Week 02

IT-230

- Shell Expansions
- Quoting
- Transferring Files
- Using Regular Expressions

IT-230 · Linux Administration

Start `workstation` and `servera` on your VDI

IT-230 · Linux Administration

Shell Expansions

Matching file names before commands run

Bash Is an Interpreter

You type a command. **Bash reads it before anything runs.**

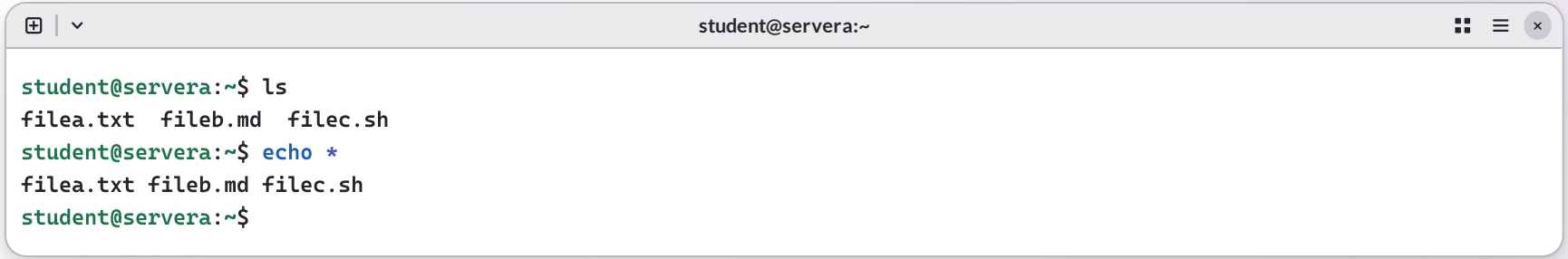
Characters such as `*`, `?`, and `[` mean something **to Bash**.

When Bash sees them in a command line:

- It searches the file system for matching names
- It replaces the pattern with matching file names
- It hands the expanded list to the command

Pathname Expansion

Bash replaces a pattern with the names of the files that match it, **before** running the command.

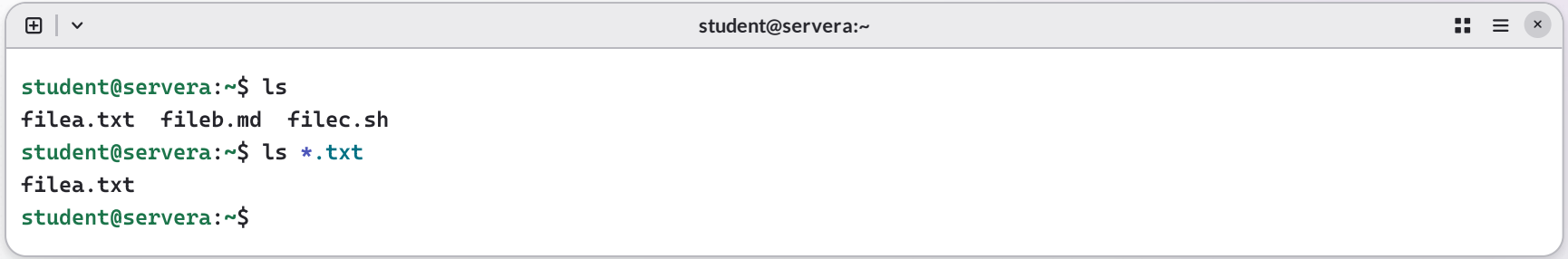


`echo` never received a `*`. It received three file names.

This is called **globbing**.

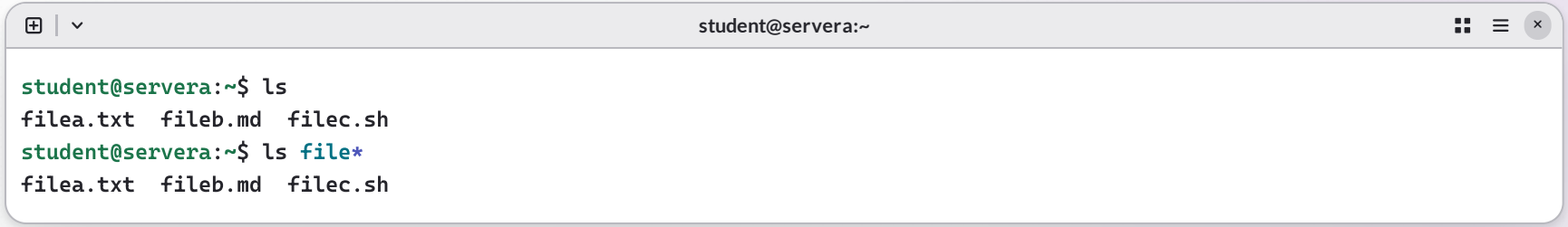
Match Any Characters: `*`

- `*` matches **any number of characters**, including none



A glob matches whole file names, so `*.txt` means "anything, then `.txt`"

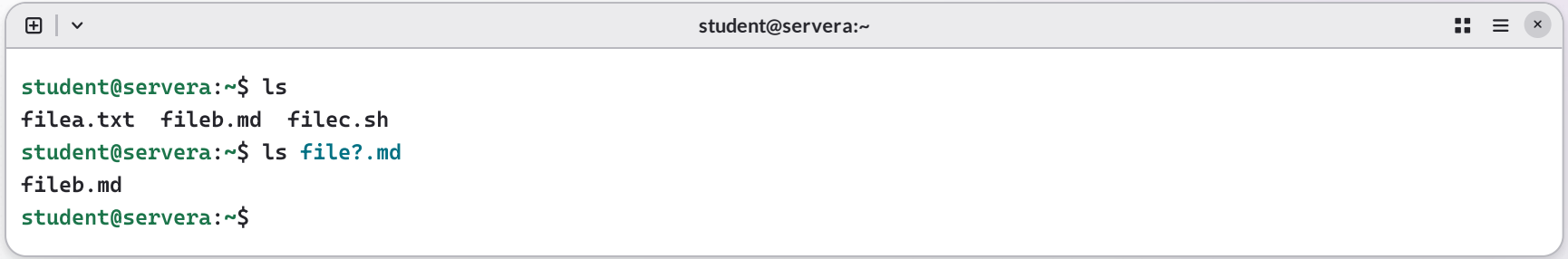
Match Any Characters: `*`



A glob matches whole file names, so `file*` means "`file` then anything"

Match Exactly One Character: `?`

`?` matches **exactly one character**



`file?.md` matches `fileb.md`, but will not match `file.md` or `file10.md`.

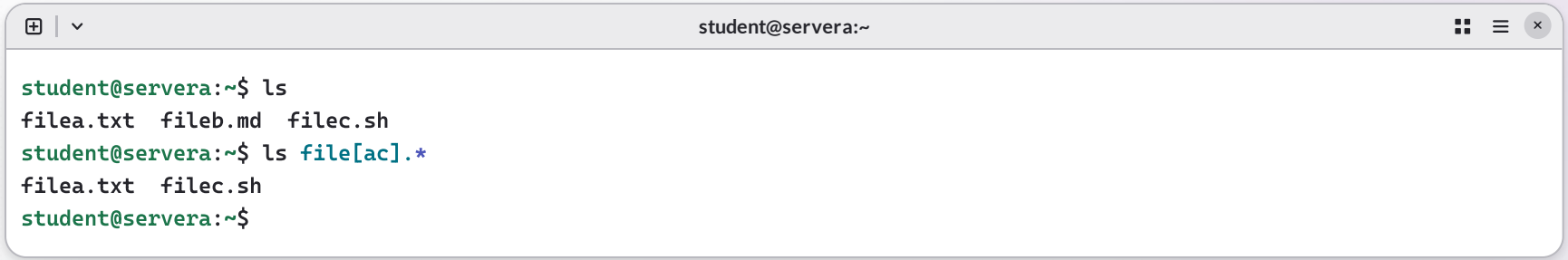
Match Exactly One Character: `?`

`file?.*` matches:

- `file`
- Followed by one character
- Followed by a dot
- Followed by anything

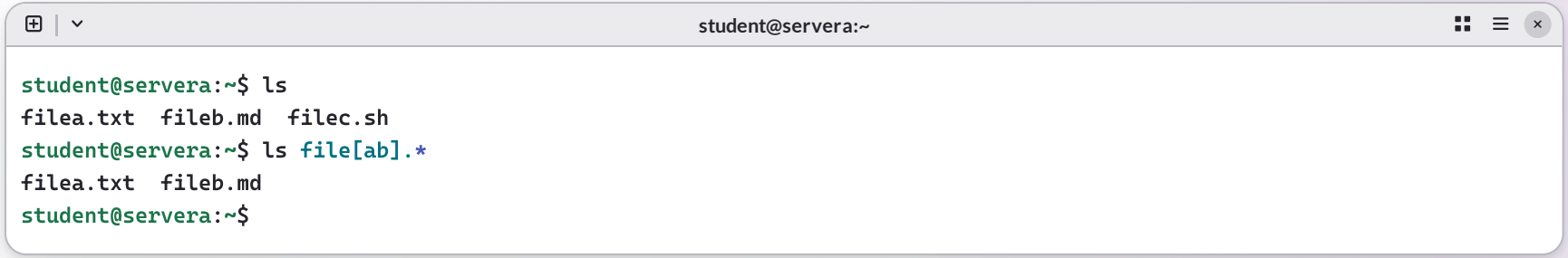
Match a Set of Characters: `[...]`

`[...]` matches **one character from the set** listed inside the brackets.



`file[ac].*` matches `filea` or `filec` followed by any extension, but ignores `fileb.md`.

