Introduction to Linux

Logistics

Class time: 9:00-4:00 Pacific
Lunch 12:00-12:40 Pacific
Breaks as needed

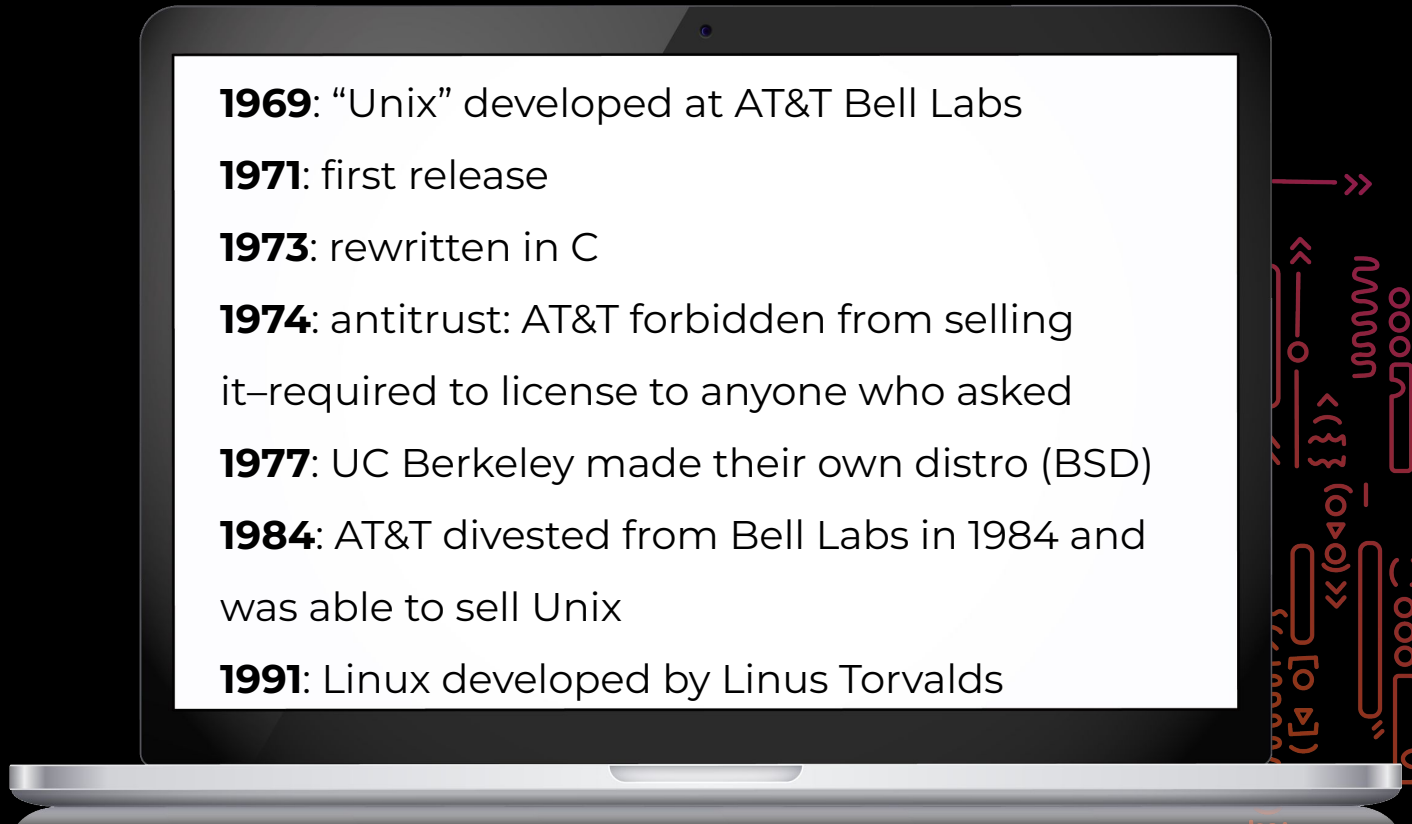
Introductions

- Who are you?
- What do you do?
- What do you want to get out of the course?
- Linux experience?
- Programming experience?
- Fun fact or perhaps a TV show you binge watched during quarantine

Course Goals

- Understand Linux mindset
- Understand Linux commands
- Gain facility at the Bash prompt
- Write Bash scripts
- Be more productive!

