**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.

# 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.

**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 "**l**i**s**ting") 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**).

**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.

# 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.

**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.

**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.

# 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.

# 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.

# 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.)

# 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.

In Unix-like operating systems, including Linux, there is a common convention to have a directory called "/tmp" (short for "temporary") at the root level of the file system. This directory is used to store temporary files and directories that are required by various applications and processes.

The "/tmp" directory is typically writable by all users, meaning any user can create temporary files or directories in this location. It provides a space for programs to store data that is only needed temporarily and is not intended to persist across system reboots.

It's important to note that the "/tmp" directory is not present within every other directory in the file system; it exists only once at the root level. However, its purpose is to be accessible system-wide, allowing applications and users from different directories to use it for temporary storage.

For example, if you were to refer to the "/tmp" directory from any location in the file system, you would use the absolute path "/tmp". Likewise, if a program or script references "/tmp" as a temporary storage location, it would always point to the same directory, regardless of the current working directory.

In summary, while the "/tmp" directory is not present in every directory in the file system, it is a system-wide directory located at the root level, serving as a common location for temporary file storage, which is the same level as home.

# 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

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.

**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

If the number of rows are lesser than 10, it displays as many lines as there are.

# 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.

# 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.

Putting a + sign in front of the number used with the -n flag simply implies displaying lines from a specific line number.

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

# 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.

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.)

-F : This option stands for "classify." When used with ls, it appends special characters to the end of each listed entry to indicate the type of the file or directory. Here are the commonly used indicators:

* / : Denotes directories. For example, if a directory is named "Documents," it will be displayed as "Documents/".
* \* : Denotes executable files.
* @ : Denotes symbolic links.
* = : Denotes sockets.
* | : Denotes FIFOs (named pipes).

There are also flags like -a and -l.

* -a: This flag stands for "all" and tells the ls command to show all files, including hidden files and directories. By default, the ls command does not display hidden files (those that start with a dot .).
* -l: This flag stands for "long format" and tells the ls command to display detailed information about the files and directories in a long list format. This detailed information includes file permissions, owner, group, size, modification date, and filename.

# 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, look at the SEE ALSO sections of the commands you already know.

# 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.

**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. Order of flag does not matter. Spaces too between the flag and numbers, but it is good practise to do so.

# 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. Note the output of cut -d : -f 2-4 on the line:

first:second:third:

Output is:

:second:third:

Due to the trailing colon, which produces a 4th field.

# 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.

**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
* -E: exact match, similar to egrep test.txt | ‘Sam|Tom’/winter.csv need the quotes to differentiate the or from pipe operator

For example, grep bicuspid seasonal/winter.csv seasonal/spring.csv prints lines from both  winter.csv and spring.csv  that contain "bicuspid".

**Introduction to text processing tool**

sed stands for "stream editor." It's a powerful and versatile text processing tool found in Unix-like operating systems (including Linux and macOS). sed is primarily used for performing text transformations on an input stream (a file or input provided through a pipeline). It's commonly used for search, replace, insertion, deletion, and other text manipulation tasks.

**Basic Syntax:**

The basic syntax of the sed command is: sed options <commands> inputfile

* **options**: These are optional flags that modify the behavior of the sed command.
* **'commands'**: This is a sequence of sed commands enclosed in single quotes.
* **inputfile**: This is the file on which you want to perform text operations. If not provided, sed reads from standard input.

**Common Use Cases:**

Here are some common use cases of the sed command:

1. **Search and Replace:**
   * sed ‘s/pattern/replacement/’ inputfile:Replaces the first occurrence of **pattern** with **replacement** in the input file.
   * sed ‘s/pattern/replacement/g’ inputfile:Replaces all occurrences of **pattern** with **replacement** in the input file.
2. **Print Specific Lines:**
   * sed -n ‘5p’ inputfile: Prints the 5th line of the input file.
   * sed -n ‘2,5p’ inputfile: Prints lines 2 to 5 of the input file.
3. **Delete Lines:**
   * sed ‘2d’ inputfile: Deletes the 2nd line from the input file.
   * sed ‘2,5d’ inputfile: Deletes lines 2 to 5 from the input file.
4. **Insert and Append:**
   * sed ‘2i\New line’ inputfile: Inserts "New line" before the 2nd line.
   * sed ‘$a\End of file’ inputfile: Appends "End of file" after the last line.
5. **Using Regular Expressions:**
   * sed ‘pattern/d’ inputfile **sed '/pattern/d' inputfile**: Deletes lines containing **pattern**.
   * sed ‘pattern/s/old/new/’ inputfile: Replaces word **old** with word **new** in lines containing **pattern**.