Match a Set of Characters: `[...]`

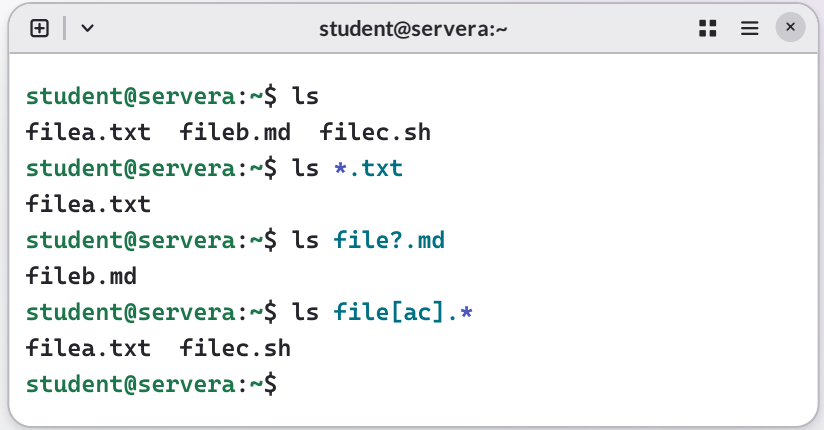


`file[ab].*` matches:

- `filea` or `fileb`
- Followed by a dot
- Followed by anything

The Three Glob Patterns

Pattern	Matches
`*`	any run of characters, including none
`?`	exactly one character
`[ab]`	one character from the set



HANDS-ON EXERCISE

Exercise: Shell Expansions

GOAL

Match pathnames with Bash wildcard patterns and observe the resulting expansion

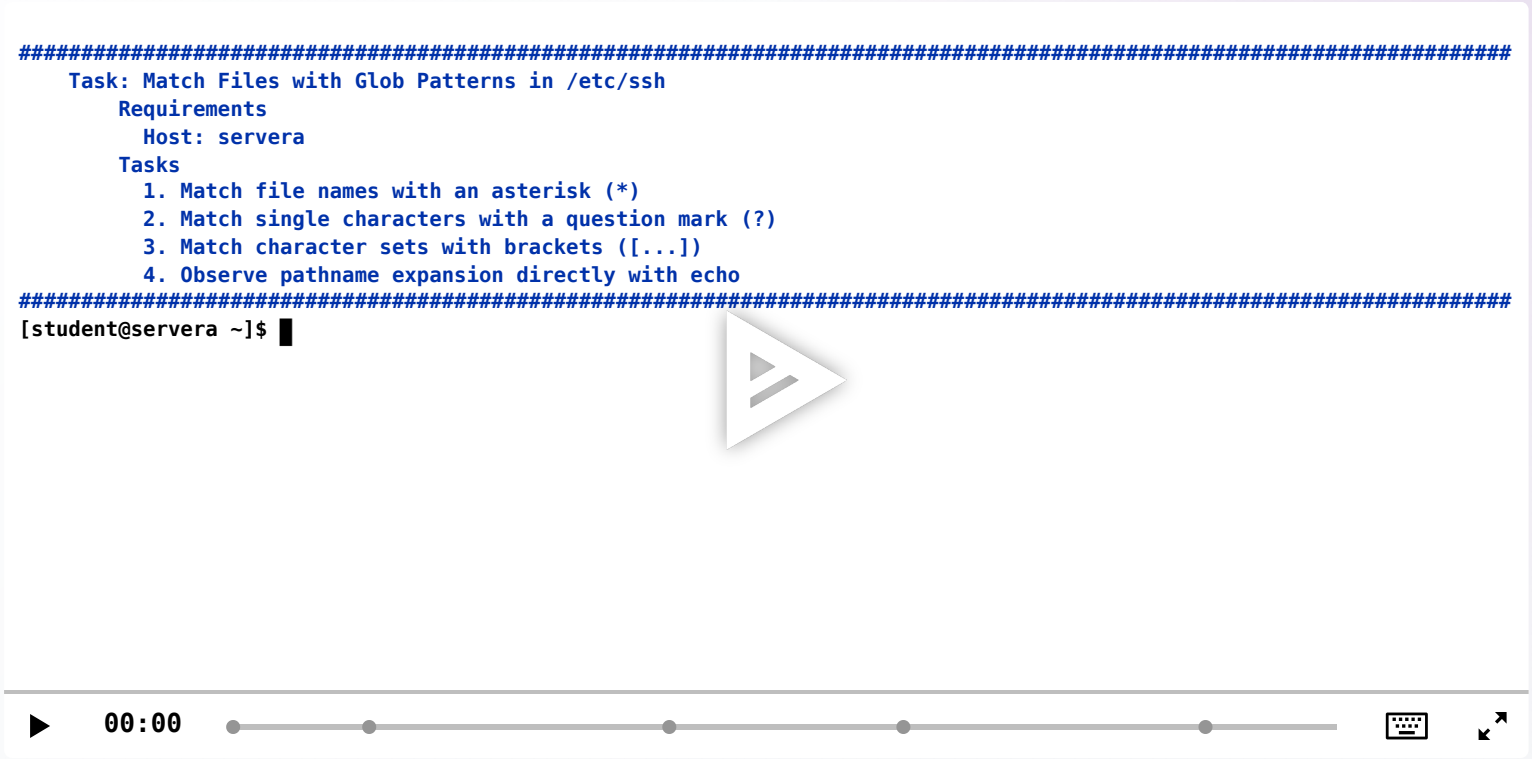
ENVIRONMENT

Host: `servera`

WORKFLOW

1. Match file names with an asterisk (`*`)
2. Match single characters with a question mark (`?`)
3. Match character sets with brackets (`[...]`)
4. Observe pathname expansion directly with `echo`

Exercise: Shell Expansions



Written exercise

IT-230 · Linux Administration

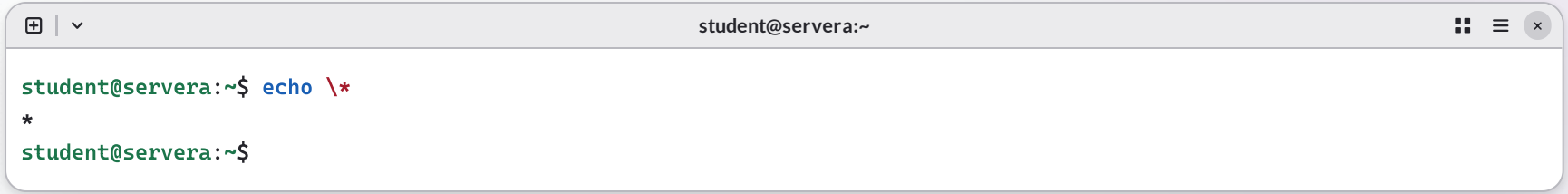
Quoting

Protecting characters from Bash expansion

Protect a Character: Backslash

Backslash `\`

Protects exactly **one** following character.

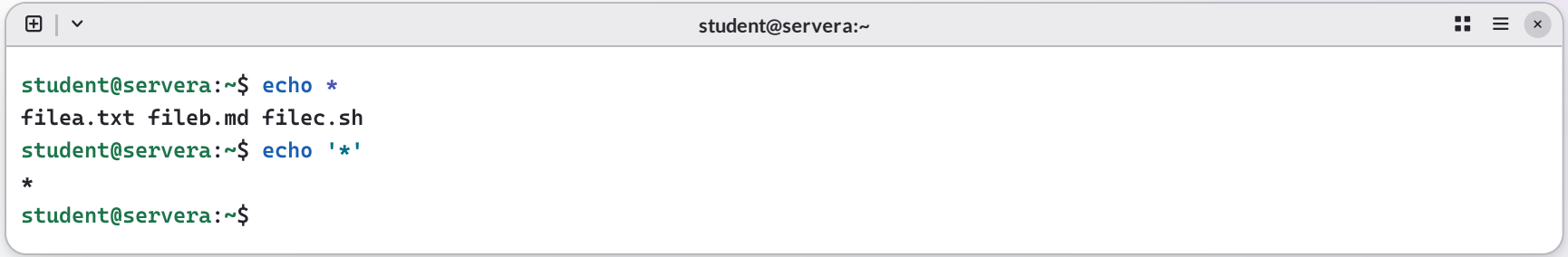


The backslash protects the asterisk without quoting the whole argument.

Protect a Character: Single Quotes

Single quotes `'...'`

Everything inside is **literal**. Bash expands nothing.



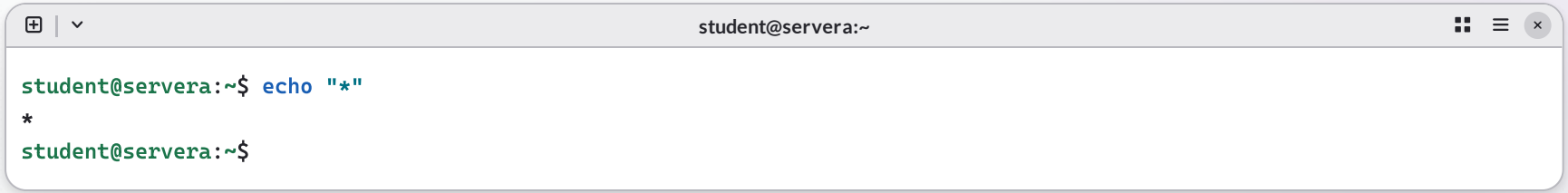
The single quotes make Bash hand the asterisk to `echo` unchanged.

Protect a Character: Double Quotes

Double quotes `"..."`

The asterisk stays **literal**

Bash can still expand variables (`$HOSTNAME`) and command substitutions (`$(...)`).



Use double quotes when some expansion should remain available.

Variables: A Preview of Next Week

Assign with `name=value`

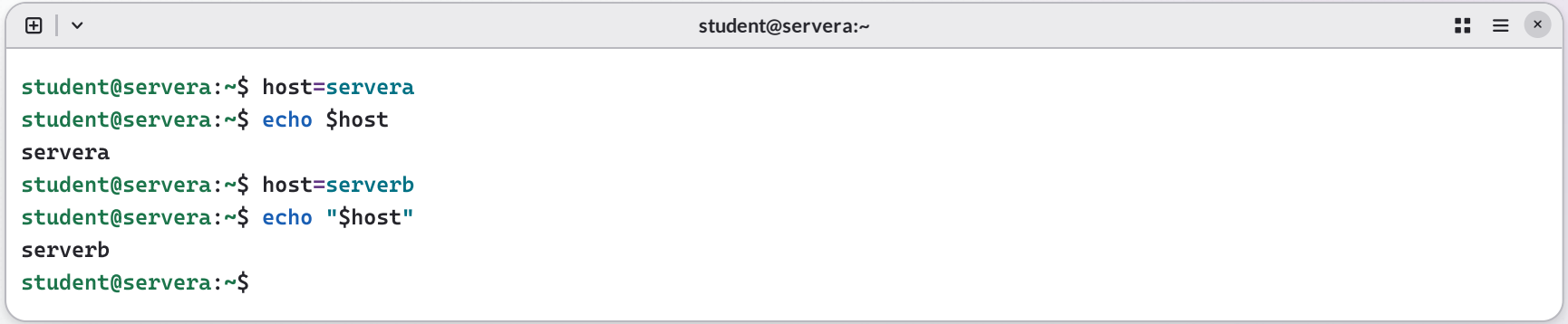
student@servera:~

```
1 student@servera:~$ host=servera
2 student@servera:~$
```

> **Warning**
>
> No spaces around the equal sign!