Brief History of Linux



1969: “Unix” developed at AT&T Bell Labs

1971: first release

1973: rewritten in C

1974: antitrust: AT&T forbidden from selling it–required to license to anyone who asked

1977: UC Berkeley made their own distro (BSD)

1984: AT&T divested from Bell Labs in 1984 and was able to sell Unix

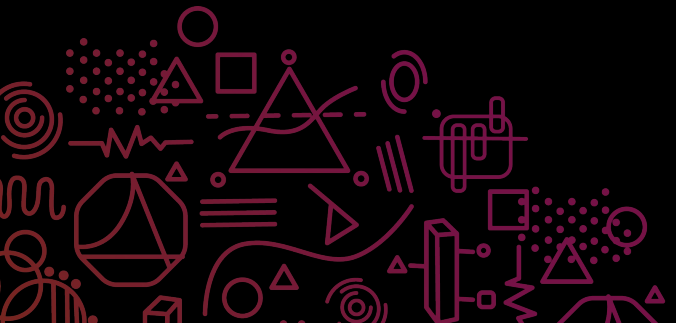
1991: Linux developed by Linus Torvalds

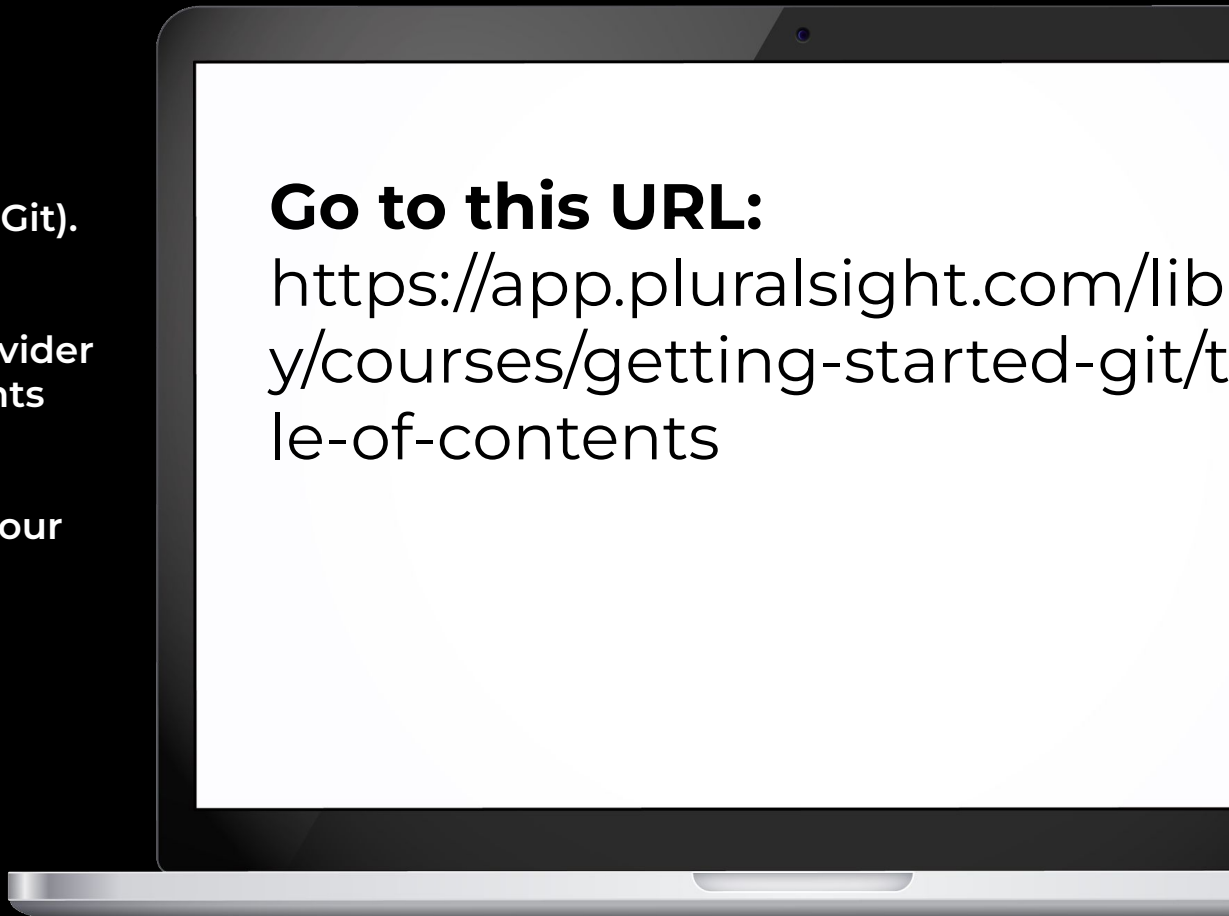
Let’s get started!

