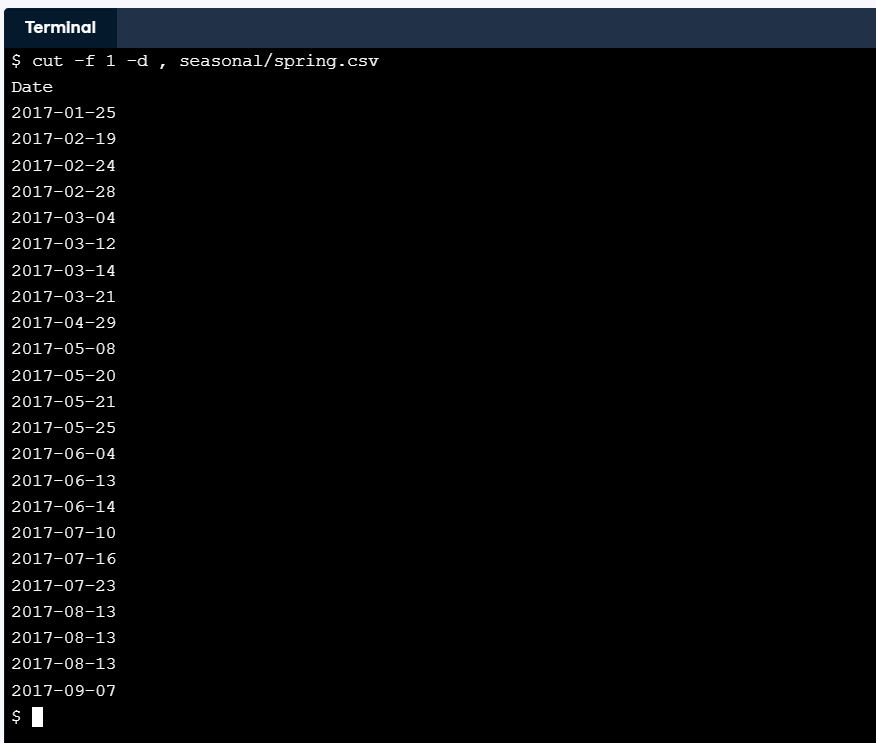
**Possible answer:**

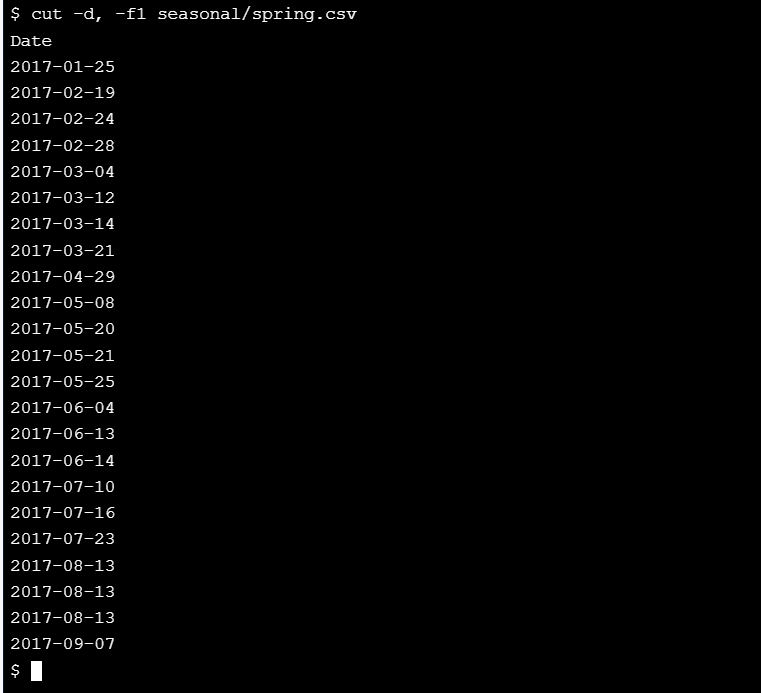
cut -d , -f 1 seasonal/spring.csv

cut -d, -f1 seasonal/spring.csv

Either of the above.

Neither of the above, because -f must come before -d.





**Note:** Adding a space after the flag is good style, but not compulsory.

### 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.

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

first:second:third:

(Note the trailing colon.)

**Possible answer:**

second

second:third

second:third:

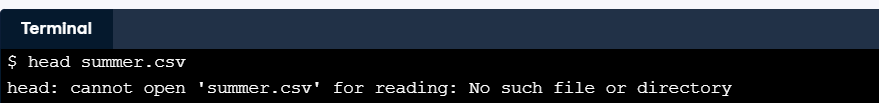
Neither of the above, because there aren’t four fields.

