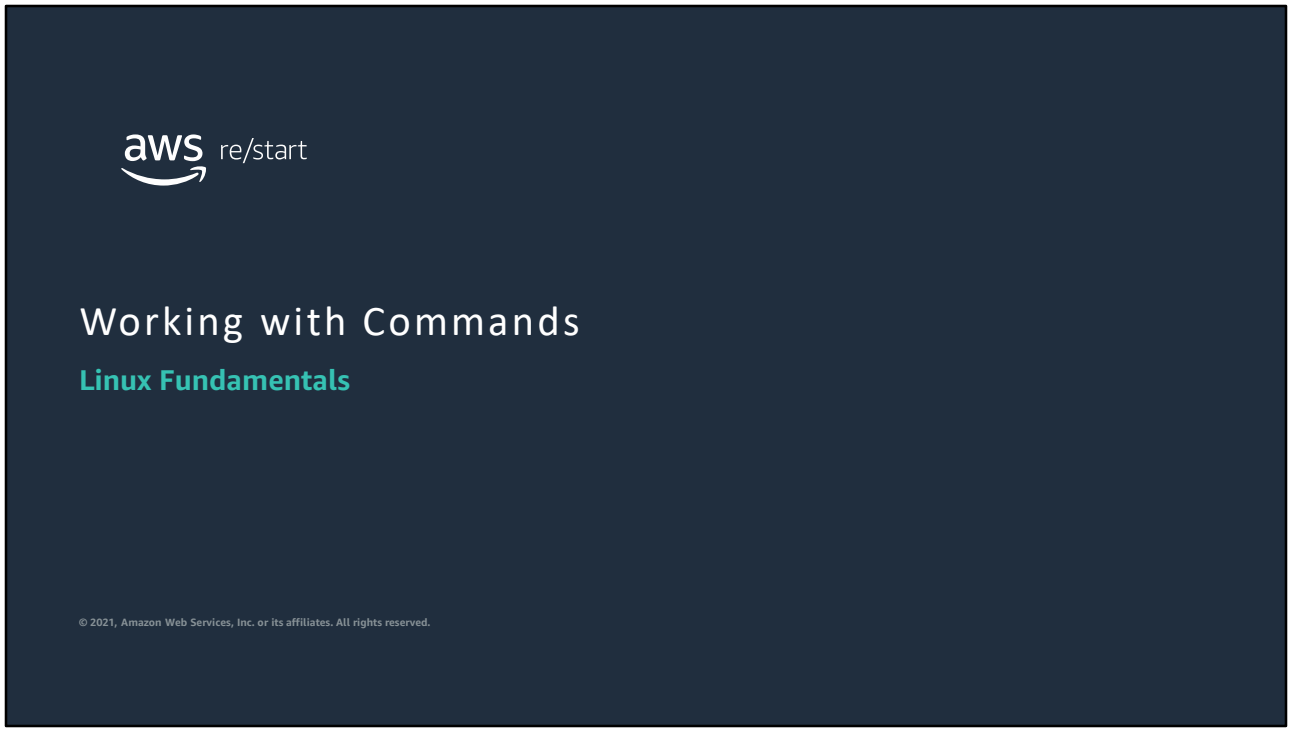


Welcome to Working with Commands.

What you will learn

At the core of the lesson

You will learn how to:

- Describe the purpose of special characters used with commands in Bash
- Describe commonly used text search and manipulation commands
- Explain redirection and describe common syntax for various redirect options

aws re/start

In this lesson, you will learn how to:

- Describe the purpose of special characters that are used with commands in Bash
- Describe commonly used text search and manipulation commands
- Explain redirection and describe common syntax for various redirect options

Special characters, wildcards, and redirection

Special characters, wildcards, and redirection are all used with commands in Bash. You'll see use cases for these options.

Using quotation marks with Bash

- In the Bash shell, a space is a *delimiter* (a separator)
  - Example: `usermod -c devuser jdoe`
    - This command adds the comment `devuser` on the `jdoe` line in the /etc/passwd file
    - The space separates the two arguments that are passed to the command `usermod`
- To force Bash to recognize the space inside an argument, a value is enclosed in quotation marks (" ")
  - Example: `usermod -c "This is a dev user"  jdoe`

```
jdoe:x:1002:1002:devuser:/home/jdoe:/bin/bash
jdoe:x:1002:1002:This is a dev user:/home/jdoe:/bin/bash
```


In the example, suppose that the user enters the following:

```
usermod -c This is a dev user  jdoe
```

Then, `usermod` will consider `This`, `is`, `a`, `dev`, `user`, and `joe` as command parameters, which will not work because `usermod -c` takes only two parameters.