We will use a Linux sandbox borrowed from another course (Git).

(You may know that Pluralsight recently acquired next.tech, provider of cloud computing environments for teaching tech skills.)

Enter the link on the screen in your browser...



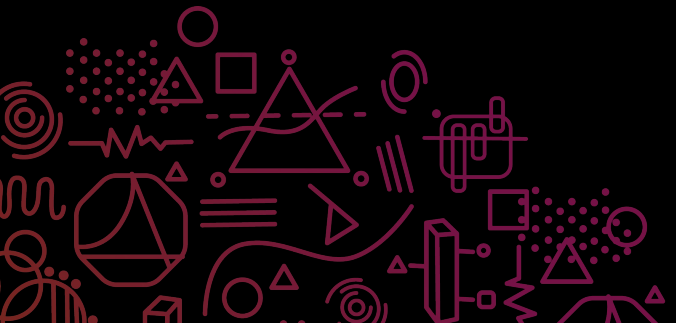


Command line/shell

What is a/the shell?

- program that accepts commands and runs them either by passing them to the operating system, or by executing them as "built-ins"

-also a command interpreter / scripting language



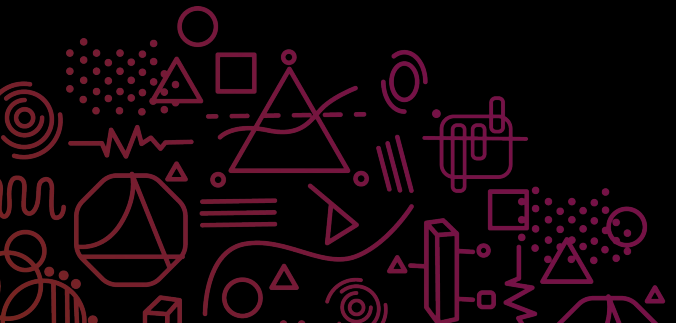
1977: sh (Bourne shell)
...developed by Stephen Bourne at Bell Labs
1978: csh (C shell)
...developed by Bill Joy at UC Berkeley
1983: tcsh (TENEX shell)... what Linux distros typically call *csh*
1989: bash (Bourne-again shell)
...what Linux distros typically call *sh*