### 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.

**Question:** Run head summer.csv in your home directory (which should fail).

**Answer:**

**Question:** Change directory to seasonal.

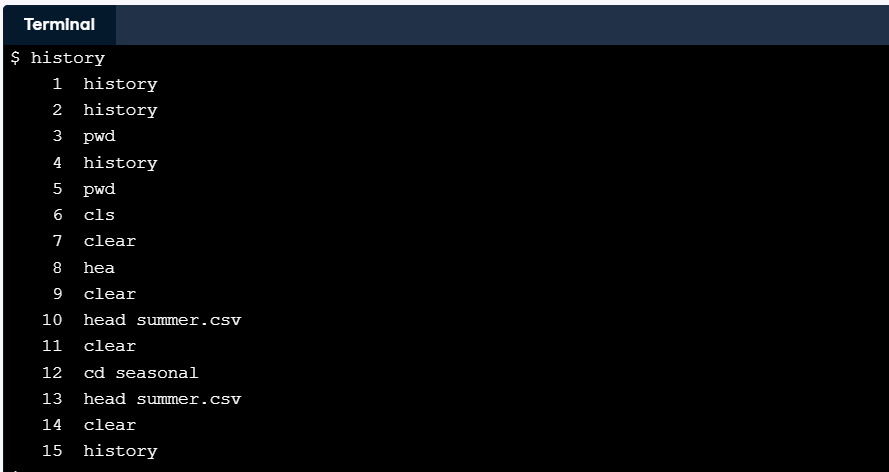
**Answer:**

**Question:** Re-run the head command with !head.

**Answer:**

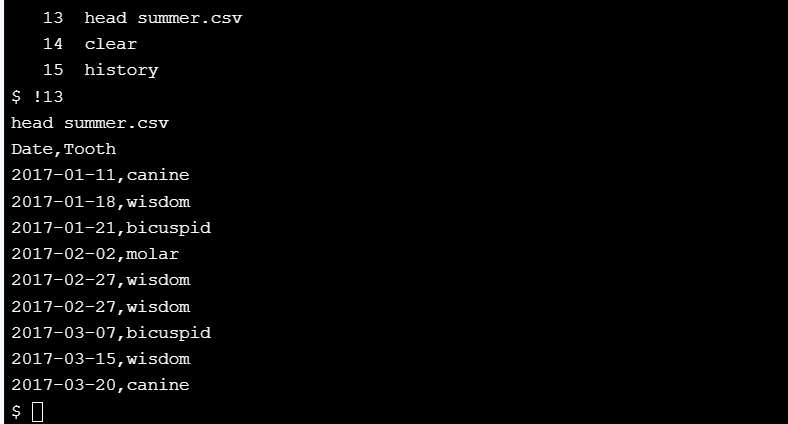


**Question:** Use history to look at what you have done.

**Answer:**

**Question:** Re-run head again using ! followed by a command number.

**Answer:**



### 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 **Combining tools**. What's more important right now is some of grep's more common flags:

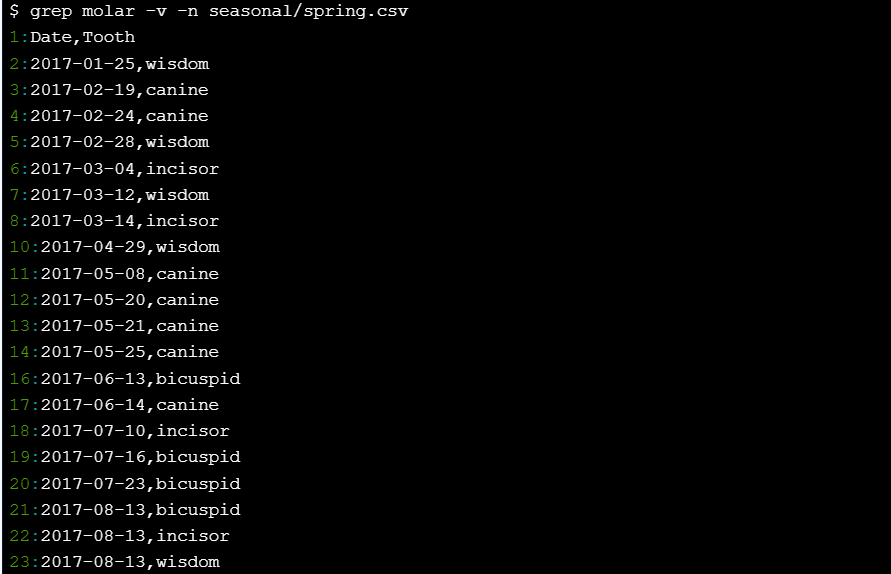
* -c: print a count of matching lines rather than the lines themselves
* -h: do *not* print the names of files when searching multiple files
* -i: ignore case (e.g., treat "Regression" and "regression" as matches)
* -l: print the names of files that contain matches, not the matches
* -n: print line numbers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

**Question:** Print the contents of all of the lines containing the word molar in seasonal/autumn.csv by running a single command while in your home directory. Don't use any flags.

**Answer:**

**Question:** Invert the match to 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".

**Answer:**



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

**Answer:**



### 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.

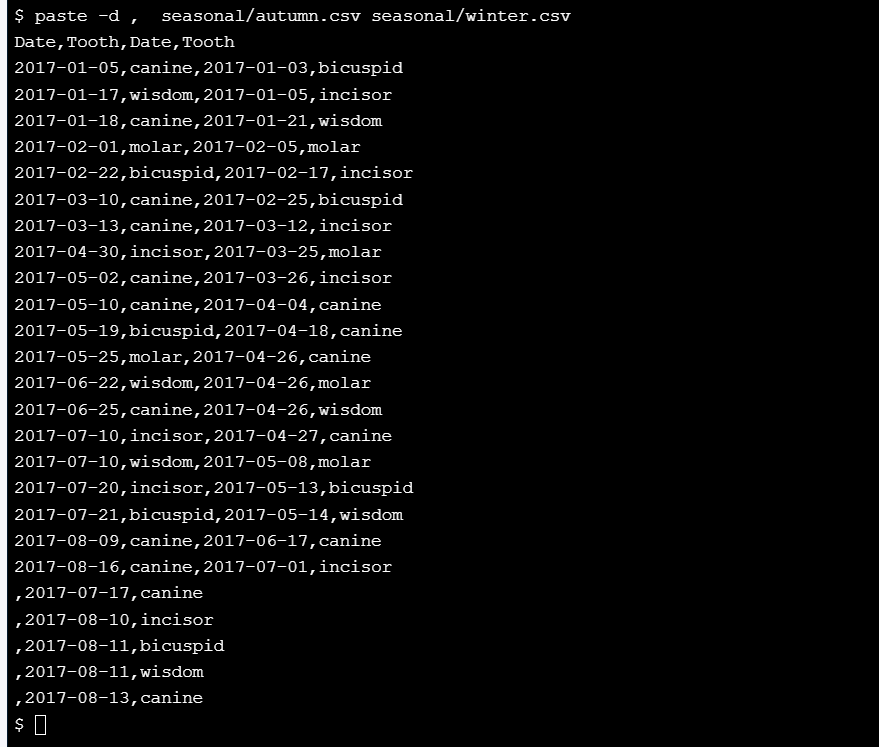
**Question:** Read the manual page for paste, and then run paste to combine the autumn and winter data files in a single table using a comma as a separator. What's wrong with the output from a data analysis point of view?