Bash metacharacters

Metacharacter	Description
* (star)	Any number of any character (wildcard)
? (hook)	Any one character (wildcard)
[characters]	Any matching characters between brackets (wildcard)
`cmd` or $cmd	Command substitution—uses backticks (`), not single quotation marks (' ')
;	Chain commands together
~	Represents the home directory of the user
-	Represents the previous working directory

Note: Bash has many more metacharacters.

aws re/start

Bash metacharacters are special characters that have a meaning to the shell and that users can use to work faster and more powerful interaction with Bash. They are especially useful when writing scripts.

Bash metacharacters: * example

```
[ec2-user@myLinux ~]$ ls
Desktop    myfile  myFilesList.txt  pic.png
documents  myFile  myfile.txt       sales_22082020.txt
[ec2-user@myLinux ~]$ ls documents/
[ec2-user@myLinux ~]$ cp *.txt documents/
[ec2-user@myLinux ~]$ ls documents/
myFilesList.txt  myfile.txt  sales_22082020.txt
[ec2-user@myLinux ~]$
```


aws re/start

As you can see, the current folder contains the following:

- .txt files – `myfile.txt`, `myFilesList.txt`, and `sales_22082020.txt`
- `myFile` – A file without extension
- A .png file – `pic.png`
- Two folders – `Desktop` and `documents`

The `documents` folder is empty.

The command **`cp`** copied all .txt files to the documents folder.

The ***** replaces any character any number of times; only files that were named `<anycharacter_multiple_times>.txt` were copied.

Bash metacharacters: ? example

```
[ec2-user@myLinux ~]$ ls
customers_2020.txt  Desktop          sales_2018.txt  sales_2020.txt
customers_2021.txt  sales_2017.txt   sales_2019.txt  sales_2021.txt
[ec2-user@myLinux ~]$ rm sales_201?.txt
[ec2-user@myLinux ~]$ ls
customers_2020.txt  customers_2021.txt  Desktop  sales_2020.txt  sales_2021.txt
[ec2-user@myLinux ~]$
```


As you can see, the current folder contains the following:

- sales .txt files – `sales_2020.txt`, `sales_2021.txt`, `sales_2019.txt`, `sales_2018.txt`, `sales_2017.txt`
- Customers .txt files: `customers_2020.txt`, `customers_2021.txt`
- One folder: `Desktop`

The `rm sales_201?.txt` command deletes all files that are named `sales_201<anycharacter>.txt`. Only `sales_2019.txt`, `sales_2018.txt`, and `sales_2017.txt` are deleted.

Bash metacharacters: [characters] bracket example

- Brackets ([ ]): Matches any character between the brackets
- Characters can be numbers, letters, or special characters
- Works with
  - A list of characters: [aef9] matches only a, e, f, and 9
  - A character set: [a–g] matches letters from a to g

```
[ec2-user@myLinux ~]$ ls
Desktop  log_a.txt  log_b.txt  log_c.txt  log_d.txt  log_e.txt  log_f.txt
[ec2-user@myLinux ~]$ ls log_[abc].txt
log_a.txt  log_b.txt  log_c.txt
[ec2-user@myLinux ~]$ ls log_[a-e].txt
log_a.txt  log_b.txt  log_c.txt  log_d.txt  log_e.txt
[ec2-user@myLinux ~]$
```


aws re/start

- [abc] is a list of fixed characters: a, b, and c.
- ls log_[abc].txt lists log_a.txt, log_b.txt, and log_c.txt because the characters list is [abc].
- [a-e] means characters from a to e.
  - ls log_[a-e] lists log_a.txt, log_b.txt, log_c.txt, log_d.txt, and log_e.txt, but not log_f.txt.
- The characters list can be a bit more complex:
  - ls log_[a-zA-Z][0-9] means list any file that:
    - Starts with log_
    - Is followed by one character between a and z, lowercase or uppercase
    - Is followed by one number between 0 and 9
- Notice that you can combine wildcards together.
  - ls log_?[a-zA-Z0-9]* lists:
    - Files that begin with log_
    - Followed by any character (? wildcard)
    - Followed by one letter uppercase or lowercase, or a one-digit number
    - Followed by any number of any characters (* wildcard)

