Lecture 03 - More shells 🐚🐚

ENSF461 - Applied Operating Systems

Instructor: Lorenzo De Carli, University of Calgary (*lorenzo.decarli@ucalgary.ca*)

Slides by Lorenzo De Carli, partly based on material by Robert Walls (WPI)

The plan for today

A bit more about the shell

- Review of **last lecture**
- An overview of UNIX **command-line utilities**
- An overview of **shell programming** in bash

Review

What is a shell?

Review

What is a shell?

- A shell is a program that presents a **prompt** and waits for **text commands**
- **Commands** are typically the names of executable programs
- Most commonly, the shell finds the **program** indicated by the **command**, **executes it**, and **displays its output**
- **Note:** every time you see a line beginning with “**$**” in my slides, it means what follows is a shell command

Review/2

What is an environment variable?

- A shell maintain some **metadata** that **alter its functioning**, or that of the programs that are executed

- This metadata is stored as **key-value pairs**:

```
VAR1_NAME = VAR1_VALUE
VAR2_NAME = VAR2_VALUE
…
```

- Each key-value pair represents a **variable** and its **value**

- Variables can be **read and/or modified** by the shell, programs, or the user

Review/3

What about working directories?

- Bash (and other shells) offer a **shorthand** for “run a command from the current working directory”
- Simply **prepend** the name of the command with “`./`“
- E.g. “`$ ./mycmd`” will look for a program called `mycmd` in the current directory and run it
- Print current working directory: `pwd`
- Change working directory: `$ cd path/to/new/working/directory`

Review/4

Do you remember what standard output/error/input are?

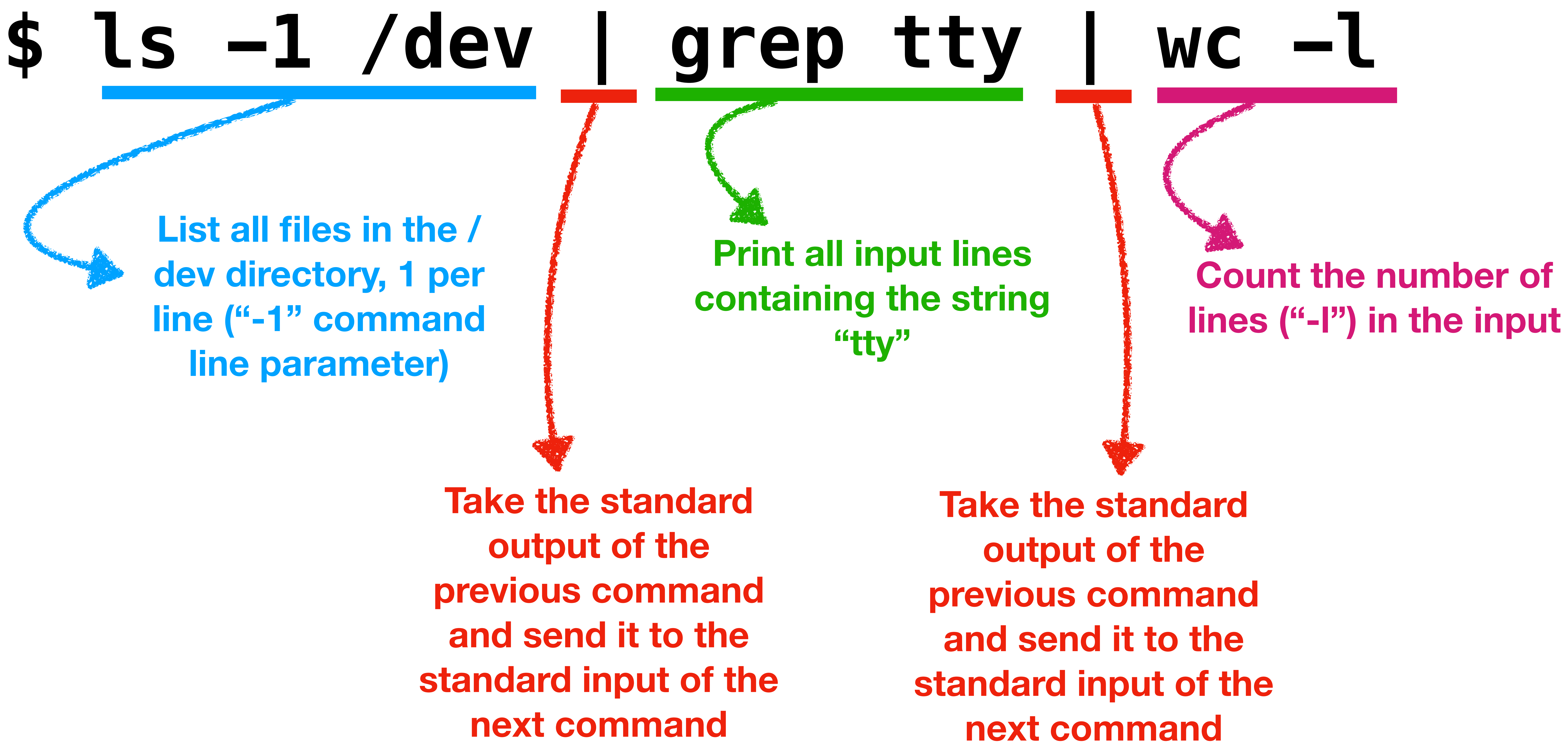
Review/4

Do you remember what standard output/error/input are?

- In a UNIX-like system, whenever a program starts it automatically opens **three file-like object**
- **Standard output:** an output stream which by is printed on the terminal
    - By convention, used for **regular output**
- **Standard error:** like standard input, but used for error messages
- **Standard input:** an input stream which receives input from the terminal

Review/5

Using stdin/out, it is possible to **pipe** commands



```
$ ls -1 /dev | grep tty | wc -l
```

List all files in the /dev directory, 1 per line ("-1" command line parameter)

Take the standard output of the previous command and send it to the standard input of the next command

Print all input lines containing the string "tty"

Take the standard output of the previous command and send it to the standard input of the next command

Count the number of lines ("-l") in the input

Let’s move on to command-line utilities 🖥️

Command-line utilities

- UNIX-like operating systems (Linux, MacOS) come with a large number of standard **command-line utilities** preinstalled
- We have already seen some examples:
  - **`ls`**: list files in a directory
  - **`grep`**: search for a string in each line of input
  - **`wc`**: counts the number of characters/words/lines in the input