**Possible answer:**

The column headers are repeated.

The last few rows have the wrong number of columns.

Some of the data from winter.csv is missing.



**Note:** joining the lines with columns creates only one empty column at the start, not two.

## Combining tools

### 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

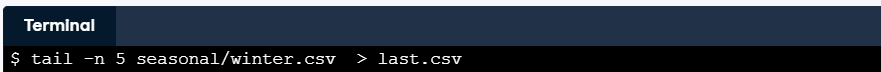
nothing appears on the screen. Instead, head's output is put in a new file called top.csv. You can take a look at that file's contents using cat:

cat top.csv

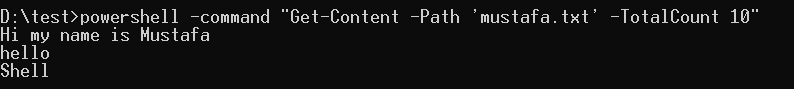
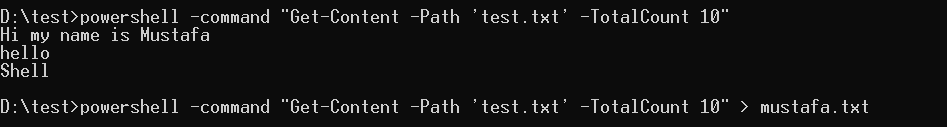
The greater-than sign > tells the shell to redirect head's output to a file. It isn't part of the head command; instead, it works with every shell command that produces output.

**Question:** Combine tail with redirection to save the last 5 lines of seasonal/winter.csv in a file called last.csv.

**Answer:**



**the Windows equivalent:**



### 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.

**Question:** Select the last two lines from seasonal/winter.csv and save them in a file called bottom.csv.

**Answer:**

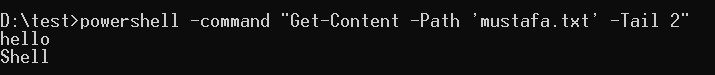


**Question:** Select the first line from bottom.csv in order to get the second-to-last line of the original file.

**Answer:**



**the Windows equivalent:**



### 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.

**Question:** Use cut to select all of the tooth names from column 2 of the comma delimited file seasonal/summer.csv, then pipe the result to grep, with an inverted match, to exclude the header line containing the word "Tooth". *cut* and *grep* were covered in detail in **Manipulating data**.

**Answer:**



### 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

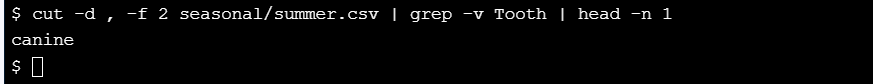
will:

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.

**Question:** In the previous exercise, you used the following command to select all the tooth names from column 2 of seasonal/summer.csv:

cut -d , -f 2 seasonal/summer.csv | grep -v Tooth

Extend this pipeline with a head command to only select the very first tooth name.

**Answer:**

**Note:** By chaining several commands together, you can build powerful data manipulation pipelines.

### How can I count the records in a file?

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

**Question:** Count how many records in seasonal/spring.csv have dates in July 2017 (2017-07).

* To do this, use grep with a partial date to select the lines and pipe this result into wc with an appropriate flag to count the lines.

**Answer:**

**Note:** Careful counting! Determining how much data you have is a great first step in any data analysis.

### 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.csv

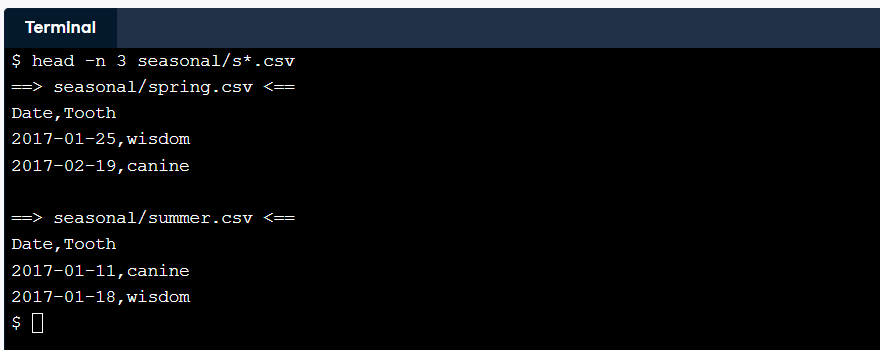
But typing the names of many files over and over is a bad idea: it wastes time, and sooner or later you will either leave a file out or repeat a file's name. To make your life better, the shell allows you to use **wildcards** to specify a list of files with a single expression. The most common wildcard 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

**Question:** 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.

**Answer:**

**Note:** This becomes even more important if your directory contains hundreds or thousands of files.

### 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.

grep -e "^[0-9]\{3\}\-[0-9]\{3\}\-[0-9]\{4\}$" -e "^([0-9]\{3\}) [0-9]\{3\}\-[0-9]\{4\}$" file.txt

this is a grep command accepting two regular expressions  
1 .^[0-9]\{3\}\-[0-9]\{3\}\-[0-9]\{4\}$  
2 .^([0-9]\{3\}) [0-9]\{3\}\-[0-9]\{4\}$