Variables: A Preview of Next Week

Read the value back by prefixing the name with `$`.



We will build on this during the shell scripting class.

Same Command, Three Quoting Styles

student@servera:~

```
1  student@servera:~$ host=servera
2  student@servera:~$ echo 'Checking $host'
3  Checking $host
4  student@servera:~$ echo "Checking $host"
5  Checking servera
6  student@servera:~$ echo "Checking ${host}.lab.example.com"
7  Checking servera.lab.example.com
8  student@servera:~$
```

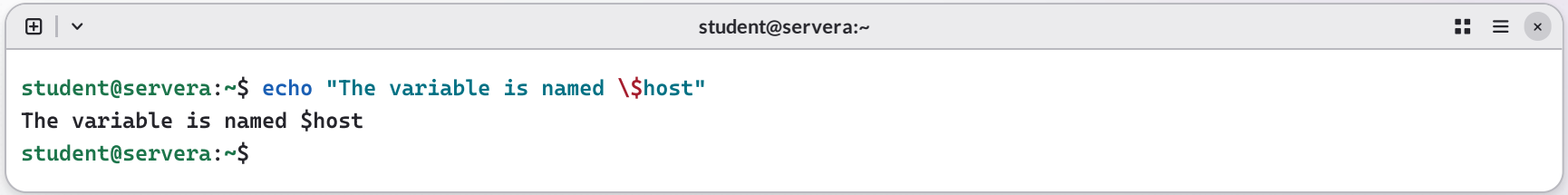
Single quotes print the name.

Double quotes print the value.

Braces mark where the name ends.

Escaping Inside Double Quotes

A backslash still works inside double quotes



Spaces Separate Arguments

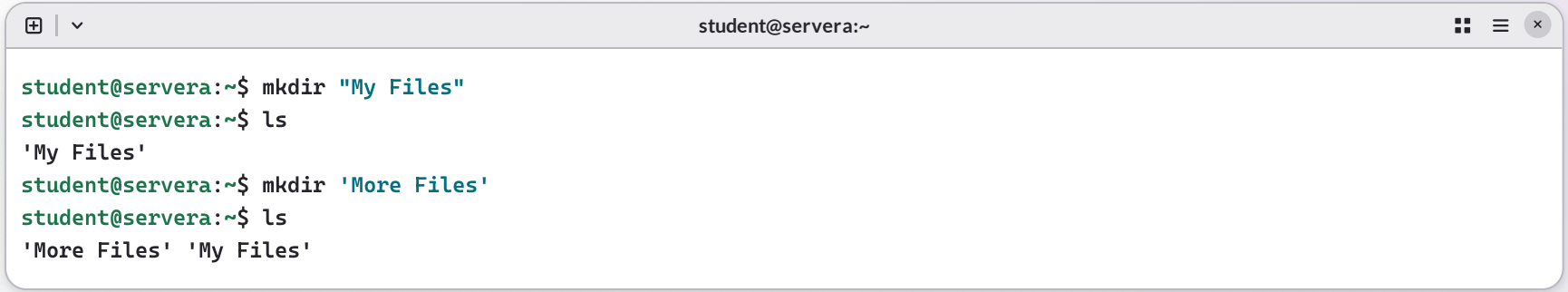
`mkdir My Files` creates **two** directories: `My` and `Files`

student@servera:~

```
1  student@servera:~$ mkdir My Files
2  student@servera:~$ ls
3  Files  My
4  student@servera:~$
```

Spaces Separate Arguments

Quote the whole name



Using single or double quotes

Spaces Separate Arguments

Escape the space

HANDS-ON EXERCISE

Exercise: Quoting

GOAL

Control how Bash interprets variables, wildcards, and spaces

ENVIRONMENT

Host: `servera`

WORKFLOW

1. Print a literal asterisk (*)
2. Assign a value to a variable
3. Use single and double quotes
4. Include spaces in arguments

Exercise: Quoting

```
##########################################################################################################################
    Task: Manage Shell Expansion with Quoting
        Requirements
          Host: servera
        Tasks
          1. Print a literal asterisk (*)
          2. Assign a value to a variable
          3. Use single and double quotes
          4. Include spaces in arguments
##########################################################################################################################
[student@servera ~]$
```

00:00

Written exercise

IT-230 · Linux Administration

scp

Secure copy

`cp` over `ssh`

`scp` is used to upload and download files from a remote machine

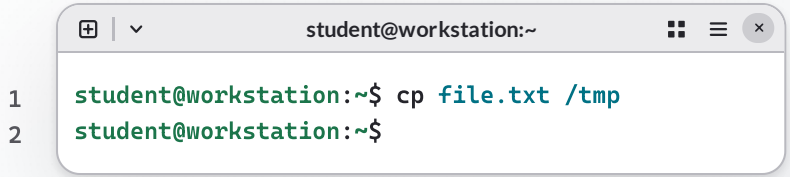
- `scp` is part of the **OpenSSH** suite and rides on top of `ssh`
- Installed by default on RHEL, macOS, and Windows and later
- Available anywhere SSH is available — nothing extra to configure
- Authenticates exactly the way `ssh` does

> **Note**
>
> If you can `ssh` to a host, you can `scp` to it.

`cp` and `scp`

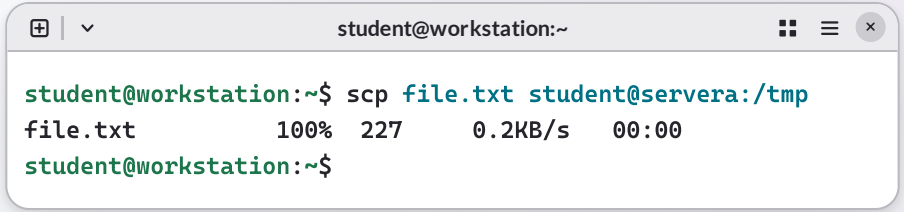
Local copy

Copy `file.txt` into `/tmp`



Remote copy

Copy `file.txt` into `/tmp` on `servera`



Uploading a File

```
scp file.txt student@servera:/tmp
```

The destination directory on that host

Download a File

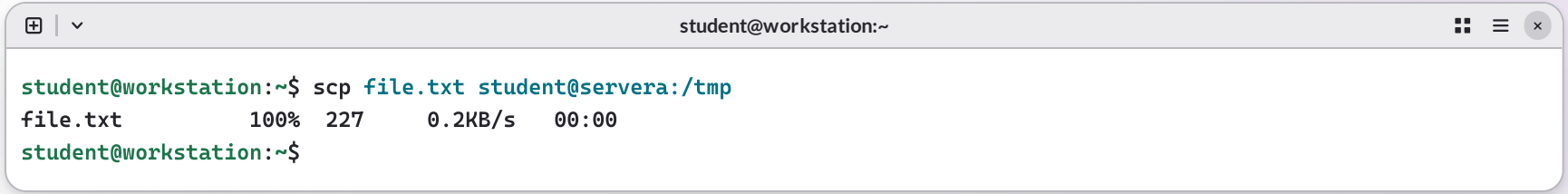
```
scp student@servera:/tmp/notes.txt ~
```

The destination local path (your home directory)

Upload or Download

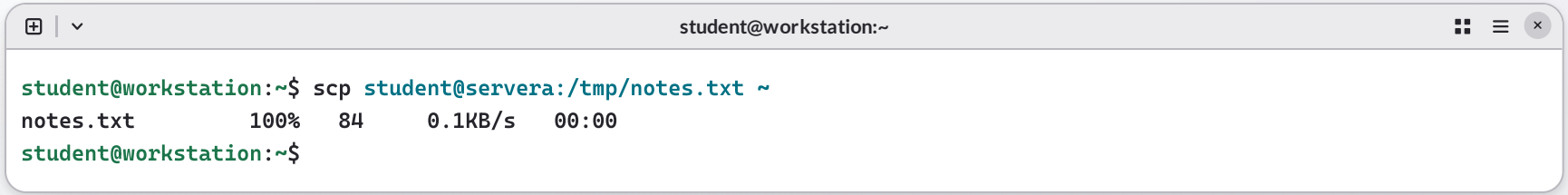
Upload a File

Local path in first argument, remote host in second argument



Download a File

Remote host in first argument, local path in second argument



Copying a Whole Directory

`-r` recursively copies entire directories, exactly as it does for `cp`

student@workstation:~

```
1 student@workstation:~$ scp -r Documents student@servera:/tmp
2 student@workstation:~$
```

> **Warning**
>
> `scp` copies. It does not compare, skip, or resume — every byte crosses the network every time.