Command-line utilities/2

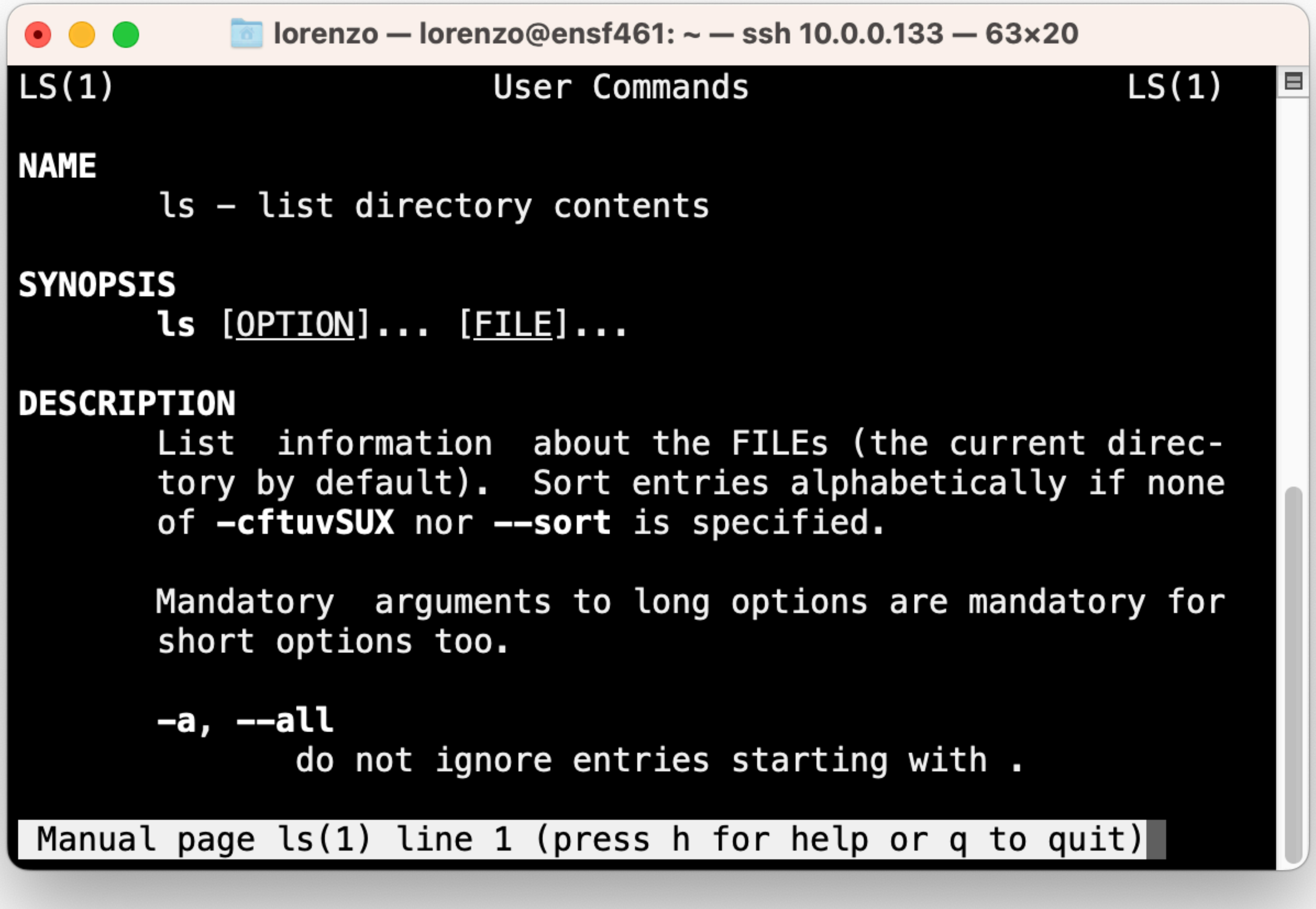
- Command-line utilities assist in **automating various tasks**:
  - **Text processing:** filtering and aggregating text data; computing statistics
  - **Operating on the file system:** changing working directory, creating and deleting files and folders
  - Show various machine statistics (e.g., free disk space)

Before we begin…

…a little help

- Typically, manual (**man**) pages are accessible, describing the syntax of various useful primitives:
  - **Command line utilities** and their options
  - Standard **C/C++ functions** (we'll look into those later)
  - Can be accessed with **`$ man <name of command>`**

Let’s see some examples



```
LS(1)                  User Commands                  LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about the FILEs (the current direc-
       tory by default).  Sort entries alphabetically if none
       of -cftuvSUX nor --sort is specified.

       Mandatory  arguments to long options are mandatory for
       short options too.

       -a, --all
              do not ignore entries starting with .

 Manual page ls(1) line 1 (press h for help or q to quit)
```

- Navigation: **`Page up/down`**
- Search: slash (“`/`“) followed by text
- Exit: **`q`**

Basic utilities

- **`echo <string>`**: print **`<string>`** on the terminal. Actually helpful! For example, to print the value of an environment variable (**`echo $PATH`**)
- **`cat <file>`**: print the content of **`<file>`** on the terminal
- Oftentimes their output is **piped into another program** for processing
- **Let’s see some examples**

More command line

The `grep utility`

- **`grep`** parses a file (or the standard input) and returns **all lines that match (or do not match) a string**
- A few options:
  - **`-i`**: makes matching case insensitive
  - **`-v`**: return all lines that do not match the pattern
  - **`-c`**: print number of matching lines per file
  - **`-l`**: print names of files with matches

More command line

The grep `utility`

- **`grep`** parses a file (or the standard input) and returns **all lines that match (or do not match) a string**
- A few options:
  - **`-i`**: makes matching case insensitive
  - **`-v`**: return all lines that do not match the pattern
  - **`-c`**: print number of matching lines per file
  - **`-l`**: print names of files with matches

} Useful when multiple files are passed as input **(what does it mean?)**

Multiple files?

Let’s talk about wildcards

- It is typically to be possible to run a utility on **more than one file** in one pass
- For example, use **`grep`** to find a string in **all files in a directory**
- This is accomplished using **wildcard patterns**:
  - **`$ grep hello *.c`**: find the string “hello” in all files ending in “**`.c`**” in the current directory
  - **`$ grep hello /home/lorenzo/prog*c`**: find the string “hello” in all files starting in “**`prog`**” and ending in “**`c`**” in the directory **`/home/lorenzo`**

Can it get even wilder?

More wildcard patterns

- Run “**`$ man 7 glob`**” and knock yourself out
  - (**Note #1:** the topic of wildcards is referred as “glob” because in early UNIX versions wildcard expansion was done by a utility called **`glob`**)
  - (**Note #2:** in this case we have to tell man that the page we want to display is in section 7 of the man pages. If you don’t, by default man displays the page for the **`glob()`** C function)
- If you want more info, visit https://tldp.org/LDP/GNU-Linux-Tools-Summary/html/x11655.htm

One more detour…

Regular expressions! (regexp for friends)

- The **`grep`** utility can match patterns based on **regular expressions**
- **What is a regular expression?**

One more detour…

Regular expressions! (regexp for friends)

- The **`grep`** utility can match patterns based on **regular expressions**
- **What is a regular expression?**
- **Definition:**
  - A **pattern** that describe a (possibly infinite) set of strings
  - Uses **special characters** to describe **sets of characters** within a string
  - Indeed, **grep** comes from a contraction of “Global search for Regular Expression and Print matching lines”

Basic regexp syntax

…all those can be combined!