**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.

Running  paste -d , to combine the autumn and winter data files in a single table using a comma as a separator. The output is wrong as the last few rows have wrong number of columns due to different number of rows.

# 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.

# 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.

**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.

**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.

# 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.

* -c: This option tells wc to display the character count of the file. It counts the number of characters present in the file, including letters, spaces, punctuation, and newline characters.
* -w: This option tells wc to display the word count of the file. It counts the number of words in the file. Words are typically separated by spaces or other whitespace characters.
* -l: This option tells wc to display the line count of the file. It counts the number of lines in the file. A line is considered as a sequence of characters terminated by a newline character ('\n').

# 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". Even s\*.csv is allowed. Using it, we can shorten the cut command above to this:

cut -d , -f 1 seasonal/\*

or:

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

**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.

# 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.

# 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.

# 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. Also uniq -c displays unique lines with a count of how often each occurs rather than using uniq and wc.

**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.

However, putting a redirection at the front of a pipeline as in:

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

The output will just be redirected to the file as usual

# 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.

**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.

# 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 DataCamp!

prints

hello DataCamp!

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".) More examples include

echo $OSTYPE

# 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, or display the first line:

echo $training

seasonal/summer.csv

head -n 1 $training

Date,Tooth

**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.

The done keyword marks the end of the loop block and informs the shell that the commands between do and done should be executed repeatedly for each value in the list (gif, jpg, and png in this case).

# 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.

**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.

# 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.

**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.

The below is a correct example:

files=seasonal/\*.csv

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

# 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.

# 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 (both single and double allowed) the files' names so that the shell treats each one as a single parameter:

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

# 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

**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.

**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 problem

# 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.

# 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.

# 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.

# 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.

For example count-records.sh

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

Above is a shell command that counts the number of lines in the input after skipping the first line. The -q flag tells tail to suppress the header when reading multiple files.

bash count-records.sh seasonal/\*.csv > num-records.out

Hence the above will output the total number of lines in all files combined, after skipping the first line in each file.

# 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.

# 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.

# 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.

# 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.

# 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.

Use Ctrl + C  to stop the running.

# Understanding the Shebang Line in Bash Scripts

Introduction: The shebang line (**#!/bin/bash**) at the beginning of a bash script plays a crucial role in determining how the script is executed. This guide delves into the significance of the shebang line, its variations, and its impact on different operating systems.

1. **Shebang Line Explained:**
   * Definition: The shebang line is the first line of a script that specifies the interpreter to execute the script.
   * Format: **#!/path/to/interpreter**.
2. **Functionality on Unix-like Systems (Linux, macOS):**
   * Purpose: Specifies the path to the desired interpreter (e.g., Bash) for running the script.
   * Examples:
     + **#!/bin/bash** for the default Bash interpreter, to identify, run **which bash**
     + **#!/usr/bin/env python3** for Python scripts.
   * Execution: The system uses the interpreter specified in the shebang line to process the script.
3. **Windows Considerations:**
   * Default Behavior: Windows Command Prompt doesn't directly interpret shebang lines.
   * Solutions:
     + Windows Subsystem for Linux (WSL):
       - Allows running Linux environment on Windows.
       - Supports shebang lines and bash scripts.
     + Git Bash, Cygwin, Third-party Bash Interpreters:
       - Enable running bash scripts on Windows.
4. **Advantages of Using Shebang Lines:**
   * Clarity: Clearly indicates which interpreter the script requires.
   * Portability: Ensures the script works consistently across different systems.
   * Ease of Use: Allows running scripts directly without specifying the interpreter.
5. **Best Practices:**
   * Include Shebang Line: Although not always mandatory on Unix-like systems, it's recommended for clarity and portability.
   * Specificity: Provide the full path to the interpreter to avoid ambiguity.
   * Documentation: Include comments to explain the script's purpose and any interpreter-specific usage.

Conclusion: The shebang line is a crucial component of bash scripts that defines the interpreter used to execute the script. While its primary role is on Unix-like systems, workarounds exist for running bash scripts on Windows. By understanding and utilizing the shebang line effectively, you enhance the readability, portability, and cross-platform functionality of your scripts.