HANDS-ON EXERCISE

Exercise: Copying Files with `scp`

GOAL

Transfer individual files and wildcard-selected groups among three systems

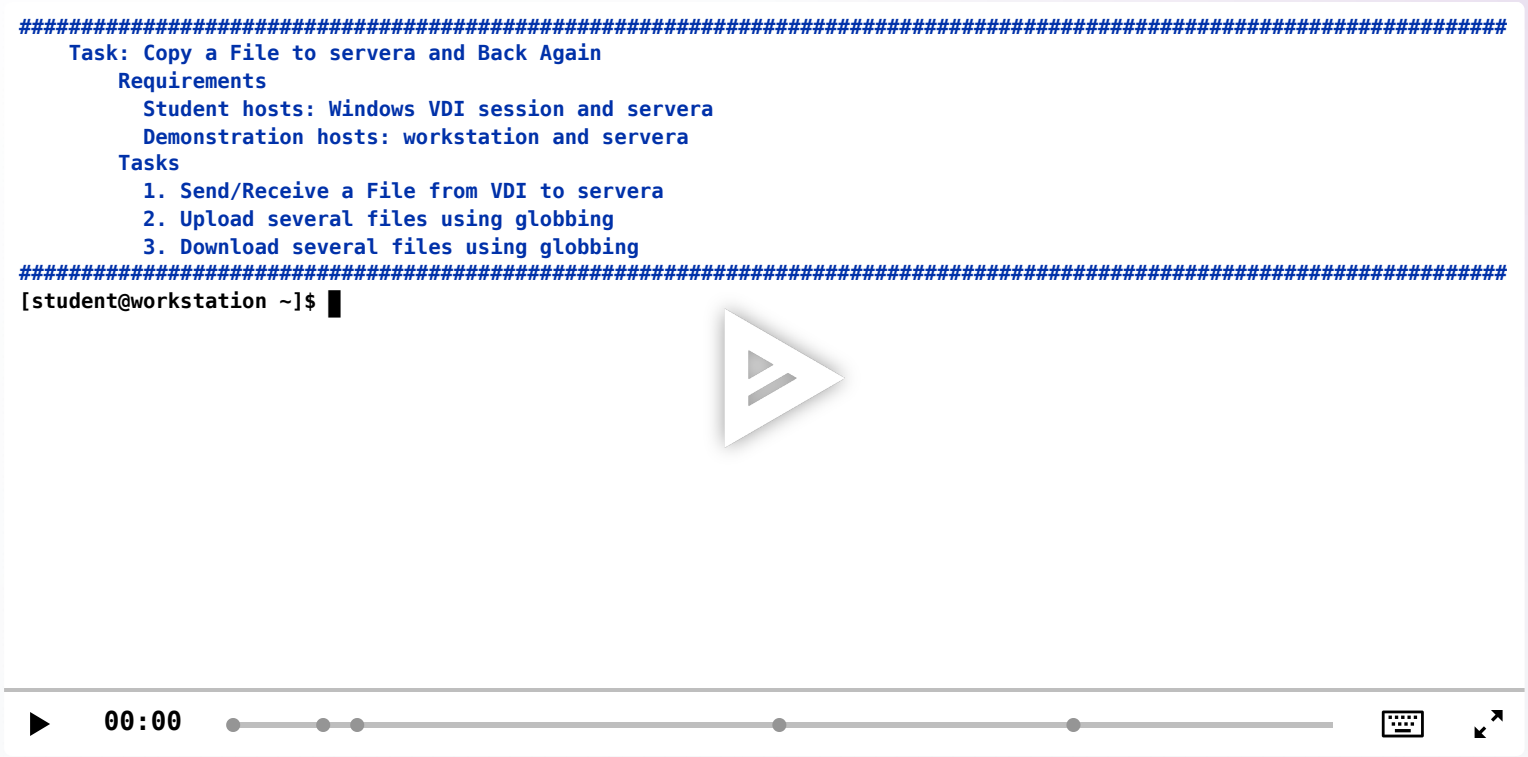
ENVIRONMENT

Hosts: your Windows VDI, `workstation`, and `servera`

WORKFLOW

1. Transfer a file between the Windows VDI and `servera`
2. Upload several files using globbing
3. Download several files using globbing

Exercise: Copying Files with `scp`



Written exercise

IT-230 · Linux Administration

`sftp`

Secure File Transfer Protocol

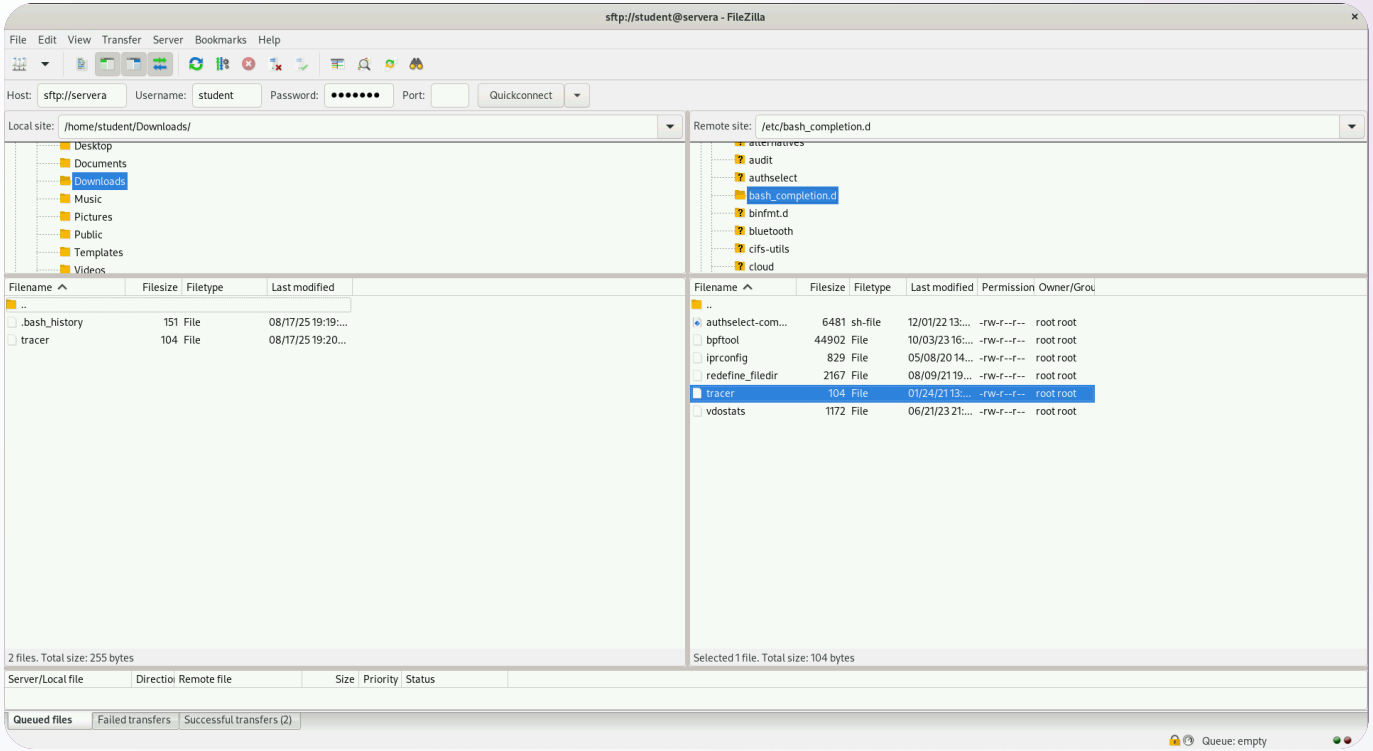
An Interactive Session Instead of One Command

`sftp` drops you into its own prompt so you can look around before deciding what to move

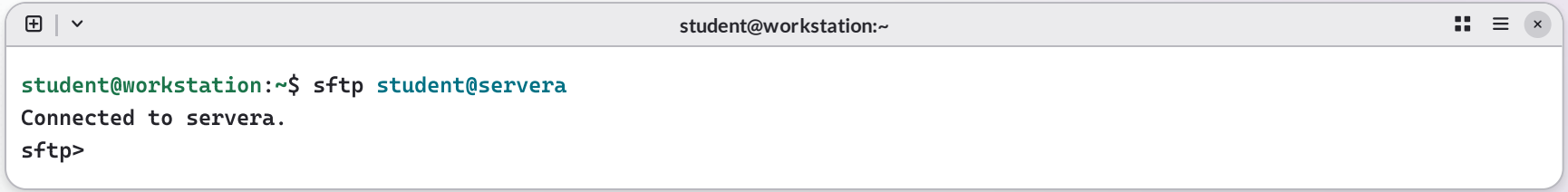
- `sftp` is also part of the **OpenSSH** suite and also rides on `ssh`
- Browse the remote directory tree
- Change directories on **both** ends
- Upload and download repeatedly in one connection

Filezilla

Local on the Left - Remote on the Right



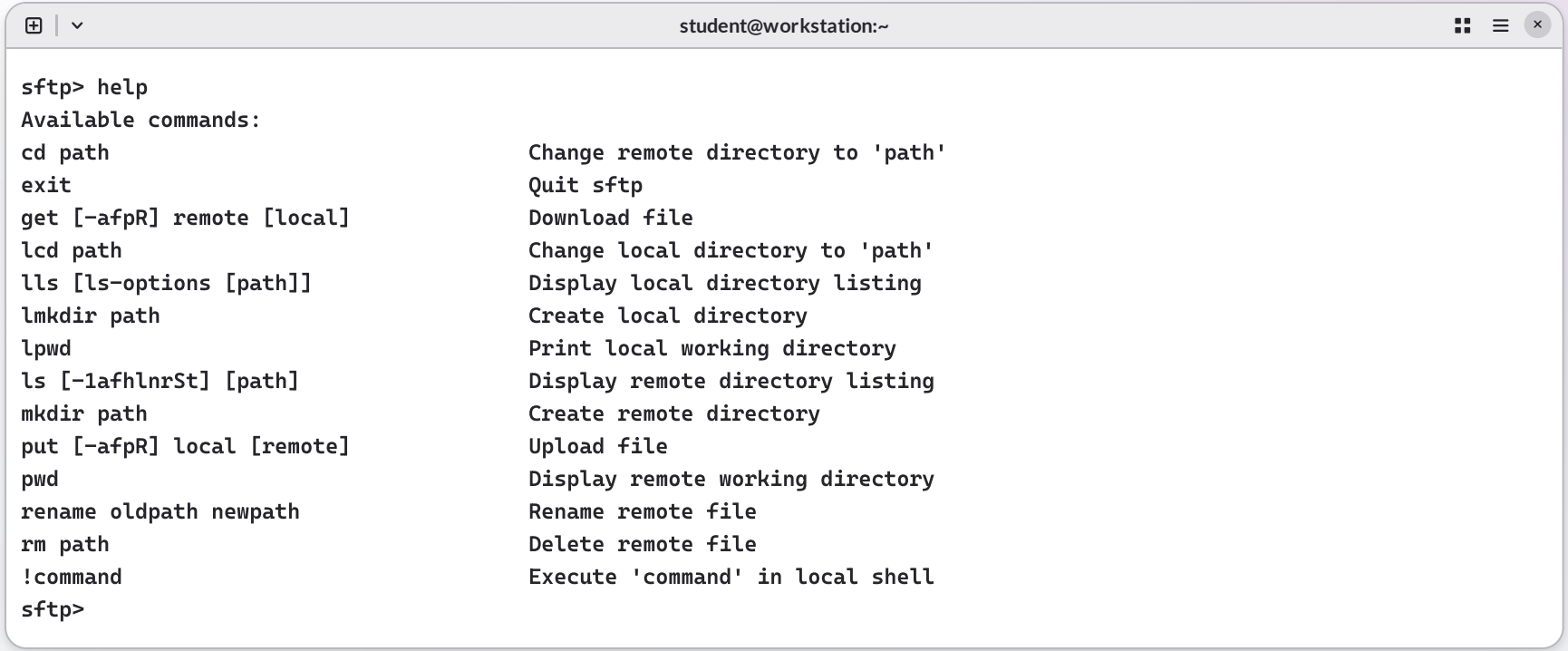
Connecting



The `sftp>` prompt replaces your shell prompt

Bash is not running here, only the commands `sftp` implements

`help` Lists Everything Available



Two Machines, One Prompt

Look carefully at that list: several commands appear twice

Remote by default

Plain commands act on **remote**

Command	Acts on the remote
`cd`	change directory
`pwd`	print directory
`ls`	list directory
`mkdir`	create directory