- **`.`** : matches any character
- **`*`** : matches a sequence of any length (including 0) of the previous character
- **`+`** : matches a sequence of length 1 or above of the previous character
- **`?`** : matches 0 or 1 occurrences of the previous character
- **`[<set of characters>]`** : matches any character in a set
- **`[a-b]`**: any lowercase letter (also works for **`[A-B]`** and **`[0-9]`**)
- **(<pattern>|<another pattern>)** : OR between patterns

Anchoring and repetitions

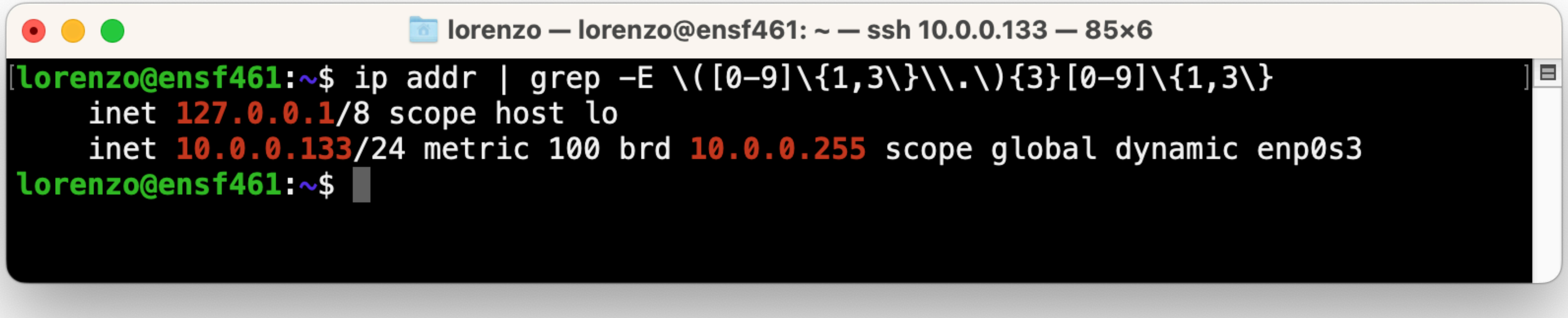
- **`^<regular expression>`** : **`<regular expression>`** must appear immediately after the beginning of a line
- **`<regular expression>$`** : **`<regular expression>`** must appear right before the end of a line
- **`{no}`** : matches preceding pattern exactly no times (e.g., **`(ab){10}`**)
- **`{min,}`** : matches preceding pattern at least **`min`** times (e.g., **`(a[0-9]+){4,}`**)
- **`{,max}`** : matches preceding pattern at least **`max`** times
- **`{min,max}`** : matches preceding pattern between **`min`** and **`max`** times

Some regexp examples

- **`hello.*`** : matches “hello” followed by any string of characters
- **`[0-9]+`** : matches any sequence of one or more digits
- **`file[a-z]?`** : matches “file” followed by 0 or 1 lowercase letters
- **`([0-9]+|hello.*)`** : matches any sequence of one or more digits OR “hello” followed by any string of characters

grep can match **regexp patterns**

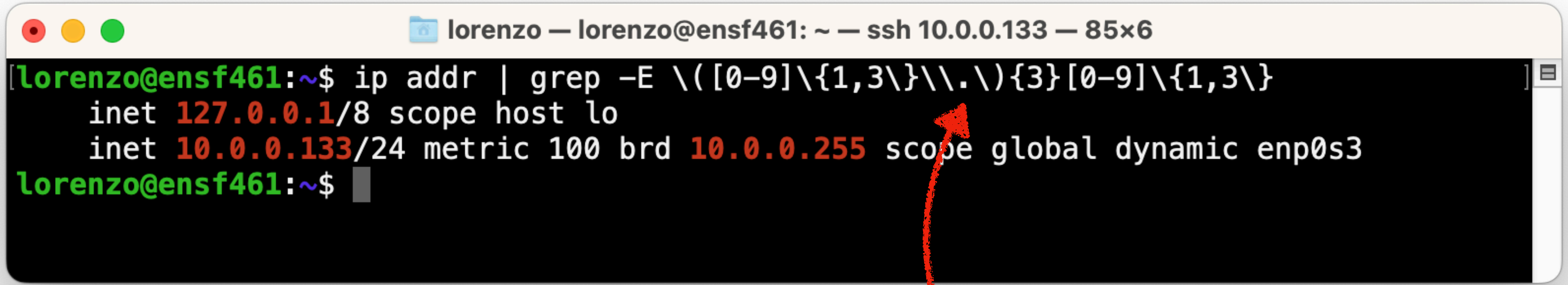
- **`grep -F <string>`** : interprets **`<string>`** as fixed string
- **`grep -E <string>`** : interprets **`<string>`** as a regular expression
- Example: **`$ echo "200" | grep -E [0-9]`**
- Example #2:



```
lorenzo@ensf461:~$ ip addr | grep -E \([0-9]\{1,3\}\\.\){3}[0-9]\{1,3\}
    inet 127.0.0.1/8 scope host lo
    inet 10.0.0.133/24 metric 100 brd 10.0.0.255 scope global dynamic enp0s3
lorenzo@ensf461:~$ 
```

grep can match **regexp patterns**

- **`grep -F <string>`** : interprets **`<string>`** as fixed string
- **`grep -E <string>`** : interprets **`<string>`** as a regular expression
- Example: **`$ echo "200" | grep -E [0-9]`**
- Example #2:

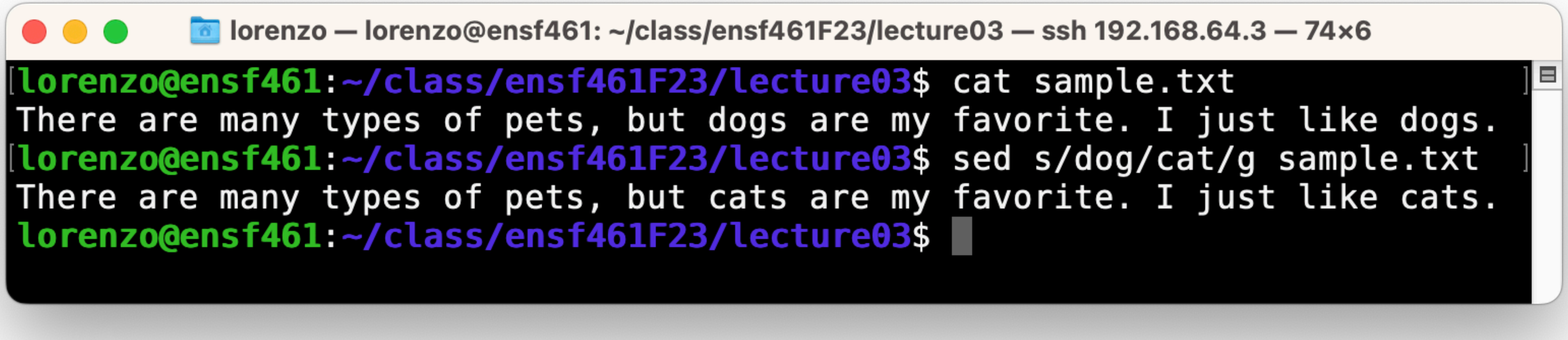


```
lorenzo@ensf461:~$ ip addr | grep -E \([0-9]\{1,3\}\\.\){3}[0-9]\{1,3\}
    inet 127.0.0.1/8 scope host lo
    inet 10.0.0.133/24 metric 100 brd 10.0.0.255 scope global dynamic enp0s3
lorenzo@ensf461:~$
```