The construction is as follows

* ^: indicates the starting of the string
* $: indicates the end of the string
* [0-9]\{3\} : represent 3 numbers (\{3\}) between the range 0-9 ([0-9] a digit in the range)
* \: suppresses the specialness of the character
* -e: to include multiple regex

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

**Possible answer:**

[sj]\*.{.pdf, .txt}

{s\*.pdf, j\*.txt}

[singh,johel]{\*.pdf, \*.txt}

{singh.pdf, j\*.txt}

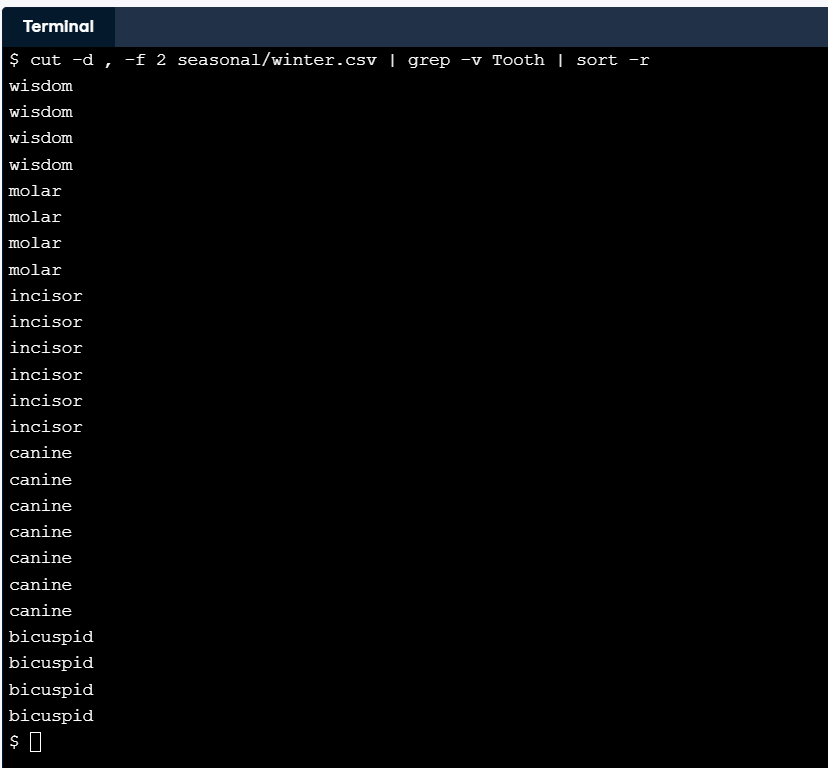
### How can I sort lines of text?

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 **f**old 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.

**Question:** Remember the combination of cut and grep to select all the tooth names from column 2 of seasonal/summer.csv?

cut -d , -f 2 seasonal/summer.csv | grep -v Tooth

Starting from this recipe, sort the names of the teeth in seasonal/winter.csv (not summer.csv) in descending alphabetical order. To do this, extend the pipeline with a sort step.

**Answer:**

sort has many uses. For example, piping sort -n to head shows you the largest values.

### 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.

**Question:** Write a pipeline to:

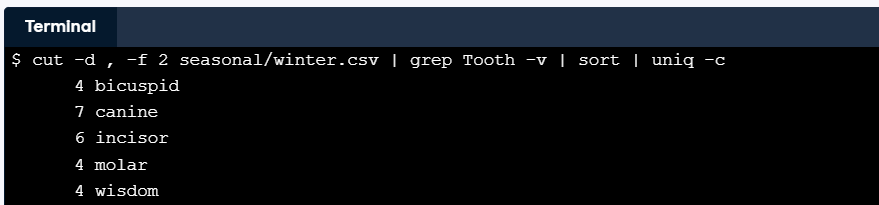
* get the second column from seasonal/winter.csv,
* remove the word "Tooth" from the output so that only tooth names are displayed,
* sort the output so that all occurrences of a particular tooth name are adjacent; and
* display each tooth name once along with a count of how often it occurs.

The start of your pipeline is the same as the previous exercise:

cut -d , -f 2 seasonal/winter.csv | grep -v Tooth

Extend it with a sort command, and use uniq -c to display unique lines with a count of how often each occurs rather than using uniq and wc.

**Answer:**



After all of this work on a pipe, it would be nice if we could store the result, no? 😊

### 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.

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

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

**Possible answer:**

The command's output is redirected to the file as usual.

The shell reports it as an error.

The shell waits for input forever.

### 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.

**Question:**

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.

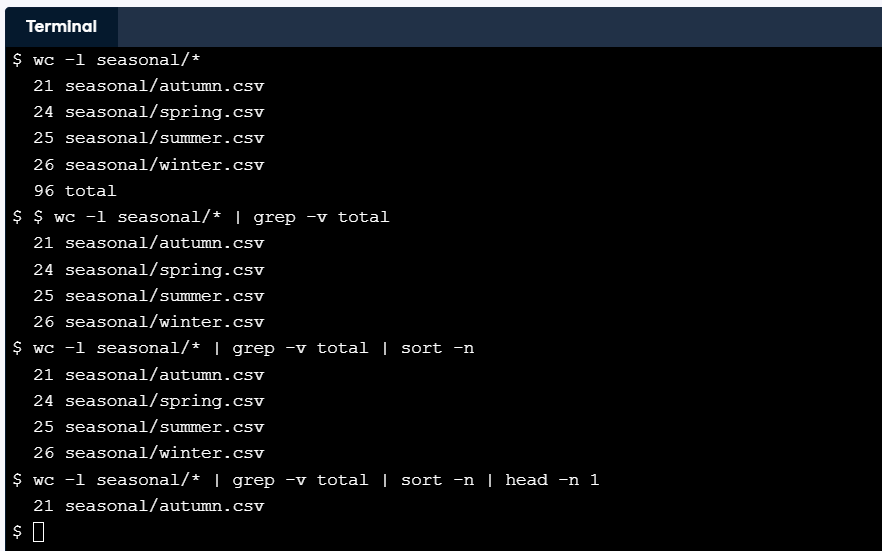
**Answer:**

### 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.

**Question:**

* 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.)
* 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 -n and head -n 1 to find the file containing the fewest lines.