Local with a leading `l`

Prefix with `l` to act on **local**

Command	Acts on the local
`lcd`	change directory
`lpwd`	print directory
`lls`	list directory
`lmkdir`	create directory

`get` is Named From Your Side

```
sftp>  get  /etc/passwd
```

Read from the remote working directory, written into the local working directory

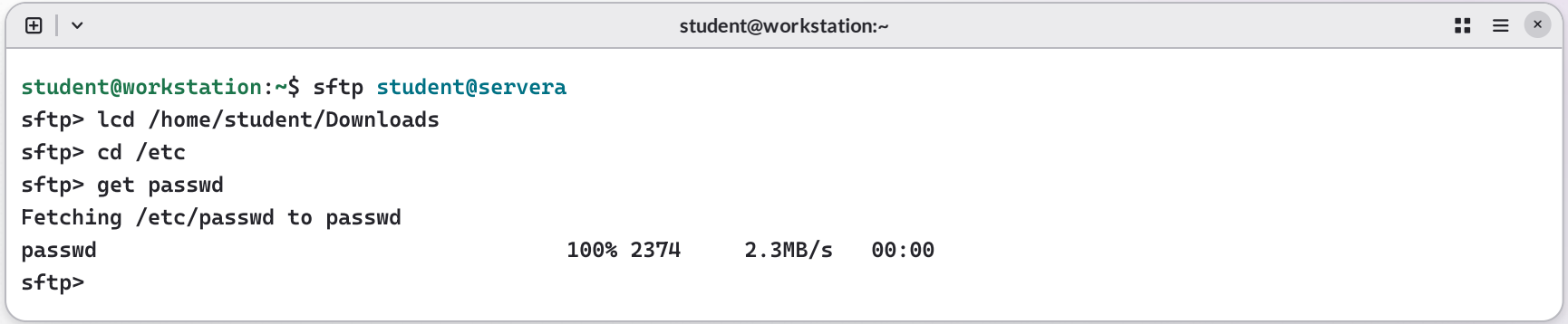
`put` is Named From Your Side

```
sftp>  put  /etc/passwd
```

Read from the local working directory, written into the remote working directory

Downloading

Set the directory on each side, then `get`



`lcd` chose where it landed (Destination)

`cd` chose where it came from (Source)

Uploading

The same two commands, in the other direction

student@workstation:~

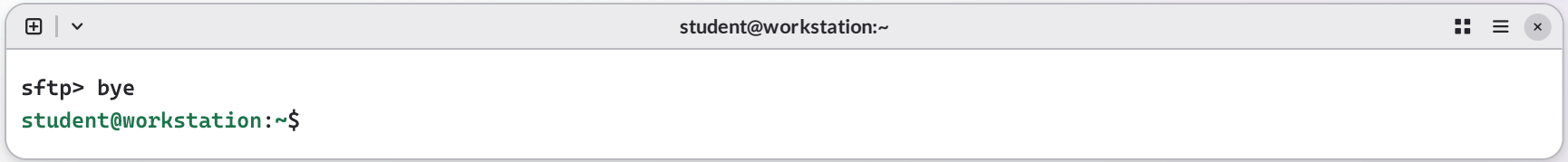
```
sftp> mkdir hostbackup
sftp> cd hostbackup
sftp> put /etc/hosts
Uploading /etc/hosts to /home/student/hostbackup/hosts
/etc/hosts                                    100%  227     0.2KB/s   00:00
sftp>
```

Note

`mkdir` created the directory on `servera`, because it has no leading `l`

Leaving

`bye`, `exit`, and `quit` all close the session and return you to your shell



`sftp` Without an Interactive Session

`sftp` can be used non-interactively similar to `scp`

student@workstation:~/Downloads

```
1 student@workstation:~/Downloads$ sftp student@servera:/etc/group
2 Connected to servera.
3 Fetching /etc/group to group
4 group                                       100% 1118     1.1MB/s   00:00
5 student@workstation:~/Downloads$
```

- Same `user@host:path` shape as `scp`
- No `sftp>` prompt when the remote path names a file
- File lands in the current local directory

HANDS-ON EXERCISE

Exercise: Downloading Files with `sftp`

GOAL

Download and verify a remote file through an interactive transfer session

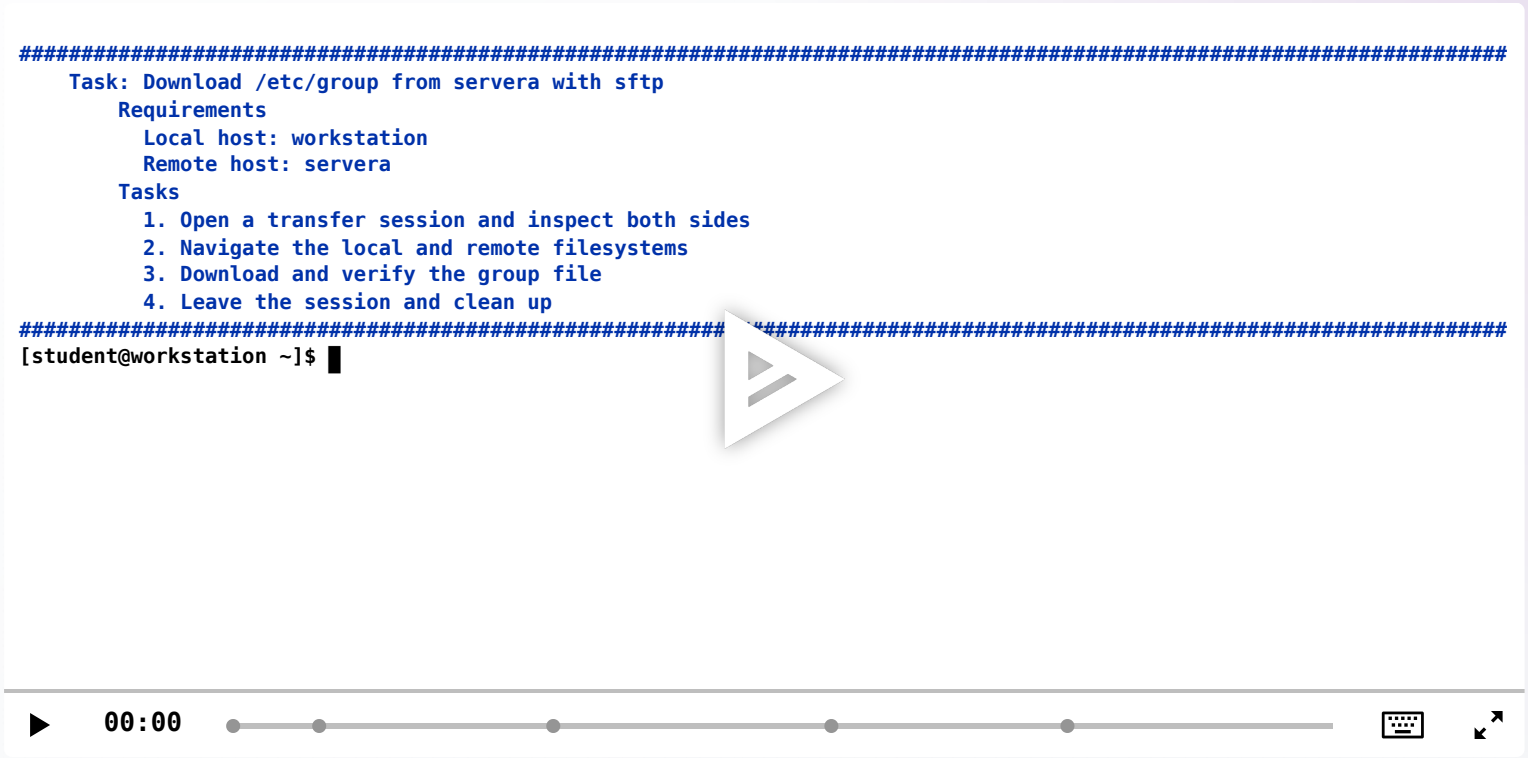
ENVIRONMENT

Hosts: `workstation` and `servera`

WORKFLOW

1. Open a transfer session and inspect both sides
2. Navigate the local side to `Downloads` and the remote side to `/etc`
3. Download and verify the `group` file
4. Exit the session and inspect the downloaded file

Exercise: Downloading Files with `sftp`



Written exercise

IT-230 · Linux Administration

`rsync`

Fast, versatile, remote (and local) file-synchronization

Copying Versus Synchronizing

`scp` copies

Every byte crosses the network, every time

Copy a 4 GB directory; 4 GB is sent

Copy the same 4 GB directory; 4 GB is sent

> **Tip**
>
> `scp` is installed with `ssh`
>
> Nearly all linux machiens will have it

`rsync` synchronizes

It compares the two sides

Sends only what differs

The first run costs the same as `scp`

Every run after that is **dramatically faster**

> **Warning**
>
> `rsync` must be installed on **both** machines
>
> It is not part of a minimal installation

Synopsis

student@workstation:~

```
student@workstation:~$ man rsync
NAME
       rsync - a fast, versatile, remote (and local) file-copying tool

SYNOPSIS
       Local:
           rsync [OPTION...] SRC... [DEST]

       Access via remote shell:
           Pull:
               rsync [OPTION...] [USER@]HOST:SRC... [DEST]
           Push:
               rsync [OPTION...] SRC... [USER@]HOST:DEST
```