Bash metacharacters: Other examples

```
[ec2-user@myLinux ~]$ echo "Current path is ["$(pwd)"]"
Current path is [/home/ec2-user]
[ec2-user@myLinux ~]$ echo "Current path is ["`pwd`"]"
Current path is [/home/ec2-user]
[ec2-user@myLinux ~]$
```

```
[ec2-user@myLinux etc]$ pwd
/etc
[ec2-user@myLinux etc]$ cd ~/Documents/
[ec2-user@myLinux Documents]$ pwd
/home/ec2-user/Documents
[ec2-user@myLinux Documents]$ echo "command1"; echo "command2"
command1
command2
[ec2-user@myLinux Documents]$
```


- The first screenshot shows command substitution that uses `$` or `` ` ``. Inside the string, `$pwd` is replaced by the actual result of the `pwd` command.
- In the second screenshot, you can see how using `~/` goes directly to the current user home folder
    - `cd ~/Document` is equivalent to `cd /home/ec2-user/Documents`
- Finally, you can see how you can use `;` to run several commands.

Redirection operators

Operator	Description
>	Sends the output of a command to a file
<	Receives the input for a command from a file
|	Runs a command and redirects its output as input to another command
>>	Appends the output of a command to the existing contents of a file
2>	Redirects errors that are generated by a command to a file
2>>	Appends errors that are generated by a command to the existing contents of a file

Alert! By default, the **>** output redirector overwrites existing file content with no warning.

aws re/start

Familiarize yourself with these common operators that are used for redirection.

Alert! By default, the > output redirector will overwrite existing file content with no warning.

How the pipe redirector is used

- Examples:
  - `ps -ef | grep sshd`
  - `ls -l /etc | less`

```
[ec2-user@myLinux ~]$ ps -ef | grep sshd
root      3167     1  0 Jun01 ?        00:00:01 /usr/sbin/sshd -D
root      8030  3167  0 06:56 ?        00:00:00 sshd: ec2-user [priv]
ec2-user  8066  8030  0 06:56 ?        00:00:00 sshd: ec2-user@pts/0
root      8737  3167  0 07:32 ?        00:00:00 sshd: ec2-user [priv]
ec2-user  8772  8737  0 07:32 ?        00:00:00 sshd: ec2-user@pts/1
root      8981  3167  0 07:43 ?        00:00:00 sshd: ec2-user [priv]
ec2-user  9016  8981  0 07:43 ?        00:00:00 sshd: ec2-user@pts/2
root      9151  3167  0 07:50 ?        00:00:00 sshd: ec2-user [priv]
ec2-user  9185  9151  0 07:50 ?        00:00:00 sshd: ec2-user@pts/3
ec2-user  9216  9186  0 07:50 pts/3    00:00:00 grep --color=auto sshd
```


aws re/start

- `ps -ef | grep sshd` lists processes and redirects the output to the grep command that looks for the `sshd` pattern in the result of `ps`.
- `ls -l /etc | less` lists the content of the `/etc` folder and redirects the result to the `less` command where the user can navigate and save the content to a file.

How the redirectors > and >> are used

- Populating an `info.txt` file:
  - `uptime > info.txt`
  - `hostname >> info.txt`
  - `ip addr show eth0 >> info.txt`

```
[userA@server00 ~]$ uptime > info.txt
[userA@server00 ~]$ hostname >> info.txt
[userA@server00 ~]$ ip addr show enp0s3 >>info.txt
[userA@server00 ~]$ cat info.txt
 22:56:33 up  1:28,  3 users,  load average: 0.00, 0.01, 0.05
server00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
group default qlen 1000
    link/ether 08:00:27:5c:57:3f brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global noprefixroute dynamic enp0s3
       valid_lft 81096sec preferred_lft 81096sec
    inet6 fe80::67ef:dc4a:90a:6b0d/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```


- The first command outputs the results of the `uptime` command to the file `info.txt`.
- The second command appends the result of the `hostname` command to the same file, hence the second line: **`server00`**.
  - Note that **`hostname > info.txt`** with one > instead of two >> would have overwritten the `info.txt` file with only the `hostname` `info`.
- The third line also appends `info` to the same file.
- An example of using < is **`less < info.txt`**, which redirects the content of `info.txt` to the `less` command to display it.

Redirect errors

Other examples are:

- `myprogram 2>error.log`
  - Runs the program `myprogram` and sends errors to the `errors.log` file
- `find ../ -name 'p*' 2>error.log`
  - Tries to find files that start with `p` in the folder `../`
  - Errors are written in the `errors.log` file