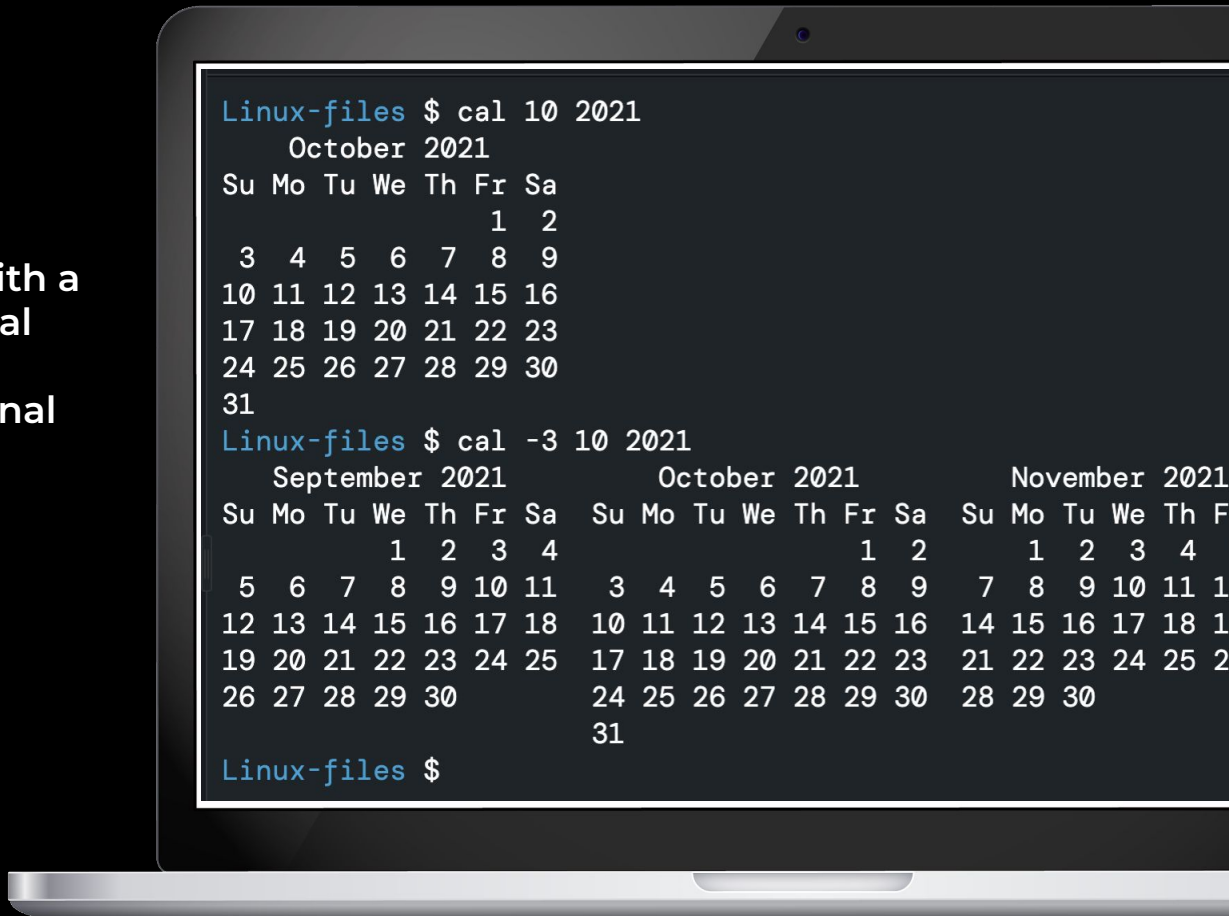
Command syntax

command options arguments

options, which typically begin with a dash, are (unsurprisingly) optional

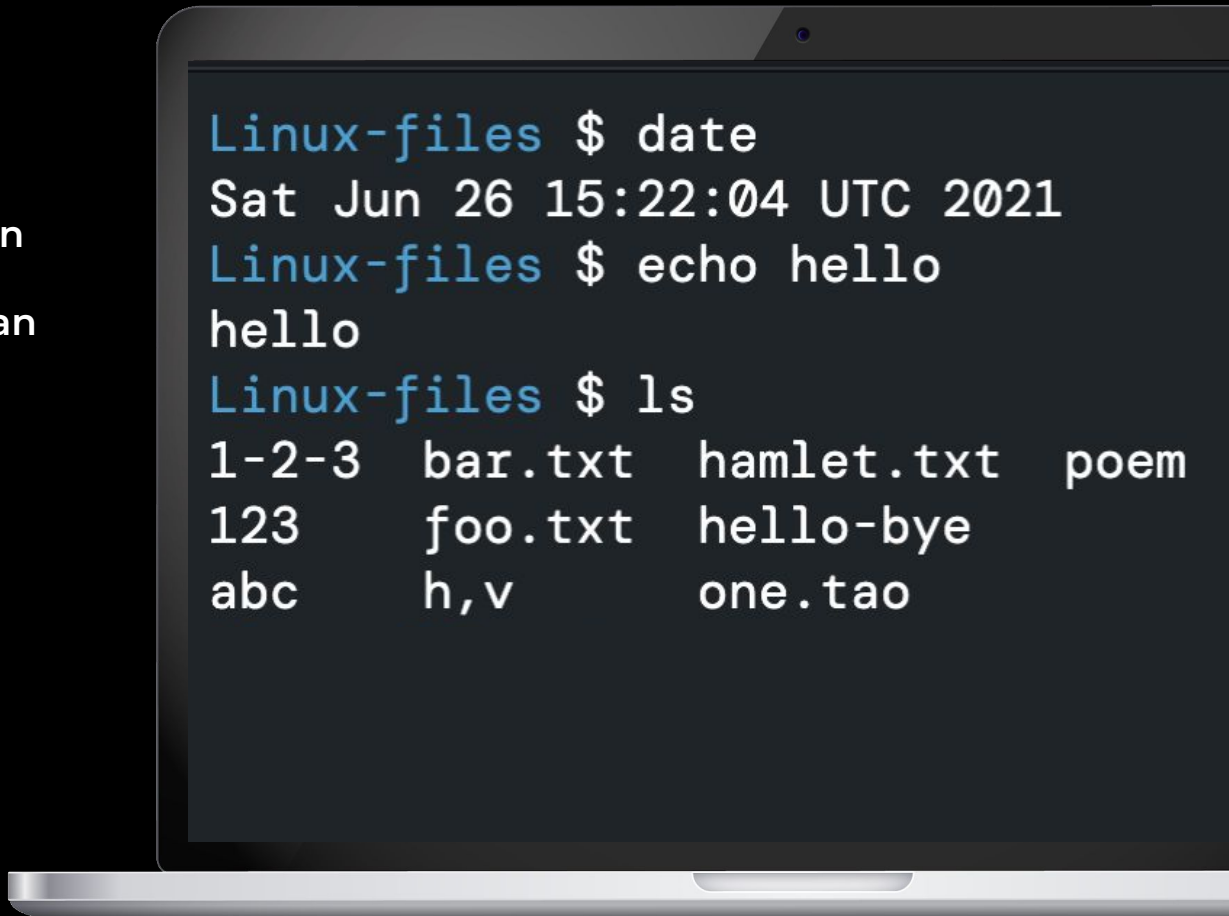
arguments are sometimes optional too





A few basic Linux commands...

date - get the date and time
echo - prints output to the screen
ls - lists files
man - get the manual entry ("man page") for a command