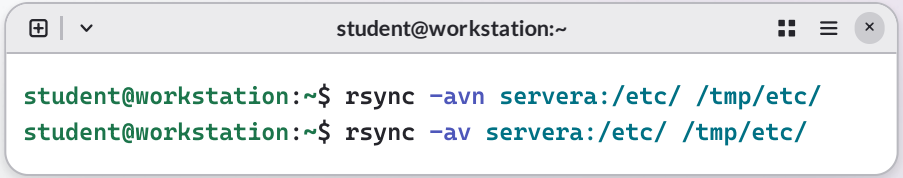
Same shape as `scp`: source first, destination second, a colon marks the remote side

Three Options Worth Knowing

`rsync` has **many** options. These three are the most important

Option	Meaning
`-a`	**a**rchive mode: recurse and preserve permissions, ownership, and timestamps
`-v`	**v**erbose: name each file as it transfers
`-n`	dry ru**n**: change nothing, report what would happen

Typical Usage



Run it with `-n` first

Read the list

Then run it again without `-n`

Reading an `rsync` Command

```
rsync -av servera:/var/log /tmp
```

The destination directory on local

The First Run Copies Everything

student@workstation:~

```
student@workstation:~$ rsync -av student@servera:~/Documents/ ~/Documents/
receiving incremental file list
./
notes.txt
projects/roadmap.md
projects/budget.csv

sent 92 bytes  received 4,412 bytes  9,008.00 bytes/sec
total size is 4,102  speedup is 0.91
student@workstation:~$
```

The Second Run Sends Almost Nothing

Change one file, then synchronize again

student@workstation:~

```
student@workstation:~$ echo "one more line" >> ~/Documents/notes.txt
student@workstation:~$ rsync -av ~/Documents/ student@servera:~/Documents/
sending incremental file list
notes.txt

sent 285 bytes  received 41 bytes  652.00 bytes/sec
total size is 4,116  speedup is 12.63
student@workstation:~$
```

Only `notes.txt` moved.

The other files were already identical on both sides

Mind the Trailing Slash

This is the mistake everyone makes exactly once

With a trailing slash

```
1 rsync -av ~/Documents/ servera:/tmp/backup/
```

Copies the **contents** of `Documents`

Result: `/tmp/backup/notes.txt`

Without a trailing slash

```
1 rsync -av ~/Documents servera:/tmp/backup/
```

Copies the **directory itself**

Result: `/tmp/backup/Documents/notes.txt`

> **Caution**
>
> A trailing slash changes what gets created. Use `-n` if unsure

HANDS-ON EXERCISE

Exercise: Synchronizing with `rsync`

GOAL

Preview and synchronize directory changes between two systems

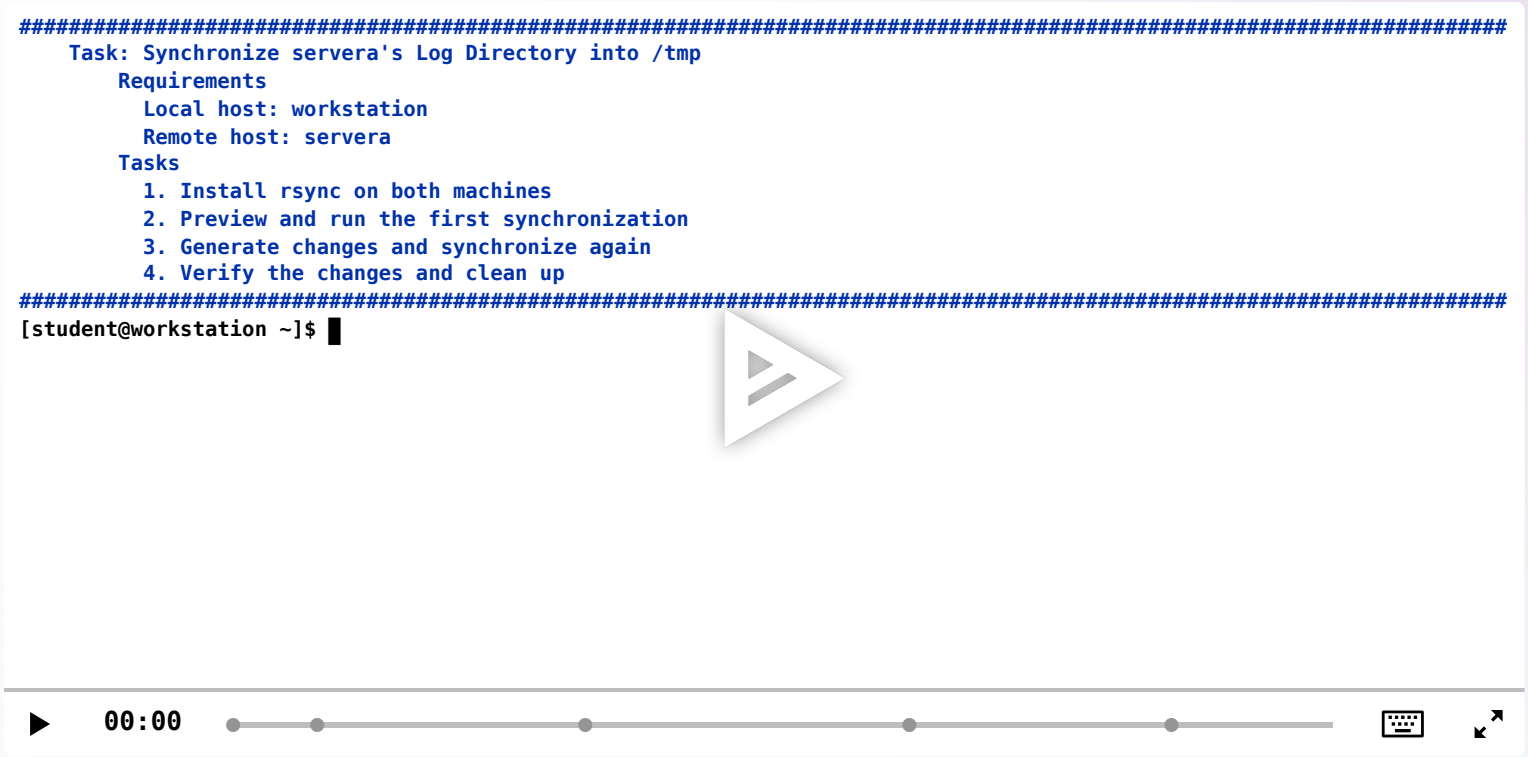
ENVIRONMENT

Hosts: `workstation` and `servera`

WORKFLOW

1. Install rsync on both machines
2. Preview and run the first synchronization
3. Generate changes and synchronize again
4. Verify the synchronized changes

Exercise: Synchronizing with `rsync`



Written exercise

IT-230 · Linux Administration

`grep`

Global · Regular Expression · Print

`Ctrl + F` on Steroids

A regular expression describes a **pattern** instead of one exact string

- Find the error messages buried in a log file
- Pull every IP address out of that log file
- Check that a password has an uppercase letter and a digit
- Find and replace across a whole project in Vim or VS Code

> **Note**
>
> If you have ever run `grep`, you have already used a regular expression

A Real-World Scenario

The task: your manager wants every unique address that failed to log in over SSH

The evidence: `/var/log/secure`

```
Jan 12 10:30:01 servera sshd[1188]: Failed password for user from 10.0.2.2
Jan 12 10:30:08 servera sshd[1192]: Accepted password for student from 172.25.250.9
Jan 12 10:30:15 servera sshd[1204]: Failed password for root from 203.0.113.55
Jan 12 10:30:20 servera sshd[1209]: Failed password for root from 203.0.113.55
```

The report: one line per address, with a count

```
      1 10.0.2.2
      2 203.0.113.55
```

By the end of this chapter you will be able to write that one-liner yourself

`grep` : Global · Regular Expression · Print

Written by **Ken Thompson** at Bell Labs and first released in **1973**

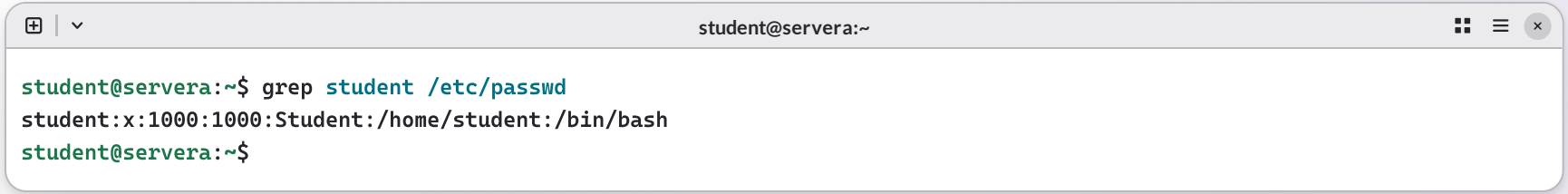
Prints every **line** of its input that matches a pattern

```
grep  PATH  ~/.bashrc
```

The file to search

Searching a File

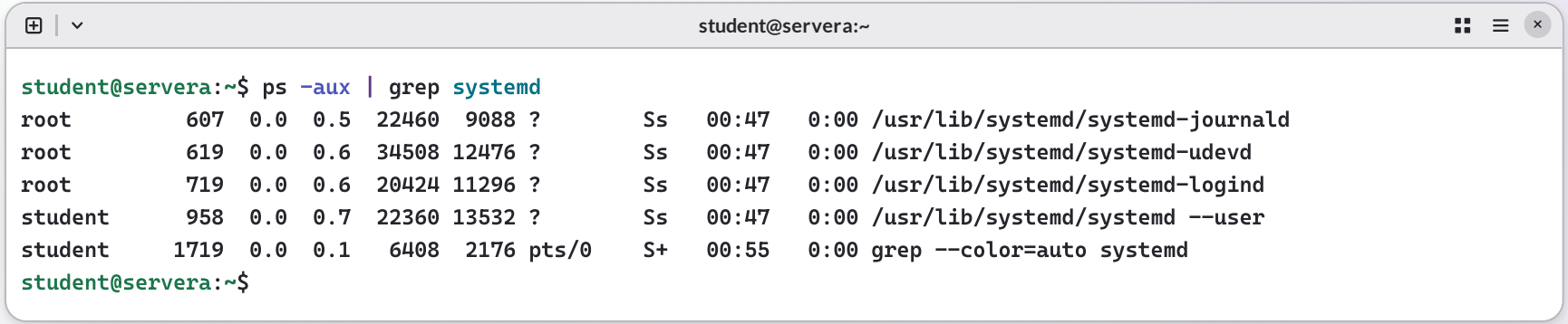
What will this command do?