Note the double backlash (escape), to tell grep that "." is to be interpreted as a literal dot, not a special character

A few more utilities

- **sed:** typically used for string replacement in files or standard input (receives a string or a regular expression describing the text to be modified)

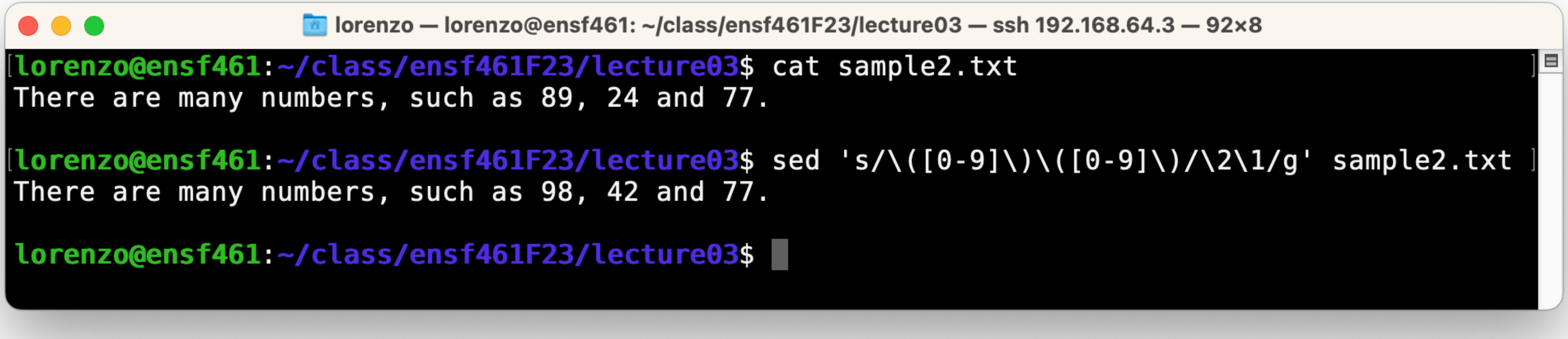


lorenzo — lorenzo@ensf461: ~/class/ensf461F23/lecture03 — ssh 192.168.64.3 — 74×6

```
lorenzo@ensf461:~/class/ensf461F23/lecture03$ cat sample.txt
There are many types of pets, but dogs are my favorite. I just like dogs.
lorenzo@ensf461:~/class/ensf461F23/lecture03$ sed s/dog/cat/g sample.txt
There are many types of pets, but cats are my favorite. I just like cats.
lorenzo@ensf461:~/class/ensf461F23/lecture03$ 
```

More on sed

- **sed** can replace patterns described by regular expressions, using **capture groups**
  - Implicitly, anything delimited by **(   )** in a regex is a group

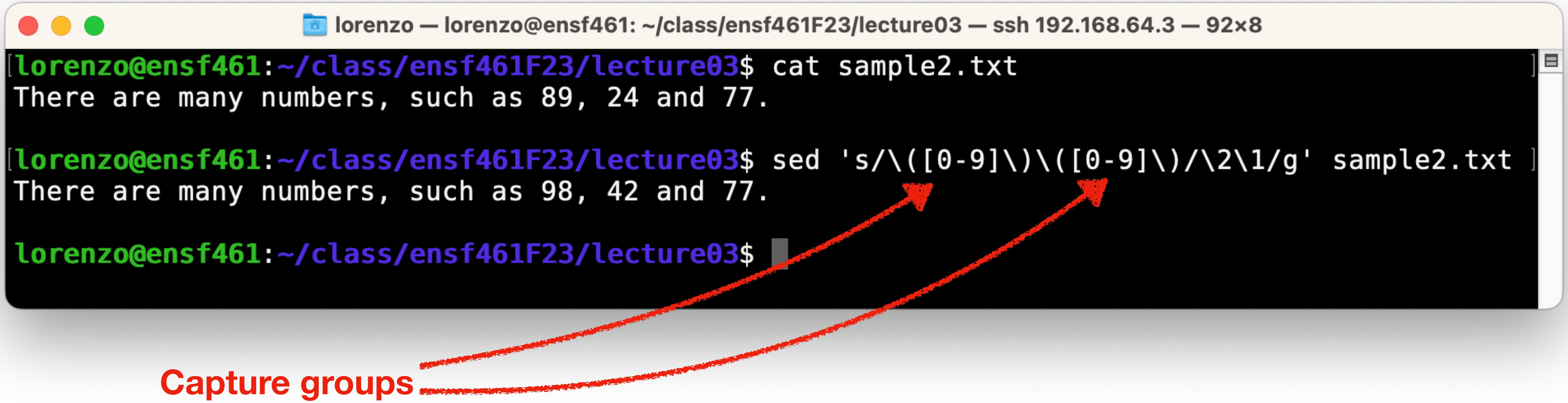


lorenzo — lorenzo@ensf461: ~/class/ensf461F23/lecture03 — ssh 192.168.64.3 — 92×8

```
lorenzo@ensf461:~/class/ensf461F23/lecture03$ cat sample2.txt
There are many numbers, such as 89, 24 and 77.

lorenzo@ensf461:~/class/ensf461F23/lecture03$ sed 's/\([0-9]\)\([0-9]\)/\2\1/g' sample2.txt
There are many numbers, such as 98, 42 and 77.

lorenzo@ensf461:~/class/ensf461F23/lecture03$ 
```

More on sed

- **sed** can replace patterns described by regular expressions, using **capture groups**
  - Implicitly, anything delimited by **( )** in a regex is a group