**Answer:**

It turns out autumn.csv is the file with the fewest lines.

## Batch processing

### 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.

**Question:** 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?

**Possible answer:**

10

500

2000

The variable is not there.



### 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. Typing

echo hello Shell!

prints

hello Shell!

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

echo USER

it will print the variable's name, USER.

To get the variable's value, you must put a dollar sign $ in front of it. Typing

echo $USER

prints

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".)

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

**Answer:**



**The Windows equivalent:**



### 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

**Question:**

* Define a variable called testing with the value seasonal/winter.csv.
* 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.

**Answer:**

### 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 filetype in gif jpg png; do echo $filetype; done

it produces:

gif

jpg

png

Notice these things about the loop:

1. The structure is 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, filetype).
4. The body of the loop that does the processing (in our case, echo $filetype).

Notice that the body uses $filetype to get the variable's value instead of just filetype, 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.

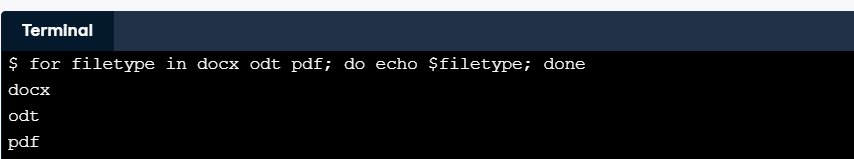
**Question:** Modify the loop so that it prints:

docx

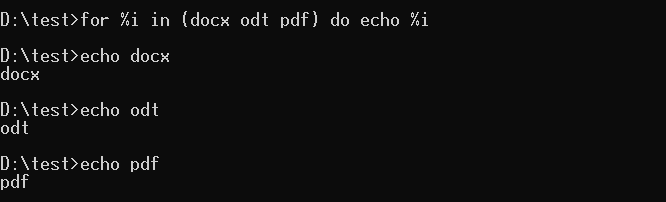
odt

pdf

Please use filetype as the name of the loop variable.

**Answer:**

**The Windows equivalent:**



**Note:** Loops are brilliant if you want to do the same thing hundreds or thousands of times.

### 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

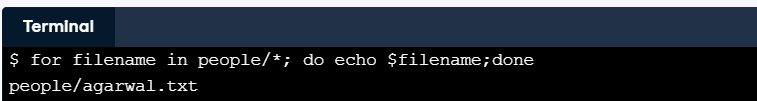
seasonal/summer.csv

seasonal/winter.csv

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

**Question:** 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.

**Answer:**



Wildcards and loops make a powerful combination.

### 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.

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

files=seasonal/\*.csv

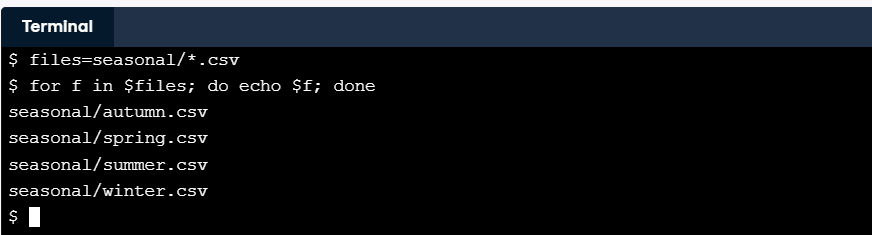
for f in $files; do echo $f; done

**Possible answer:**

None: since files is defined on a separate line, it has no value in the second line.

One: the word "files".

Four: the names of all four seasonal data files.



for f in seasonal/\*.csv; do echo $f; done

### A variable's name versus its value

A common mistake is to forget to use $ before the name of a variable. When you do this, the shell uses the name you have typed rather than the value of that variable.

A more common mistake for experienced users is to mis-type the variable's name. For example, if you define datasets like this:

datasets=seasonal/\*.csv

and then type:

echo $datsets

the shell doesn't print anything, because datsets (without the second "a") isn't defined.

**Question:** If you were to run these two commands in your home directory, what output would be printed?

files=seasonal/\*.csv

for f in files; do echo $f; done

(Read the first part of the loop carefully before answering.)

**Possible answer:**

One line: the word "files".

Four lines: the names of all four seasonal data files.

Four blank lines: the variable f isn't assigned a value.

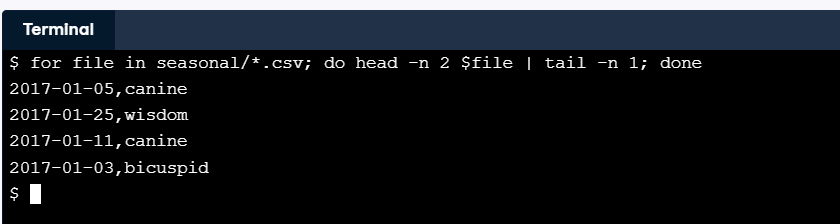
The loop uses **files** instead of **$files**, so the list consists of the word "files".

### 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.

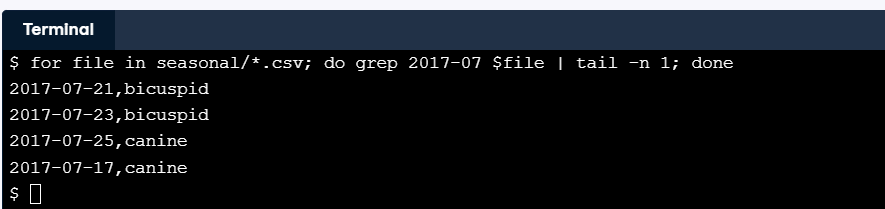


**Question:** Write a loop that prints the last entry from July 2017 (2017-07) in every seasonal file. It should produce a similar output to:

grep 2017-07 seasonal/winter.csv | tail -n 1

but for **each** seasonal file separately. Please use file as the name of the loop variable, and remember to loop through the list of files seasonal/\*.csv (instead of 'seasonal/winter.csv' as in the example).