```
[ec2-user@myLinux ~]$ find ../ -name 'p*' 2>error.log
../ec2-user/.vnc/passwd
[ec2-user@myLinux ~]$ cat error.log
find: '../mmajor': Permission denied
find: '../jdoe': Permission denied
[ec2-user@myLinux ~]$
```


aws re/start

In this example, the `find` command fails because `ec2-user` does not have access to folders of other users. The errors are logged in the `error.log` file.

The noclobber variable

- By default, output redirect overwrites an existing file with no warning.
- The noclobber variable can be set to prevent this behavior. It is not set on most Linux distributions by default.
- Examples:
  - set -o noclobber
  - echo "test1" > textfile.txt
  - echo "test2" > textfile.txt

```
[userA@server00 ~]$ set -o noclobber
[userA@server00 ~]$ echo "test1" > textfile.txt
[userA@server00 ~]$ echo "test2" > textfile.txt
bash: textfile.txt: cannot overwrite existing file
```


aws re/start

These commands are equivalent to the following:

```
echo "test1" > textfile.txt
echo "test2" >> textfile.txt
```

The pipe redirector

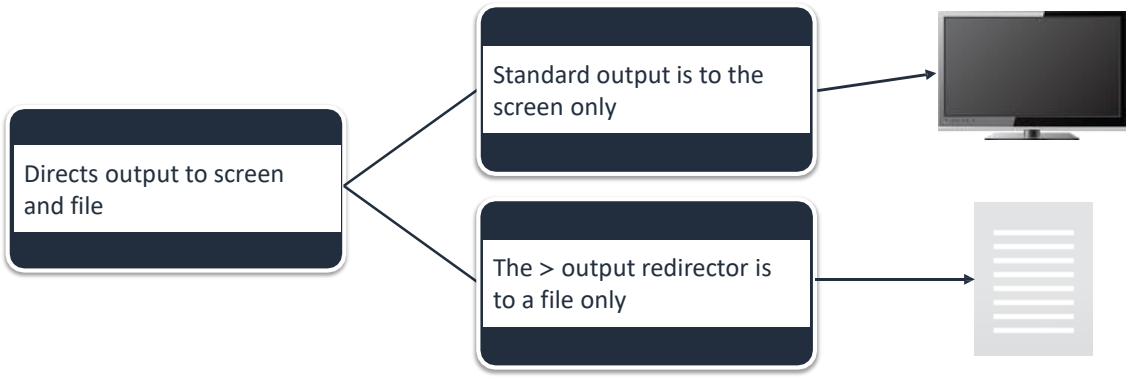
```
[ec2-user@myLinux ~]$ ps -au
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      3124  0.0  0.1 121284  1808 tty1     Ss+  Jun01   0:00 /sbin/agetty --
root      3125  0.0  0.2 120932  2124 ttyS0    Ss+  Jun01   0:00 /sbin/agetty --
ec2-user  9575  0.0  0.4 124844  4112 pts/4    Ss+  08:09   0:00 -bash
ec2-user 10586  0.0  0.4 124944  4144 pts/5    Ss+  09:07   0:00 -bash
ec2-user 11205  0.0  0.3 124844  3912 pts/0    Ss+  09:42   0:00 -bash
ec2-user 11674  0.0  0.3 124844  3908 pts/1    Ss   10:06   0:00 -bash
ec2-user 12065  0.0  0.3 164360  3836 pts/1    R+   10:24   0:00 ps -au
[ec2-user@myLinux ~]$ ps -au | grep ec2-user
ec2-user  9575  0.0  0.4 124844  4112 pts/4    Ss+  08:09   0:00 -bash
ec2-user 10586  0.0  0.4 124944  4144 pts/5    Ss+  09:07   0:00 -bash
ec2-user 11205  0.0  0.3 124844  3912 pts/0    Ss+  09:42   0:00 -bash
ec2-user 11674  0.0  0.3 124844  3908 pts/1    Ss   10:06   0:00 -bash
ec2-user 12074  0.0  0.3 164360  3920 pts/1    R+   10:25   0:00 ps -au
ec2-user 12075  0.0  0.0 119416   940 pts/1    S+   10:25   0:00 grep --color=au
to ec2-user
[ec2-user@myLinux ~]$ ps -au | grep ec2-user | awk '{print $1 $2}'
ec2-user9575
ec2-user10586
ec2-user11205
ec2-user11674
ec2-user12083
ec2-user12084
ec2-user12085
```


aws re/start

- You can chain several commands by using pipes (multi-stage piping), which is referred to as a *pipeline.*

- The output of the first command becomes the input of the second command:
  - `ps -au` lists processes.
  - The result of `ps` is sent to `grep ec2-user`, which looks for the word `ec2-user`.
  - The result of `grep` is sent to the `awk` command, which prints the first two columns of the previous result, the process id, and the user name (`awk` is a scripting language and is an advanced Linux topic).

The `tee` command