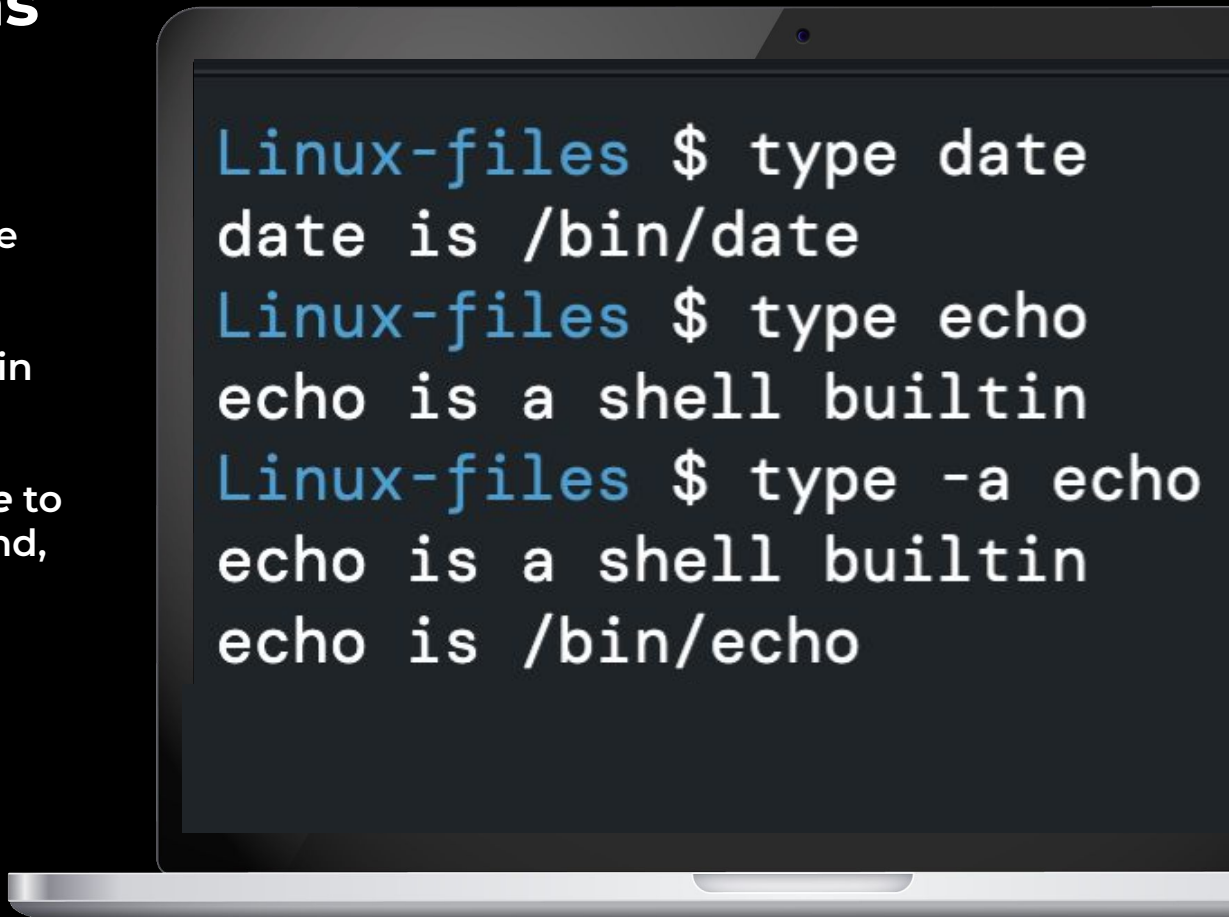


Commands: builtins vs. standalone

when you type in the name of a command, it's often a standalone program run by Bash