**Answer:**



### 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:

**Possible answer:**

The shell will print an error message because last and year.csv do not exist.

The shell will delete current.csv.

Both of the above.

Nothing.

**Note:** You can use single quotes, ', or double quotes, ", around the file names.

### 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

**Question:** 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?

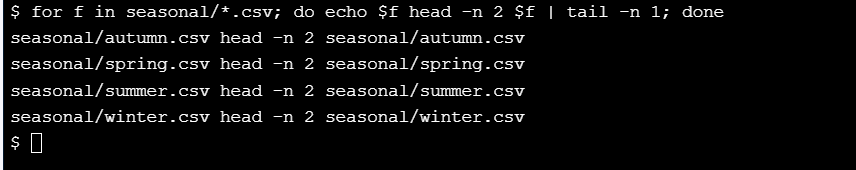
**Possible answer:**

Print an error message.

Print one line for each of the four files.

Print one line for autumn.csv (the first file).

Print the last line of each file.



echo produces one line that includes the filename twice, which tail then copies.

## Creating new tools

### 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'). *You will also need to press Enter to confirm the filename!*
* Ctrl + X: exit the editor.

**Question:**

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

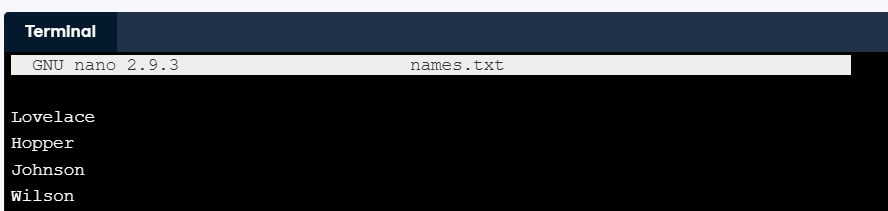
Lovelace

Hopper

Johnson

Wilson

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

**Answer:**

### How can I record what I just did?

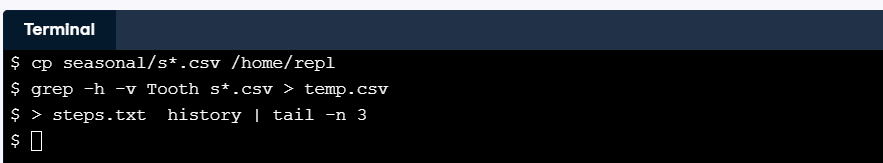
When you are doing a complex analysis, you will often want to keep a record of the commands you used. You can do this with the tools you have already seen:

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.

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.

**Question:**

* 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.
* 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.)

**Answer:**

### 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.

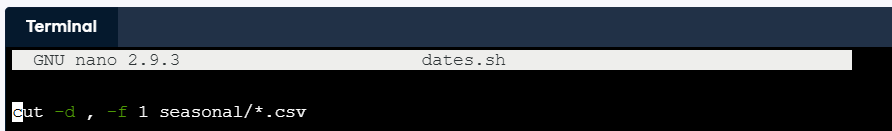
**Question:**

Use nano dates.sh to create a file called dates.sh that contains this command:

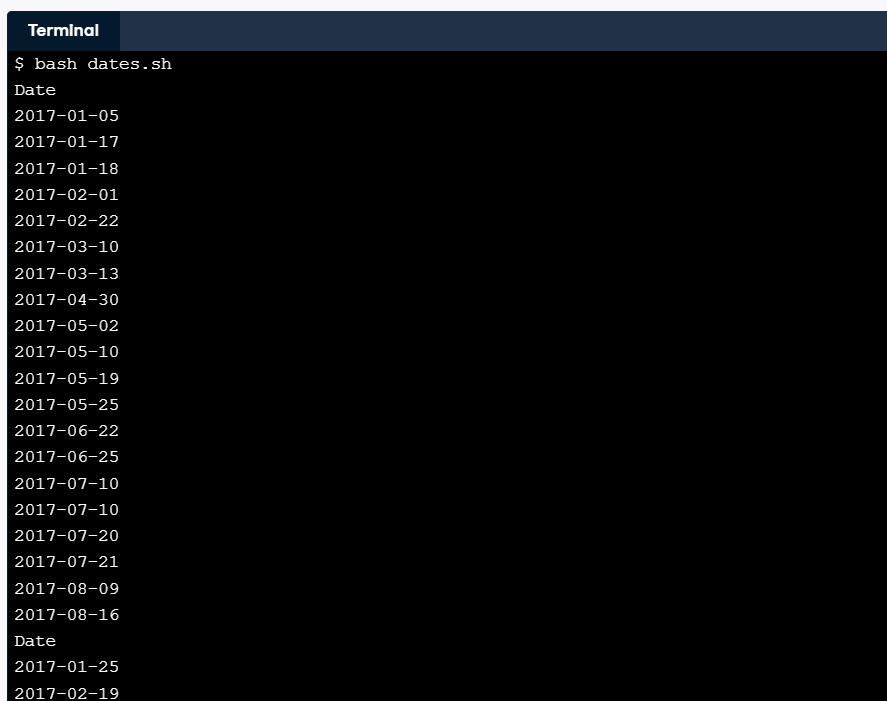
cut -d , -f 1 seasonal/\*.csv

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

**Answer:**



**Question:**   
Use bash to run the file dates.sh.

**Answer:**

### 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 can also contain pipes. For example, if all-dates.sh contains this line:

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

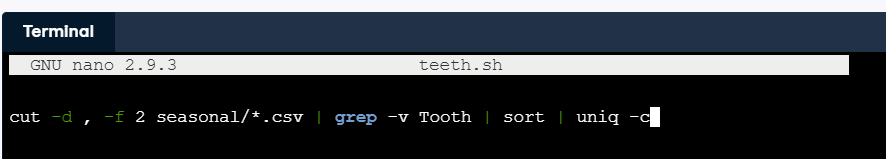
then:

bash all-dates.sh > dates.out

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

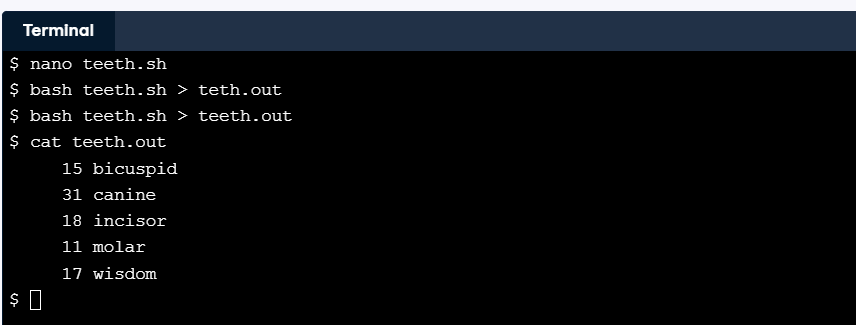
**Question:**

A file teeth.sh in your home directory has been prepared for you, but contains some blanks. Use Nano to edit the file 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.

**Answer:**

**Question:** Use bash to run teeth.sh and > to redirect its output to teeth.out.

**Answer:**

**Question:** Run cat teeth.out to inspect your results.

**Answer:**

This all may feel contrived at first, but the nice thing is that you are automating parts of your workflow step by step. Something that comes in really handy as a data scientist!

### 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 sort $@ | uniq, 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:

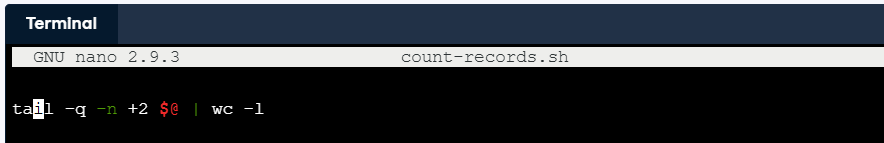
bash unique-lines.sh seasonal/summer.csv seasonal/autumn.csv

it processes two data files, and so on.

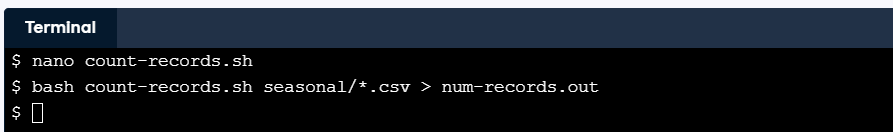
As a reminder, to save what you have written in Nano, type *Ctrl* + *O* to write the file out, then Enter to confirm the filename, then *Ctrl* + *X* to exit the editor.

**Question:**   
Edit the script count-records.sh with Nano and fill in the two \_\_\_\_ placeholders with $@ and -l (the letter) respectively so that it counts the number of lines in one or more files, excluding the first line of each.

**Answer:**



**Question:**   
Run count-records.sh on seasonal/\*.csv and redirect the output to num-records.out using >.

**Answer:**

### 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.

**Question:** 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?

**Possible answer:**

head -n $1 $2 | tail -n 1 | cut -d , -f $3

head -n $2 $1 | tail -n 1 | cut -d , -f $3

head -n $3 $1 | tail -n 1 | cut -d , -f $2

head -n $2 $3 | tail -n 1 | cut -d , -f $1

### How can one shell script do many things?

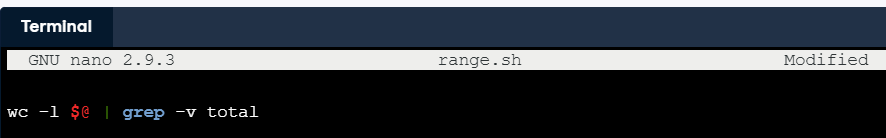
Our shells scripts so far have had a single command or pipe, but a script can contain many lines of commands. For example, you can create one that tells you how many records are in the shortest and longest of your data files, i.e., the range of your datasets' lengths.

Note that in Nano, "copy and paste" is achieved by navigating to the line you want to copy, pressing CTRL + K to cut the line, then CTRL + U twice to paste two copies of it.

As a reminder, to save what you have written in Nano, type *Ctrl* + *O* to write the file out, then Enter to confirm the filename, then *Ctrl* + *X* to exit the editor.

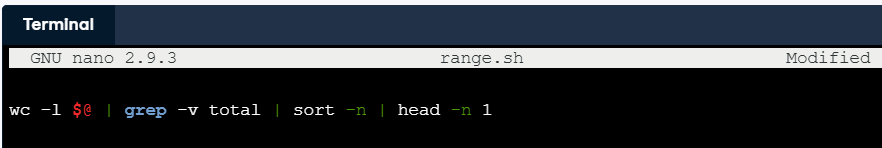
**Question:** Use Nano to edit the script range.sh and replace the two \_\_\_\_ placeholders with $@ and -v so that it lists the names and number of lines in all of the files given on the command line without showing the total number of lines in all files. (Do not try to subtract the column header lines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

**Answer:**

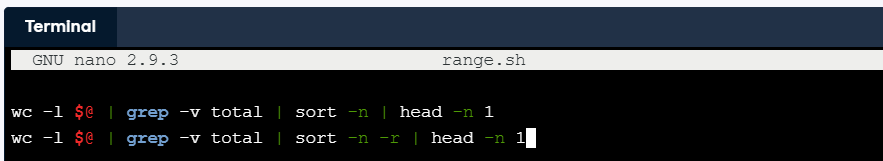


**Question:** Use Nano again to add sort -n and head -n 1 in that order to the pipeline in range.sh to display the name and line count of the shortest file given to it.