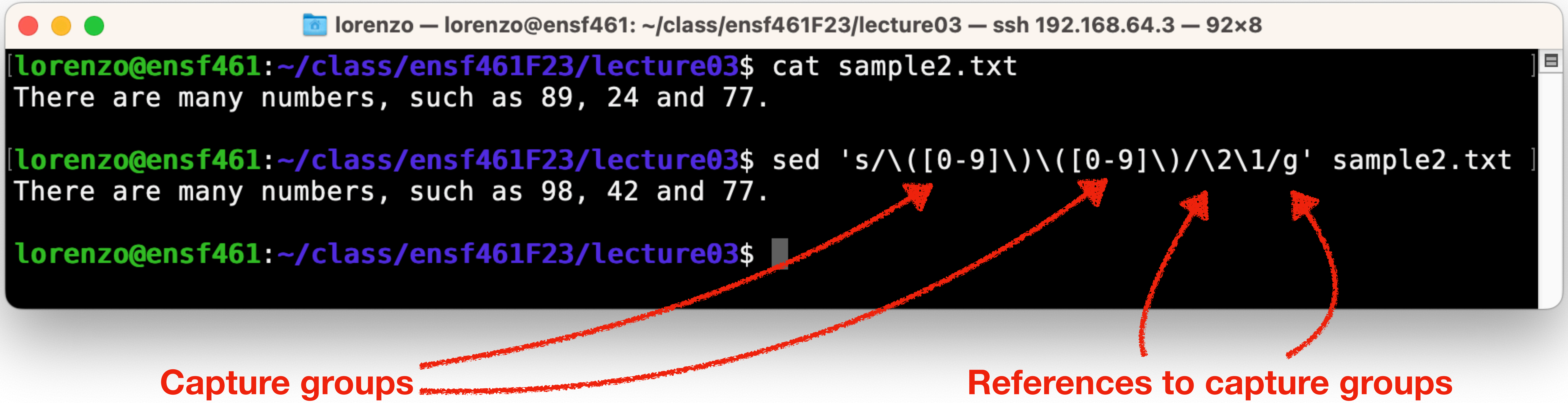
lorenzo — lorenzo@ensf461: ~/class/ensf461F23/lecture03 — ssh 192.168.64.3 — 92×8

```
[lorenzo@ensf461:~/class/ensf461F23/lecture03$ cat sample2.txt                               ]
There are many numbers, such as 89, 24 and 77.

[lorenzo@ensf461:~/class/ensf461F23/lecture03$ sed 's/\([0-9]\)\([0-9]\)/\2\1/g' sample2.txt ]
There are many numbers, such as 98, 42 and 77.

lorenzo@ensf461:~/class/ensf461F23/lecture03$ 
```

Capture groups

More on sed

- **sed** can replace patterns described by regular expressions, using **capture groups**
  - Implicitly, anything delimited by **( )** in a regex is a group



lorenzo — lorenzo@ensf461: ~/class/ensf461F23/lecture03 — ssh 192.168.64.3 — 92×8

```
lorenzo@ensf461:~/class/ensf461F23/lecture03$ cat sample2.txt
There are many numbers, such as 89, 24 and 77.

lorenzo@ensf461:~/class/ensf461F23/lecture03$ sed 's/\([0-9]\)\([0-9]\)/\2\1/g' sample2.txt
There are many numbers, such as 98, 42 and 77.

lorenzo@ensf461:~/class/ensf461F23/lecture03$ 
```

Capture groups

References to capture groups

More text processing

- `cut`: extract fields from content with separators
- `rev`: inverts text
- `awk`: parse/process text (quite a powerful tool; will not discuss it here)
- `sort:` sort all input lines numerically or alphabetically
- `uniq:` remove all duplicate lines
- **Let’s see some examples…**

Let’s take a break and quiz!

Navigate to D2L->Quizzes->Quiz 3

File system operations

- **`ls`**: list files in a directory
- **`type`**: print path of program
- **`cd`**: change working directory
- **`pwd`**: print working directory
- **`cp:`** copy file
- **`mv:`** move/rename file
- **`rm`**: delete file/directory (can also use rmdir for the latter)
- **`touch`**: create empty file (**why**)?
- **`du`**: display size of files

Let’s move on to shell programming

Shell programming

Why?

- Most shells support (relatively) simple **programming syntax**
- Typically used to write **shell scripts** to automate **various tasks**
- **Examples:**
  - Replace a string of text across all files in a directory
  - Delete files with a certain string in the name
  - List all files sorted by size
  - …

My first shell program

```
#!/bin/bash

echo "Hello, world!”
```

A few things to unpack here…

- What’s this “/bin/bash” business?
- **How do I run this?**

Running stuff in the shell

- Typically, a shell is able to run **two types of programs**:
  - “Proper” programs, consisting of **executable files**
    - Will talk about this at length in this class!
  - **Interpreted scripts**
    - Those are just a bunch of text which gets passed as input to an **interpreter**
    - Regardless, they are all run with **`$ ./program_name`**

Let’s talk more about interpreted scripts?

How does the shell know where to find the interpreter

- There are many **interpreted languages**
- Most shells (e.g., bash) can **run scripts**
- **Other interpreted languages?**

Let’s talk more about interpreted scripts?

How does the shell know where to find the interpreter

- There are many **interpreted languages**
- Most shells (e.g., bash) can **run scripts**
- **Other interpreted languages?**
    - python
    - node
    - perl
    - …

Let’s look again at the script

```
#!/bin/bash

echo "Hello, world!”
```

Let’s look again at the script

```
#!/bin/bash

echo "Hello, world!”
```

This header right here has the following format:

1. “**#!**”: tells bash **this is a script**
2. **“/bin/bash**”: tells bash **where to find the interpreter** for the script

bash will treat anything starting with “#!” as a script!

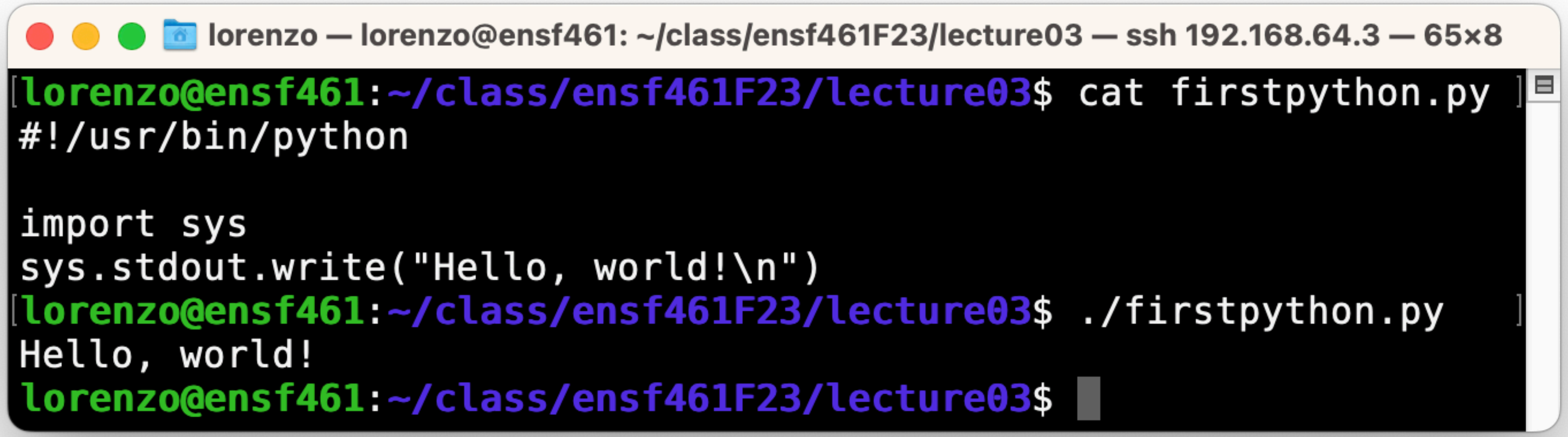
Let’s look again at the script

```
#!/bin/bash

echo "Hello, world!”
```