```
[userA@server00 ~]$ hostname | tee file1.txt
server00
[userA@server00 ~]$ cat file1.txt
server00
```


aws re/start

The `tee` command reads the standard input (stdin) and writes the data to both to the standard output and files.

In the example, the command `hostname` is directed to `tee` through a pipe `|` .

The standard input for `tee` in the output of the command `hostname`. The `tee` command then writes the hostname to the file `file1.txt` and to the screen (in the shell).

Command substitution, chaining, and filtering

This section provides details on command substitution, chaining, and filtering, which are used to nest commands, run a series of commands, and extract text.

Command substitution

- Allows a command to be nested in a command line or within another command. The result of that command is displayed or used by the rest of the command.
- Used with the backtick (`) (older form).
- Can be accomplished with `$(command)` (newer form).

```
[root@server00 ~]# cat demo.sh
#!/bin/bash
DATE=`date`
echo "Today's date is $DATE."
USERS=`who | wc -l`
echo "There are $USERS logged in."
[root@server00 ~]# ./demo.sh
Today's date is Mon Mar 11 00:35:58 GMT 2019.
There are 2 logged in.
[root@server00 ~]#
```


aws re/start

Command substitution is useful when writing Bash scripts.

Using the semicolon to chain commands

A semicolon (;) is used to run a series of commands, all written on a single line.

```
[root@server00 ~]# date ; w ; uptime
Mon Mar 11 16:02:49 GMT 2019
 16:02:49 up 15:49,  2 users,  load average: 0.08, 0.04, 0.05
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
student0 :0       :0               00:27   ?xdm?   5:18   0.37s /usr/libexec/gn
student0 pts/0    :0               00:27    1.00s  0.41s  6.67s /usr/libexec/gn
 16:02:50 up 15:49,  2 users,  load average: 0.08, 0.04, 0.05
```


aws re/start

Chaining commands is similar to writing small scripts. Another example is the following:

- `yum update ; yum install httpd ; systemctl start httpd &`

This command updates packages on the system, installs an Apache HTTP server, and starts it.

You can use & to run tasks in the background so that you can keep working on the shell.

Chaining and running tasks in the background can save you a lot of time.

Using | grep

- `grep` is commonly used after another command, along with a pipe (|).
- Examples:
  - `ps -ef | grep sshd`
  - `cat /var/log/secure | grep fail`

```
[root@server00 ~]# ps -ef | grep -i sshd
root      1221     1  0 00:13 ?        00:00:00 /usr/sbin/sshd -D
root     28208  6128  0 16:04 pts/0    00:00:00 grep --color=auto -i sshd
[root@server00 ~]# rpm -qa | grep samba
samba-common-libs-4.7.1-6.el7.x86_64
samba-common-4.7.1-6.el7.noarch
samba-client-libs-4.7.1-6.el7.x86_64
```


The `grep` command is used to search text and strings in a given file.

The cut command

- Cuts sections from lines of text by character, byte position, or delimiter
- Displays that information to standard output
- Can be used to pull relevant information out of text files and display that information to you
- Output can be piped to a new file

```
[ec2-user@myLinux ~]$ cat names.csv
Alejandro,Rosalez,42,Cherbourg,FR
Carlos Salazar,33, Paris,FR
Li Juan,25,Bordeaux,FR
[ec2-user@myLinux ~]$ cut -d ',' -f 1 names.csv
Alejandro
Carlos Salazar
Li Juan
[ec2-user@myLinux ~]$
```