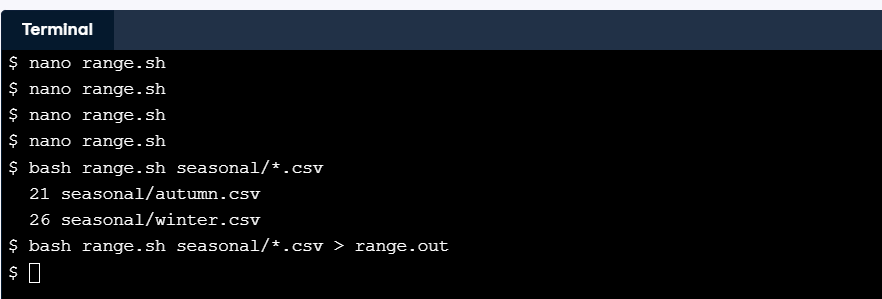
**Answer:**



**Question:** Again using Nano, add a second line to range.sh to print the name and record count of the longest file in the directory as well as the shortest. This line should be a duplicate of the one you have already written, but with sort -n -r rather than sort -n.

**Answer:**

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

**Answer:**

### 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

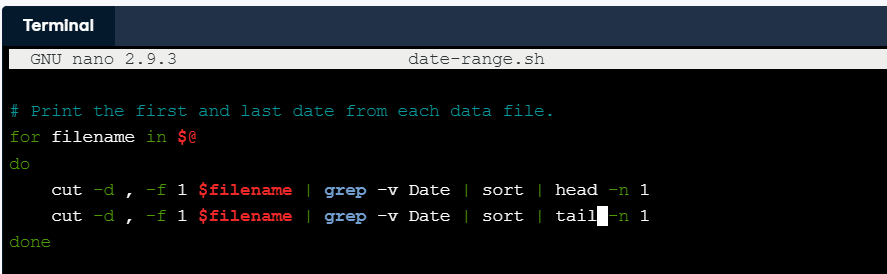
(You don't have to indent the commands inside the loop, but doing so makes things clearer.)

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.

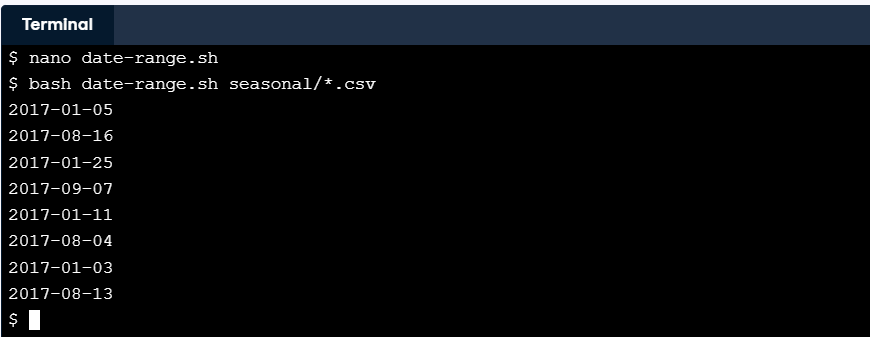
As a reminder, to save what you have written in Nano, type *Ctrl* + *O* to write the file out, then Enter to confirm the filename, then *Ctrl* + *X* to exit the editor.

**Question:** 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.

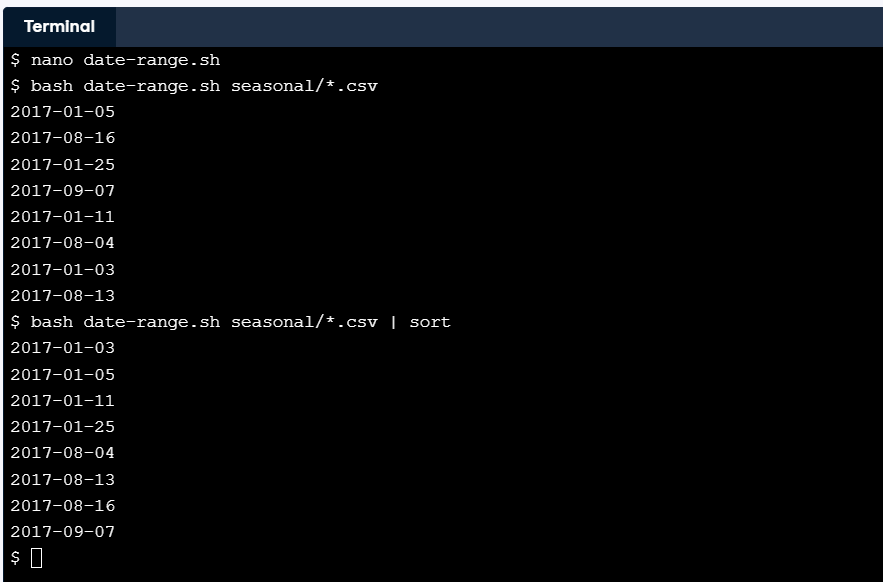
**Answer:**



**Question:** Run date-range.sh on all four of the seasonal data files using seasonal/\*.csv to match their names.

**Answer:**

**Question:** 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.

**Answer:**

### 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.

**Question:** Suppose you do accidentally type:

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

What should you do next?

**Possible answers**

Wait 10 seconds for head to time out.

Type somefile.txt and press Enter to give head some input.

Use Ctrl + C to stop the running head program.

**Note:** You should use Ctrl + C to stop a running program.