This header right here has the following format:

1. “**#!**”: tells bash **this is a script**
2. “**/bin/bash**”: tells bash **where to find the interpreter** for the script

bash will treat anything starting with “#!” as a script!


lorenzo — lorenzo@ensf461: ~/class/ensf461F23/lecture03 — ssh 192.168.64.3 — 65×8

```
lorenzo@ensf461:~/class/ensf461F23/lecture03$ cat firstpython.py
#!/usr/bin/python

import sys
sys.stdout.write("Hello, world!\n")
lorenzo@ensf461:~/class/ensf461F23/lecture03$ ./firstpython.py
Hello, world!
lorenzo@ensf461:~/class/ensf461F23/lecture03$
```

Note: works for most interpreters

Can I just write a script and execute it then?

Well, there is one more thing

- UNIX systems (like most OS'es) have **permissions**
- At the very least, any UNIX system supports the following:
  - Each file has t**hree type of permissions: read, write, execute**
  - Permissions are specified for **three different entities**:
    - **User**
    - **Group**
    - **All users**

Scripts must be given permission to execute

- Suppose you have created a script named `hello.sh` and you want to run it
- First, you must tell the system that **all users** have the **execute permission** for **hello.sh** (only need to do it once)
- You use chmod for this: `$ chmod a+x hello.sh`

Scripts must be given permission to execute

- Suppose you have created a script named **`hello.sh`** and you want to run it
- First, you must tell the system that **all users** have the **execute permission** for **hello.sh** (only need to do it once)
- You use chmod for this: **`$ chmod a+x hello.sh`**

Change permissions:

Scripts must be given permission to execute

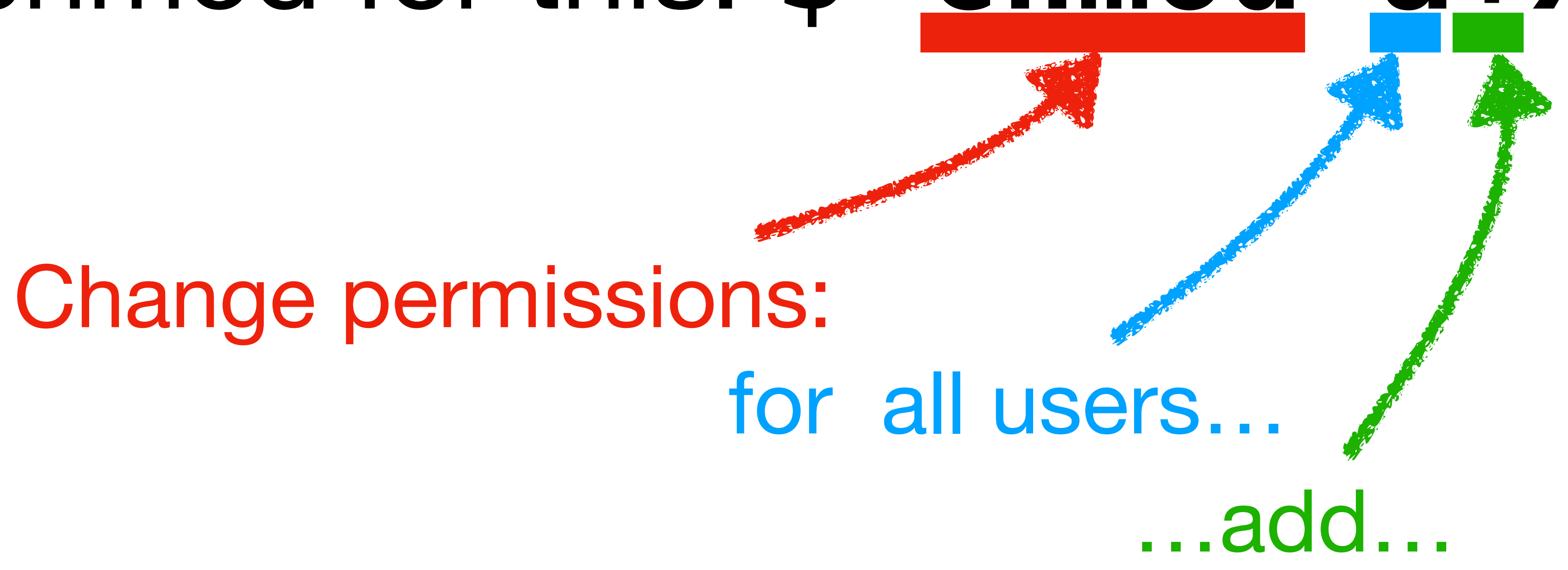
- Suppose you have created a script named **`hello.sh`** and you want to run it
- First, you must tell the system that **all users** have the **execute permission** for **hello.sh** (only need to do it once)
- You use chmod for this: **`$ chmod a+x hello.sh`**

Change permissions:

for all users…

Scripts must be given permission to execute

- Suppose you have created a script named **`hello.sh`** and you want to run it
- First, you must tell the system that **all users** have the **execute permission** for **hello.sh** (only need to do it once)
- You use chmod for this: **`$ chmod a+x hello.sh`**



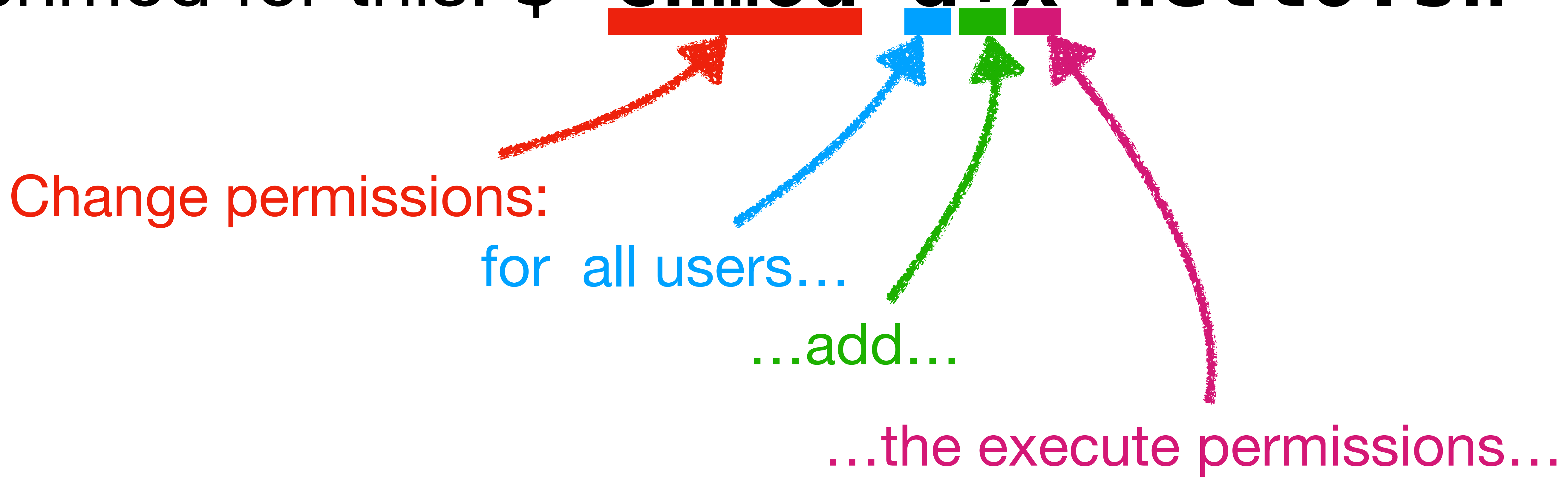
Change permissions:

for all users…

…add…

Scripts must be given permission to execute

- Suppose you have created a script named **`hello.sh`** and you want to run it
- First, you must tell the system that **all users** have the **execute permission** for **hello.sh** (only need to do it once)
- You use chmod for this: **`$ chmod a+x hello.sh`**



Change permissions:

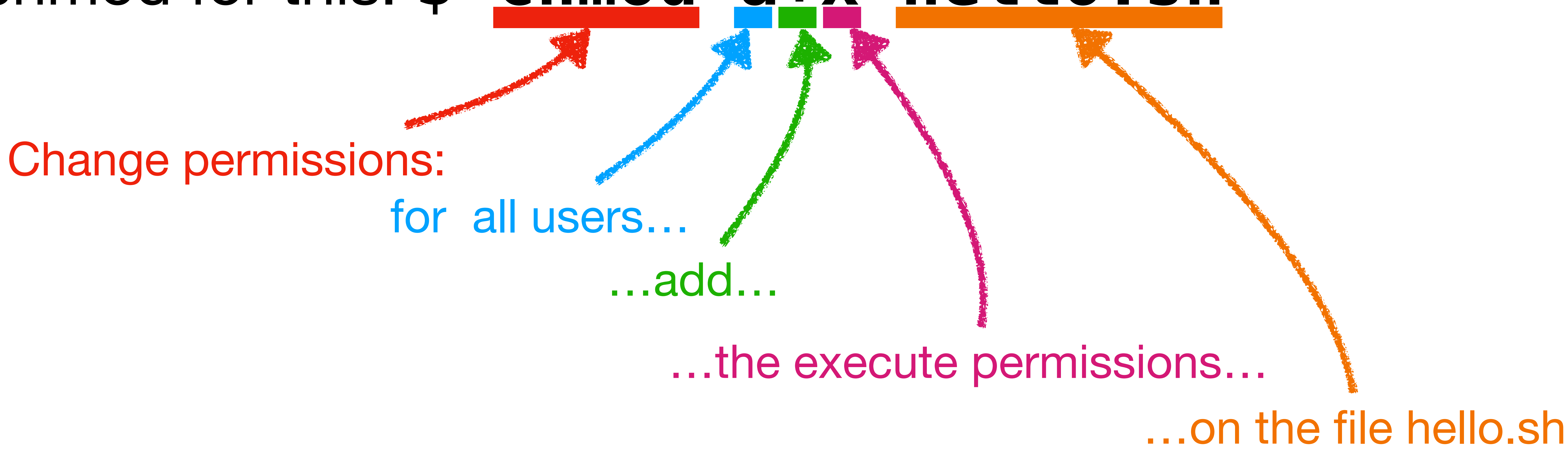
for all users…

…add…

…the execute permissions…

Scripts must be given permission to execute

- Suppose you have created a script named **`hello.sh`** and you want to run it
- First, you must tell the system that **all users** have the **execute permission** for **hello.sh** (only need to do it once)
- You use chmod for this: **`$ chmod a+x hello.sh`**



Change permissions:

for all users…

…add…

…the execute permissions…

…on the file hello.sh

Summarizing…

- Create your script (e.g., **`hello.sh`**)
- Add **`#!/bin/bash`** at the beginning
- **`$ chmod a+x hello.sh`**
- **`$ ./hello.sh`**
- Note: can also pass the script to bash explicitly: **`$ bash hello.sh`**

More bash programming

- **Bash scripts** are not limited to lists of commands
- Can include **programming constructs**
  - **Variables** (use environment variables)
  - **Arithmetic operations** (using the **`expr`** utility)
  - **Conditionals** (if/then/else)
  - **Loops** (while/for)
- There is more, but this is enough to cover most use cases

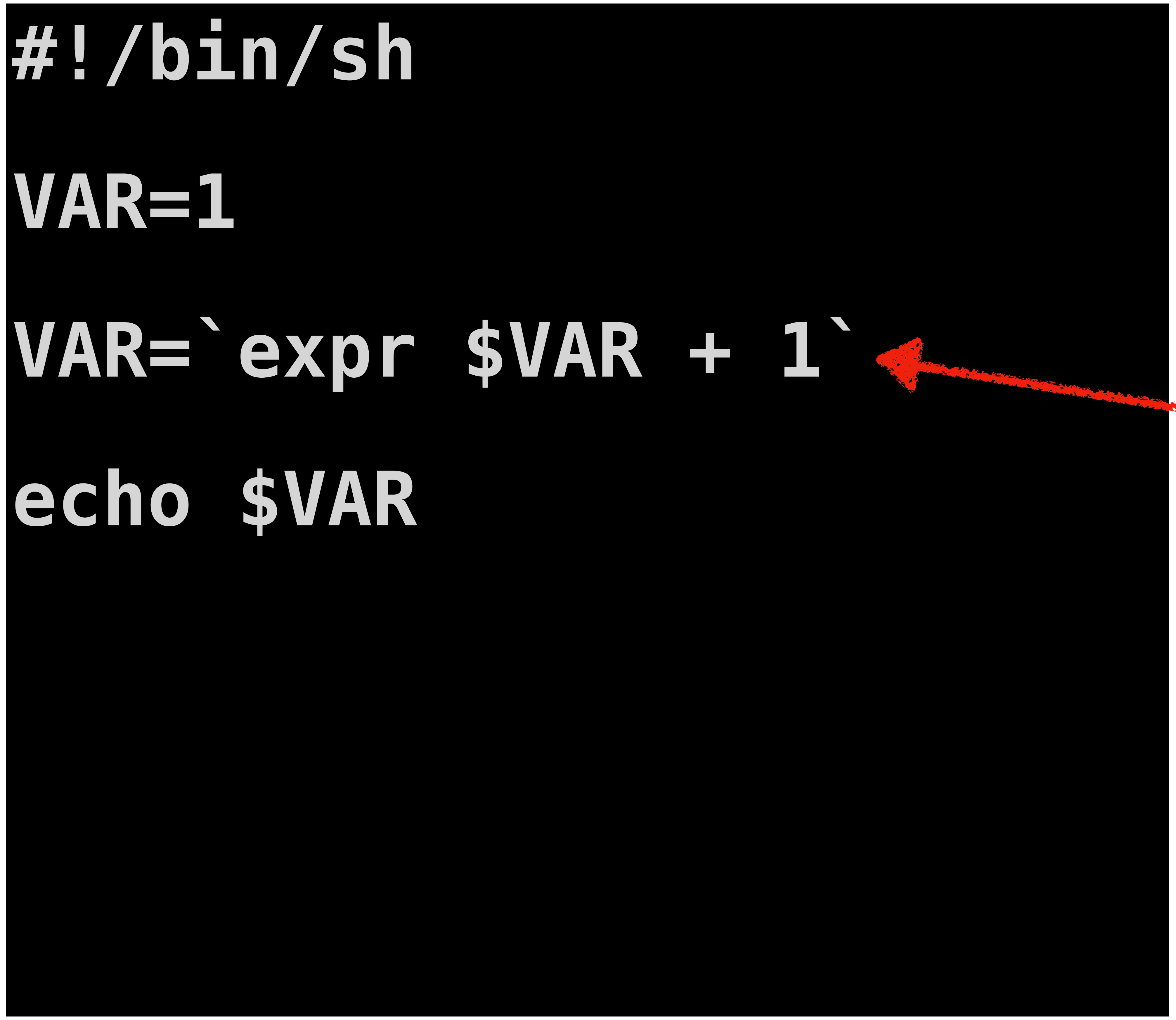
Bash variable example

```
#!/bin/sh

VAR=1

echo $VAR
```