```
[ec2-user@myLinux ~]$ cut -c 1-2 names.csv
Al
Ca
Li
[ec2-user@myLinux ~]$ cut -b 1-5 names.csv
Aleja
Carlo
Li Ju
[ec2-user@myLinux ~]$
```


aws re/start

The `cut` command requires the user to specify bytes, fields, or characters to extract. When using a field, you must specify the delimiter of the file.
Command options are listed as follows.
`-b`: Byte
`-c`: Column
`-f`: Field
`-d`: Delimiter
Examples are as follows.

- `cut -d ',' -f 1 names.cvs`: Extracts the first field of each record. The separator is the comma (`,`).
- `cut -c 1-2 names.csv`: Extracts the first two characters of each line.
- `cut -b 1-5 names.csv`: Extracts the first five bytes of each line. Depending on the encoding, one letter can be encoded by using one or more bytes.

The `b` and `c` options can also be used with lists.

- `cut -c 1,6,7 names.csv`: Extracts the characters 1, 6, and 7 of each record.

Or you can use it as follows:

- `cut -c 4- names.csv`: Extracts from characters 4 to the end.
- `cut -c -3 names.csv`: Extracts from the first character to the third character of each record.

Text manipulation and searching

In this section, you will learn about commonly used text search and manipulation commands.

The sed command

- A non-interactive text editor
- Edits data based on the rules that are provided (can insert, delete, search, and replace)
- Supports regular expression

```
[ec2-user@myLinux ~]$ echo "example.com page" |sed 's/page/website/'
example.com website
[ec2-user@myLinux ~]$ cat example.txt
example.com page
[ec2-user@myLinux ~]$ sed 's/page/website/' example.txt
example.com website
[ec2-user@myLinux ~]$
```


aws re/start

The examples in the screenshot are equivalent. The first takes the `echo` output as an input (use of the pipe redirector `|`), and the second works on a file.

`sed 's/page/website/' example.txt` replaces `page` occurrences with `website` in the `example.txt` file. (It takes the content of the file as input but does not save the file.)

By default, `sed` replaces only the first occurrence on each line.

You can use `/n` to replace the nth occurrence;

`sed 's/page/website/5' example.txt` replaces the fifth occurrence.

You can use `/g` to replace all occurrences.

You can use other options to do more advanced text manipulations (delete and replace strings on ranges of lines).

The `sort` command

- Sorts file contents in a specified order: alphabetical, reverse order, number, or month
- Examples:
  - `sort file.txt`: Outputs lines in alphabetical order
  - `sort -r file.txt`: Outputs lines in reverse alphabetical order
  - `sort -u file.txt`: Removes duplicate entries (useful for log files)
  - `sort -M file.txt`: Outputs lines in order of month

```
[ec2-user@myLinux ~]$ cat names.csv
Alejandro,Rosalez,42,Cherbourg,FR
Carlos Salazar,33, Paris,FR
Li Juan,25,Bordeaux,FR
John Doe,51,Lyon,FR
[ec2-user@myLinux ~]$ sort -c names.csv
sort: names.csv:4: disorder: John Doe,51,Lyon,FR
```

```
[ec2-user@myLinux ~]$ sort names.csv
Alejandro,Rosalez,42,Cherbourg,FR
Carlos Salazar,33, Paris,FR
John Doe,51,Lyon,FR
Li Juan,25,Bordeaux,FR
```


aws re/start

By default, the entire line is taken as a sort key:

- Lines that begin with a number will appear first.
- Lines that begin with an *a* appear before lines that begin with other letters.
- Lines that begin in lowercase appear before lines that begin in uppercase.
- `-o` outputs the result to a file (`sort file.txt –o sortedfile.txt` is like `sort file.txt > sortedfile.txt`)
- `-r` sorts in reverser order
- `-n` sorts numerically if the file contains numbers
- `-k` sorts according to the kth column (if the file is formatted as a table)
- `-u` removes duplicates
- `-c` tells whether a file is already sorted

The awk command

- Is used to write small programs to transform data
- Defines variables
- Uses string and arithmetic operators
- Uses control flow and loops
- Generates formatted reports
- Syntax: Two ways to invoke awk – one with an explicit program, one with the program in a file
  - `awk option –f program-file input-file`
  - `awk option 'program' input-file`
- Options:
  - `-F fs` To specify a field separator (the default separator any number of spaces or tab)
  - `-f source-file` To specify a file that contains awk script
  - `-v var=value` To declare a variable

aws re/start

The awk command does not require compiling. It is aimed at writing small programs. The name comes from the names of the three developers: Aho, Weinberger, and Kernighan.

The awk command continued