It prints every line of `/etc/passwd` that contains the string `student`

Searching a Pipe

`grep` reads standard input when you give it no file

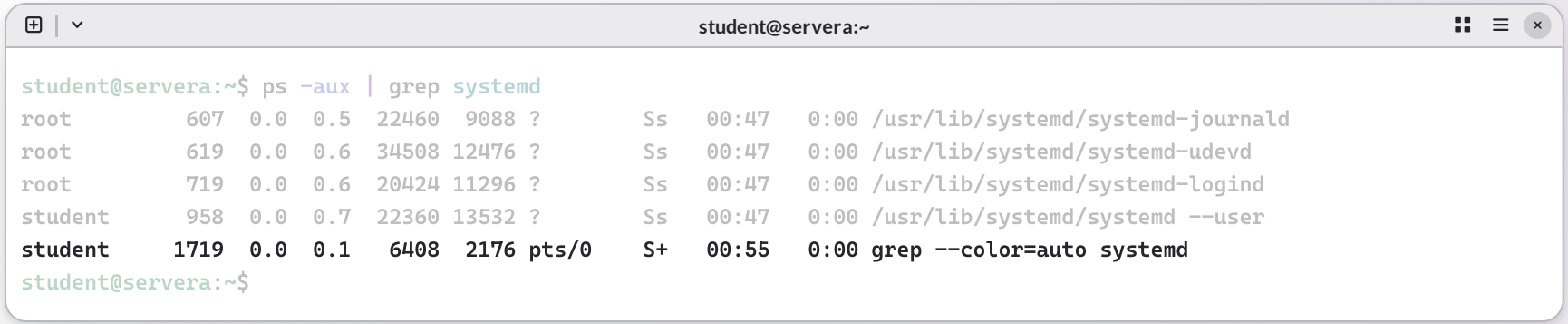


The output of `ps -aux` becomes the input to `grep`

`grep` Finds Itself

The last line is the `grep` command you just ran

It was in the process table when `ps` looked, and it contains the word `systemd`



Invert the Match with `-v`

`-v` prints every line that **does not** match

student@servera:~

```
student@servera:~$ ps -aux | grep systemd | grep -v grep
root         607  0.0  0.5  22460  9088 ?        Ss   00:47   0:00 /usr/lib/systemd/systemd-journald
root         619  0.0  0.6  34508 12476 ?        Ss   00:47   0:00 /usr/lib/systemd/systemd-udevd
root         719  0.0  0.6  20424 11296 ?        Ss   00:47   0:00 /usr/lib/systemd/systemd-logind
student      958  0.0  0.7  22360 13532 ?        Ss   00:47   0:00 /usr/lib/systemd/systemd --user
student@servera:~$
```

The stray `grep --color=auto systemd` line is gone

Several Patterns with `-e`

Match `systemd or sshd`; give `-e` once per pattern

student@servera:~

```
student@servera:~$ ps -aux | grep -e systemd -e sshd | grep -v grep
root         607  0.0  0.5  22460  9088 ?        Ss   00:47   0:00 /usr/lib/systemd/systemd-journald
root         619  0.0  0.6  34508 12476 ?        Ss   00:47   0:00 /usr/lib/systemd/systemd-udevd
root         719  0.0  0.6  20424 11296 ?        Ss   00:47   0:00 /usr/lib/systemd/systemd-logind
root         765  0.0  0.5  15852  9212 ?        Ss   00:47   0:00 sshd: /usr/sbin/sshd -D [listener]
root         819  0.0  0.6  18820 11464 ?        Ss   00:47   0:00 sshd: student [priv]
student      958  0.0  0.7  22360 13532 ?        Ss   00:47   0:00 /usr/lib/systemd/systemd --user
student     1055  0.0  0.3  19024  7008 ?        S    00:47   0:00 sshd: student@pts/0
student@servera:~$
```

Ignore Case with `-i`

Without `-i`, `Finished` and `finished` are different patterns

student@servera:~

```
student@servera:~$ sudo grep -i finished /var/log/messages
Aug 26 00:47:59 servera systemd[1]: Finished Record Runlevel Change in UTMP.
Aug 26 00:47:59 servera systemd[1]: Finished User Runtime Directory /run/user/1000.
Aug 26 00:47:59 servera systemd[958]: Finished Create User's Volatile Files and Directories.
Aug 26 00:47:59 servera systemd[958]: Startup finished in 74ms.
Aug 26 00:47:59 servera systemd[1]: Finished Crash recovery kernel arming.
Aug 26 00:47:59 servera systemd[1]: Startup finished in 1.299s (kernel) + 1.274s (initrd) + 2.308s (userspace).
```

Capital `Finished` on the first click, lowercase `finished` on the second; only `-i` returns both

Context Around the Match

A matching line is rarely the whole story; three options print its neighbors

Option	Prints
`-B3`	3 lines before the match
`-A3`	3 lines after the match
`-C3`	3 lines on both sides

Each has a long form: `--before-context=3`, `--after-context=3`, `--context=3`

Choosing one

- You have an **error code** and want the message that follows: use `-A`
- You have a **symptom** and want what led to it: use `-B`
- You want the **whole neighborhood**: use `-C`

HANDS-ON EXERCISE

Exercise: Searching Log Files with `grep`

GOAL

Search log data and refine the surrounding context of each match

ENVIRONMENT

Host: `servera`

WORKFLOW

1. Find matching log entries
2. Show context around matches
3. Refine results with grep options

Exercise: Searching Log Files with `grep`

Written exercise

IT-230 · Linux Administration

Regular Expressions

Also known as regexp, or just regex

Saying These Symbols Out Loud

Regular expressions are built from punctuation, and punctuation is hard to talk about in a classroom

We will borrow names from **INTERCAL**, a parody programming language from 1972

- It was designed to be as unlike every other language as possible
- Its compiler wants you to write `PLEASE`, but gets annoyed if you overdo it
- Its reference manual has **Tonsils** instead of Appendices, another removable organ
- It renamed "asterisk" to **Splat**

> **Note**
>
> The names are a memory aid, not standard terminology; the symbols are what matter

Tonight's Two Anchors

Symbol	Keyboard	INTERCAL name	Matches
`^`	Shift + 6	Shark fin	The beginning of a line
`$`	Shift + 4	Big money	The end of a line

Neither one matches a character; each matches a **position**

> **Note**
>
> Every pattern from here on is wrapped in single quotes, for the reason we saw earlier: `grep` should read it, not Bash

Anchor to the Start of a Line with `^`

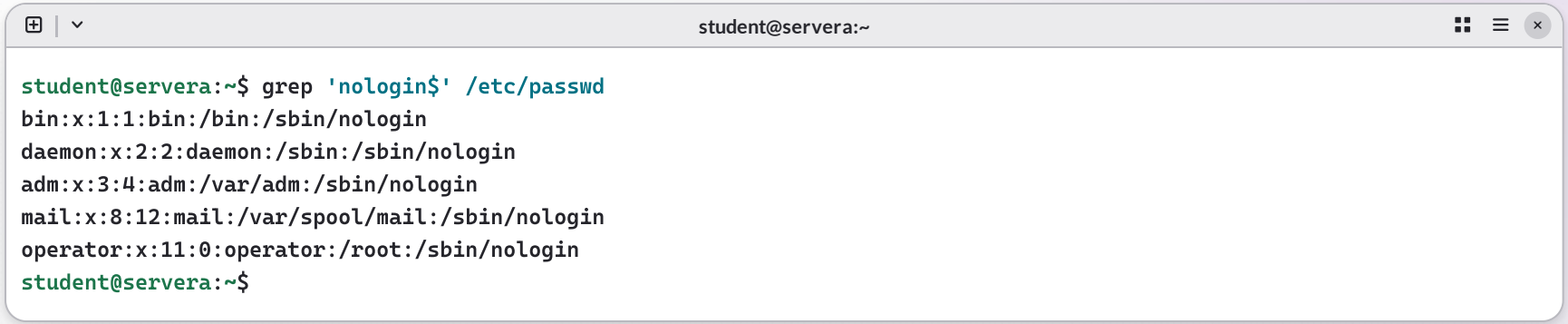
Without an anchor, `cat` matches anywhere on the line

student@servera:~

```
student@servera:~$ grep 'cat' /usr/share/dict/words
Muscat
bobcat
cat
catalog
duplicate
scatter
student@servera:~$ grep '^cat' /usr/share/dict/words
cat
catalog
student@servera:~$
```

Anchor to the End of a Line with `$`

Every account that cannot log in ends its line with `nologin`



The string `nologin` must be followed immediately by the end of the line

Both Anchors Together

Anchoring both ends means the line contains that string and **nothing else**

student@servera:~

```
student@servera:~$ grep '^cat$' /usr/share/dict/words
cat
student@servera:~$
```

So what does `^$` match?