Arithmetic expressions

```
#!/bin/sh

VAR=1

VAR=`expr $VAR + 1`

echo $VAR
```

Arithmetic expressions



```
#!/bin/sh

VAR=1

VAR=`expr $VAR + 1`

echo $VAR
```

What’s going on here? Let’s discuss it!

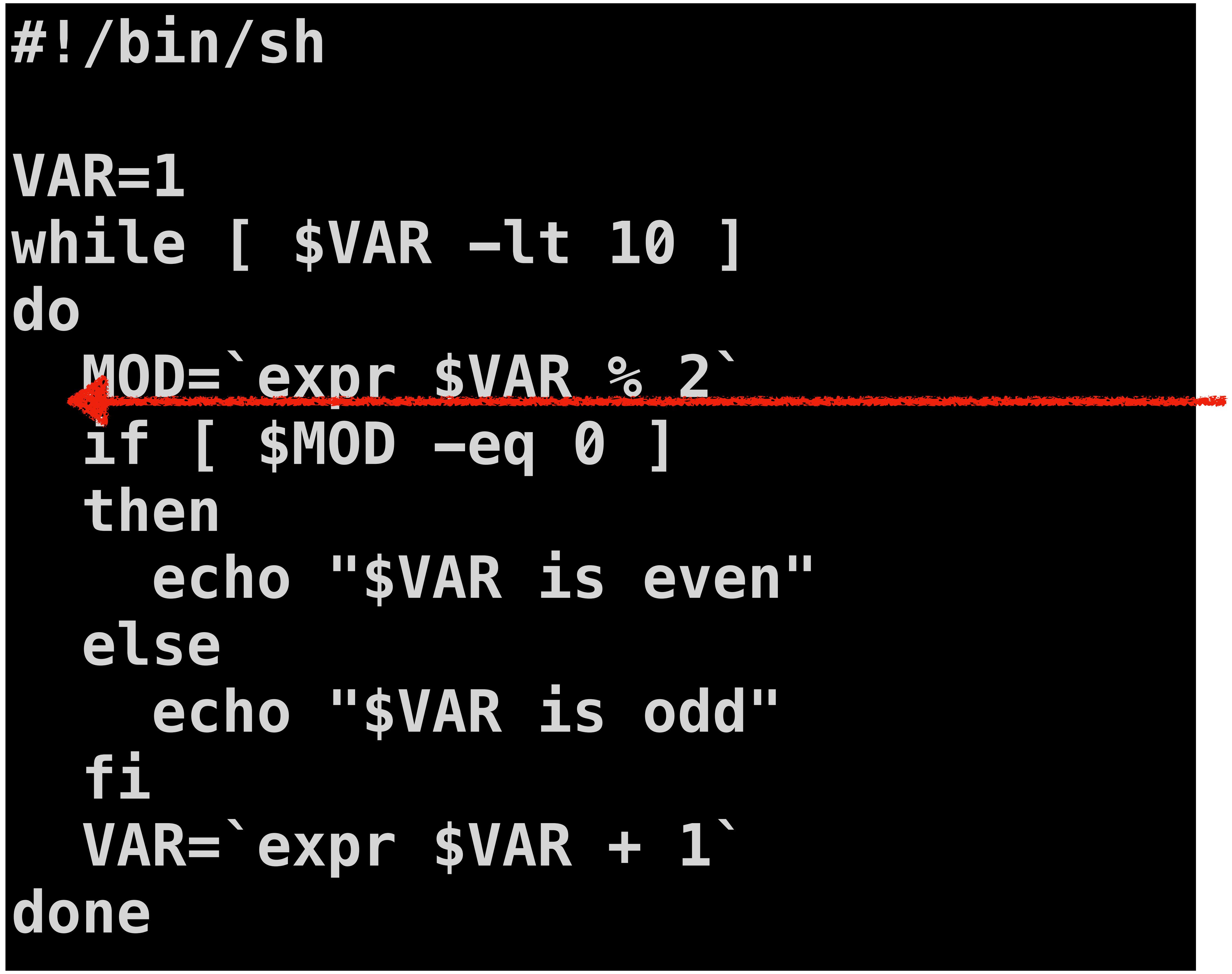
Conditionals

```
#!/bin/sh

VAR=1
VAR=`expr $VAR + 1`
MOD=`expr $VAR % 2`
if [ $MOD –eq 0 ]
then
  echo "Even"
else
  echo "Odd"
fi
```

Loops

```
#!/bin/sh

VAR=1
while [ $VAR –lt 10 ]
do
  MOD=`expr $VAR % 2`
  if [ $MOD –eq 0 ]
  then
    echo "$VAR is even"
  else
    echo "$VAR is odd"
  fi
  VAR=`expr $VAR + 1`
done
```

Loops



```
#!/bin/sh

VAR=1
while [ $VAR -lt 10 ]
do
  MOD=`expr $VAR % 2`
  if [ $MOD -eq 0 ]
  then
    echo "$VAR is even"
  else
    echo "$VAR is odd"
  fi
  VAR=`expr $VAR + 1`
done
```

Note, indentation is not required (this is not Python!). I just added it for clarity

More loops

Iterate over a list (similar to Python)

```
#!/bin/sh

for i in `ls –1 /dev`
do
  echo $i
done

for i in 1 2 3 4 5
do
  echo `expr $i + 1`
done
```

That’s all for today!