```
[ec2-user@myLinux ~]$ cat names.csv
Alejandro Rosalez,42,Cherbourg,FR,arosalez@company.com
Carlos Salazar,33,Paris,FR,csalazar@company.com
Li Juan,25,Bordeaux,FR,ljuan@company.com
John Doe,,Lyon,FR,jdoe@company.com
[ec2-user@myLinux ~]$ awk -F , '{print$3}' names.csv
Cherbourg
Paris
Bordeaux
Lyon
[ec2-user@myLinux ~]$ awk -F @ '{print$1}' names.csv
Alejandro Rosalez,42,Cherbourg,FR,arosalez
Carlos Salazar,33,Paris,FR,csalazar
Li Juan,25,Bordeaux,FR,ljuan
John Doe,,Lyon,FR,jdoe
```

```
[ec2-user@myLinux ~]$ awk -F , '/[0-9][0-9]/ {print $1 }' names.csv
Alejandro Rosalez
Carlos Salazar
Li Juan
```


aws re/start

- awk is used as follows: `awk options 'program' inputFile`
- program can be in the form {action}
  - `awk –F , '{ print $3 }' customers.txt`
    - The field separator is a comma (,)
    - The program is `print $3`: Prints the third field on each record in the `teams.txt` file
  - `awk –F @ '{print $1}' customers.txt`
    - The field separator is @ so the first field becomes everything that is before that @ instead of the first name as before
- program can also be in the form `select_record_or_field {action}`
- The program can use a regular expression to match records or rules against specific rules:
  - `awk options ' /regexp/ { action}'`
  - For example: `awk -F , '/[0-9][0-9]/ {print $1 }' names.csv` selects lines that contain a two-digit number. For this reason, the last line of the file is filtered out.
  - `awk -F , '$2 > 35 {print $1}' names.csv` selects only records for which the second field is > 35 : Alejandro Rosalez
- awk can use a special pattern to perform actions before and after record is processed:
  - `awk 'BEGIN { print "Start Processing." }; { print $1 }; END { print "Done ! :]" }' names.csv`
  - This program prints `Start Processing`, then displays the first field of each record, and finally displays `Done!:]`

Checkpoint questions

When do you use the ? wildcard instead of the * wildcard?

How does the `uniq` command help with log-file analysis?

How can command substitution make a Bash script run faster? Does command substitution have any other advantages?

aws re/start

1. The question mark (?) specifies that only a single character should be considered as the wildcard. The asterisk (*) specifies that the wildcard can contain one or more characters.
2. With the `uniq` command, you can more easily manage the administration of files, such as log, with many duplicate lines of text. For example, running `uniq -c logfile.log` would output each unique line. Using the `-c` option includes the count of occurrences from the log file.
3. If a command must run for a long time to complete, then running the same command repeatedly will negatively affect the script's performance. Running the command once—and storing the output in a variable—can make it faster. Command substitution also improves script maintenance because the update requires only one change. Without command substitution, updating the script would require searching the entire file for every occurrence of the command. Important: If the output from the command might change while the script is running, do not use command substitution.

Key takeaways

- Quotation marks (" ") override the usual Bash interpretation of an argument with a space as two separate arguments.
- Metacharacters are powerful tools to control output, wildcards, and chaining commands.
- Standard I/O for Bash is keyboard in, monitor out.
- Wildcards are used to specify one to many unknown characters or a set of limited and specific values in a search.
- The output of one command can be sent to another command by using a pipe (|).
- `grep` can be used to search the piped output of a previous command.
- The `sed`, `sort`, and `awk` commands are used for text manipulation and searching.

aws re/start

The following are the key takeaways for the Working with Commands lesson:

- Quotation marks (" ") override the usual Bash interpretation of an argument with a space as two separate arguments.
- Metacharacters are powerful tools to control output, wildcards, and chaining commands.
- Standard I/O for Bash is keyboard in, monitor out.
- Wildcards are used to specify one to many unknown characters or a set of limited and specific values in a search.
- The output of one command can be sent to another command by using a pipe (|).
- `grep` can be used to search the piped output of a previous command.
- The `sed`, `sort`, and `awk` commands are used for text manipulation and searching.

Thank you

aws re/start

Thank you.