The Empty Pattern `^$`

The start of the line, immediately followed by the end of the line: a **blank line**

Combine it with `-v` to strip the blank lines out of a configuration file

student@servera:~

```
1  student@servera:~$ grep -v '^$' /etc/ssh/sshd_config | head -n4
2  # To modify the system-wide sshd configuration, create a *.conf file under
3  #  /etc/ssh/sshd_config.d/ which will be automatically included below
4  Include /etc/ssh/sshd_config.d/*.conf
5  # If you want to change the port on a SELinux system, you have to tell
6  student@servera:~$
```

A Practical Use: Reading `ls -l`

The first character of a long listing is the file type

Searching for `d` without an anchor matches every line that happens to contain the letter

```
student@servera:~$ ls -l /etc | grep 'd'
drwxr-xr-x.  4 root root  4096 Jan  6 dconf
-rw-r--r--.  1 root root   483 Jan  6 gdbinit
lrwxrwxrwx.  1 root root    17 Jan  6 ld.so.conf
student@servera:~$
```

Anchoring to the start returns **only directories**

```
student@servera:~$ ls -l /etc | grep '^d'
drwxr-xr-x.  4 root root  4096 Jan  6 cups
drwxr-xr-x.  2 root root  4096 Jan  6 ssh
drwxr-xr-x.  6 root root  4096 Jan  6 yum
student@servera:~$
```

Escaping Inside the Pattern

`.` means "any character" to `grep`; to match a literal dot, escape it

student@servera:~

```
1  student@servera:~$ ls -l /etc | grep '\.conf$' | head -n3
2  -rw-r--r--.  1 root root  1362 Jan  6 dracut.conf
3  -rw-r--r--.  1 root root   969 Jan  6 libaudit.conf
4  -rw-r--r--.  1 root root   604 Jan  6 logrotate.conf
5  student@servera:~$
```

> **Warning**
>
> The single quotes protect the backslash from Bash so that `grep` receives `\.` intact

HANDS-ON EXERCISE

Exercise: Line Anchors

GOAL

Use line anchors to distinguish complete values from partial matches

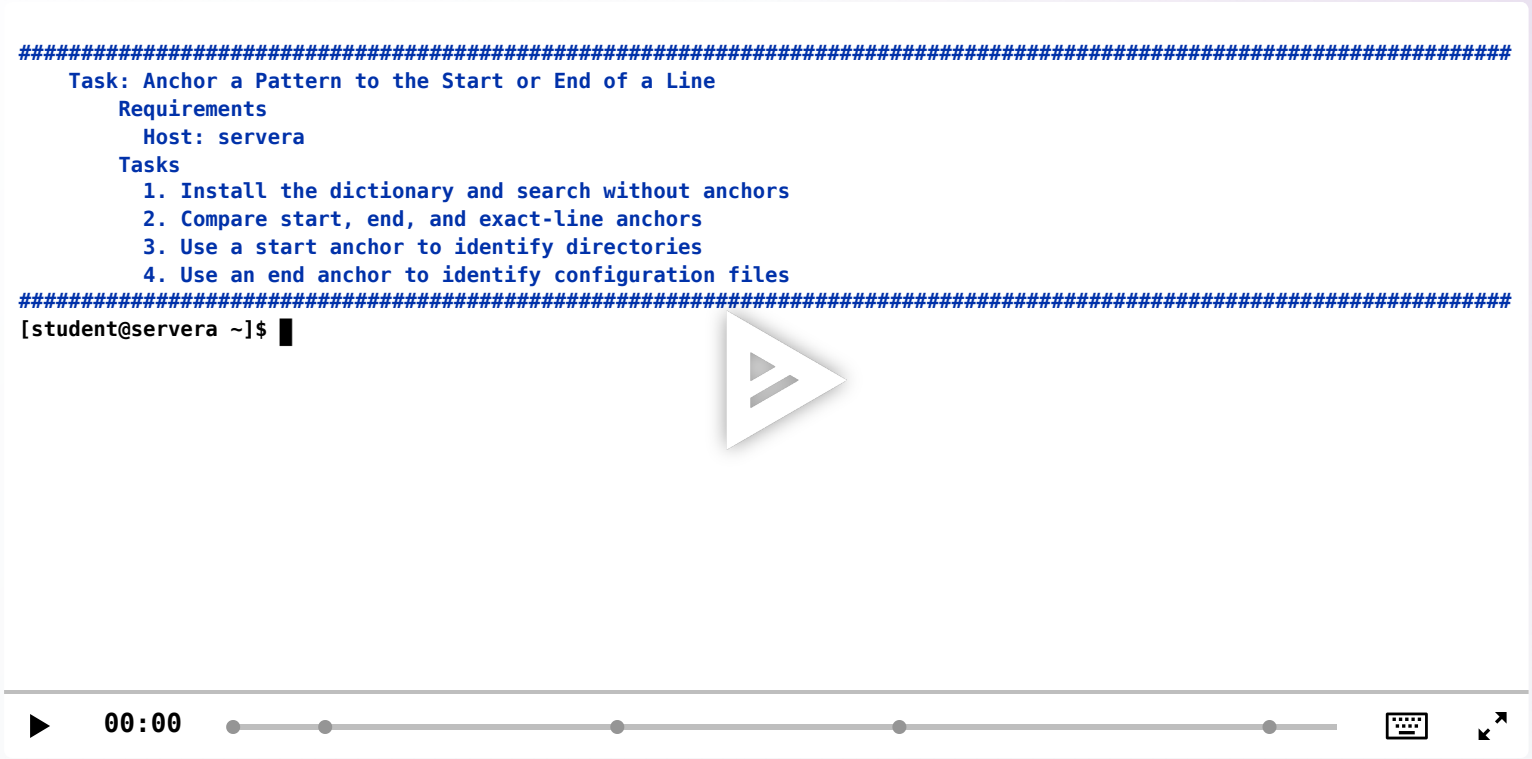
ENVIRONMENT

Host: `servera`

WORKFLOW

1. Install the dictionary and search without anchors
2. Compare start, end, and exact-line anchors
3. Use a start anchor to identify directories
4. Use an end anchor to identify configuration files

Exercise: Line Anchors



Written exercise

IT-230 · Linux Administration

Ranges and Quantifiers

Describing which characters, and how many

Ranges: One Character From a Set

`[...]` matches exactly **one** character out of everything listed inside

Range	Matches
`[a-z]`	any lowercase letter
`[A-Z]`	any uppercase letter
`[0-9]`	any digit
`[aeiou]`	any one of five vowels

You have seen these brackets already tonight

As a **glob**, `[ac]` matched one character of a file name

As a **regex**, it means the same thing: one character from the set

> **Note**
>
> Brackets are the one symbol that means the same thing in both

Ranges: Letters and Digits

Three-letter words that begin with `c` and end with `t`, then three digits in a row

student@servera:~

```
student@servera:~$ grep '^c[a-z]t$' /usr/share/dict/words
cat
cot
cut
student@servera:~$ sudo grep 'systemd\[[0-9][0-9][0-9]\]' /var/log/messages | head -n2
Aug 26 00:47:59 servera systemd[958]: Startup finished in 74ms.
Aug 26 00:47:59 servera systemd[958]: Reached target Main User Target.
student@servera:~$
```

Three brackets in a row means three characters, one match per bracket

The Wildcard `.` (Spot)

`.` matches **any single character**: letter, digit, symbol, or space

```
student@servera:~$ grep '^c.t$' /usr/share/dict/words
cat
cot
cut
student@servera:~$
```

Candidate	Result	Why
`cat`	match	one character between
`c@t`	match	a symbol is a character
`c t`	match	a space is a character
`coat`	no match	two characters between
`ct`	no match	no characters between

When You Do Not Know How Many

You cannot type one `.` per character when the length varies

`*` (Splat): the preceding item, zero or more times

`5[0-9]*` matches a `5` followed by any number of digits:

- `5` : zero digits after the five
- `50` : one digit
- `599` : two digits
- `52394723948729384` : as many as you like

Putting the Two Together

What does `.*` match?

- `.` is any character
- `*` is zero or more times

`.*` matches anything at all

Including nothing, the regex equivalent of a shrug

The Other Quantifiers

`?` (What): at most once

Makes the preceding item optional

`colou?r` matches `color` and `colour`

`+` (Intersection): one or more

`Go+gle` matches `Google` and `Gooooogle`, but **not** `Ggle`

`{n}` and `{n,m}` (Embrace): an exact count

- `Go{2}gle` : exactly two `o`
- `Go{1,2}gle` : one or two `o`
- `[0-9]{1,3}` : one to three digits

> **Note**
>
> `{1,3}` is how you say "an IP address octet" without typing three separate ranges

Quantifier Reference

Symbol	Extended only?	INTERCAL name	The preceding item is matched…
`.`	No	Spot	(any single character)
`*`	No	Splat	zero or more times
`?`	**Yes**	What	at most once
`+`	**Yes**	Intersection	one or more times
`{n}`	**Yes**	Embrace	exactly n times
`{n,m}`	**Yes**	Embrace	between n and m times

"Extended only" is the subject of the next slide

Globs Are Not Regular Expressions

You met these at the start of class, when **Bash** used them to name files; only the brackets kept their meaning

Symbol	As a glob, to Bash	As a regex, to `grep`
`*`	any run of characters	the preceding item, zero or more times
`?`	exactly one character	the preceding item, at most once
`[ab]`	one character from the set	one character from the set
`.`	an ordinary dot	any single character

This is why every pattern tonight has been wrapped in **single quotes**

Two Dialects

One of the most confusing things about regex is that there is more than one grammar

Basic (BRE)

The default for `grep` and `sed`

`?`, `+`, and `{}` are **ordinary text** until you escape them

```
1 grep 'colou\?r' /some/file
```

Extended (ERE)

What Python, JavaScript, and `grep -E` use

`?`, `+`, and `{}` are **special by default**

```
1 grep -E 'colou?r' /some/file
```

The Same Question, Three Commands

Command	Matches `color`	Matches `colour`	Matches the literal `colou?r`
`grep 'colou?r'`	no	no	yes
`grep 'colou\?r'`	yes	yes	no
`grep -E 'colou?r'`	yes	yes	no

Rule of thumb

Reach for `grep -E` whenever your pattern uses `?`, `+`, or `{}`

Groups: A Real SELinux Pattern

This is how you label a directory **and everything inside it**

So it matches `/website` on its own **and** `/website/css/style.css`

Print Only the Match with `-o`

`grep` normally prints the whole line; `-o` prints just the part that matched

student@servera:~

```
student@servera:~$ sudo grep 'Failed password' /var/log/secure \
> | grep -oE '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'
10.0.2.2
203.0.113.55
203.0.113.55
student@servera:~$ sudo grep 'Failed password' /var/log/secure \
> | grep -oE '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}' \
> | uniq -c
      1 10.0.2.2
      2 203.0.113.55
student@servera:~$
```

That is the report from the beginning of the chapter

HANDS-ON EXERCISE

Exercise: Ranges and Quantifiers

GOAL

Find structured values with regular-expression ranges and quantifiers

ENVIRONMENT

Host: `servera`

WORKFLOW

1. Seed and verify failed-login entries
2. Extract and count the IP addresses
3. Create and match a sample directory tree

Exercise: Ranges and Quantifiers

Written exercise