...but bash has a number of builtin commands too

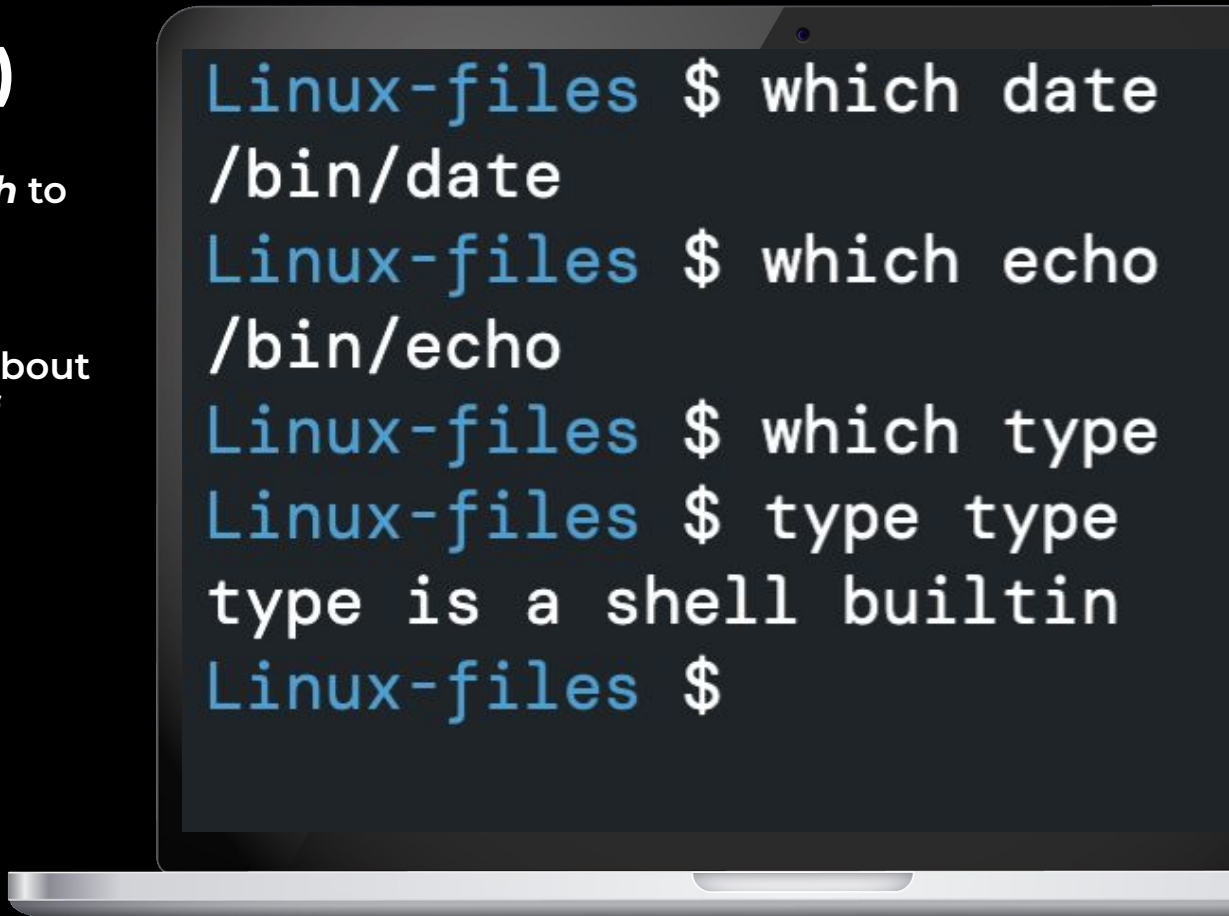
you can use the shell builtin ***type*** to determine the type of a command, either builtin or external



builtins vs. standalone (cont'd)

you can use the command ***which*** to find the full path of an external command

but it doesn't tell you anything about *shell builtins*, so you're better off using ***type***



Getting Help

- ***man*** - get the Linux manual entry ("man page") for a standalone command
- superseded by ***info***, but because ***man*** still exists, many people (myself included) still use it
- ***help*** - for getting information about bash builtins (doesn't work for standalone commands–try it)

Lab: Basic Commands

1. try the commands we've learned so far: ***date***, ***cal***, ***echo***, ***man***, ***type***
2. use ***man*** to learn about the commands, and try various options/arguments
3. what happens if you mistype the name of a command?

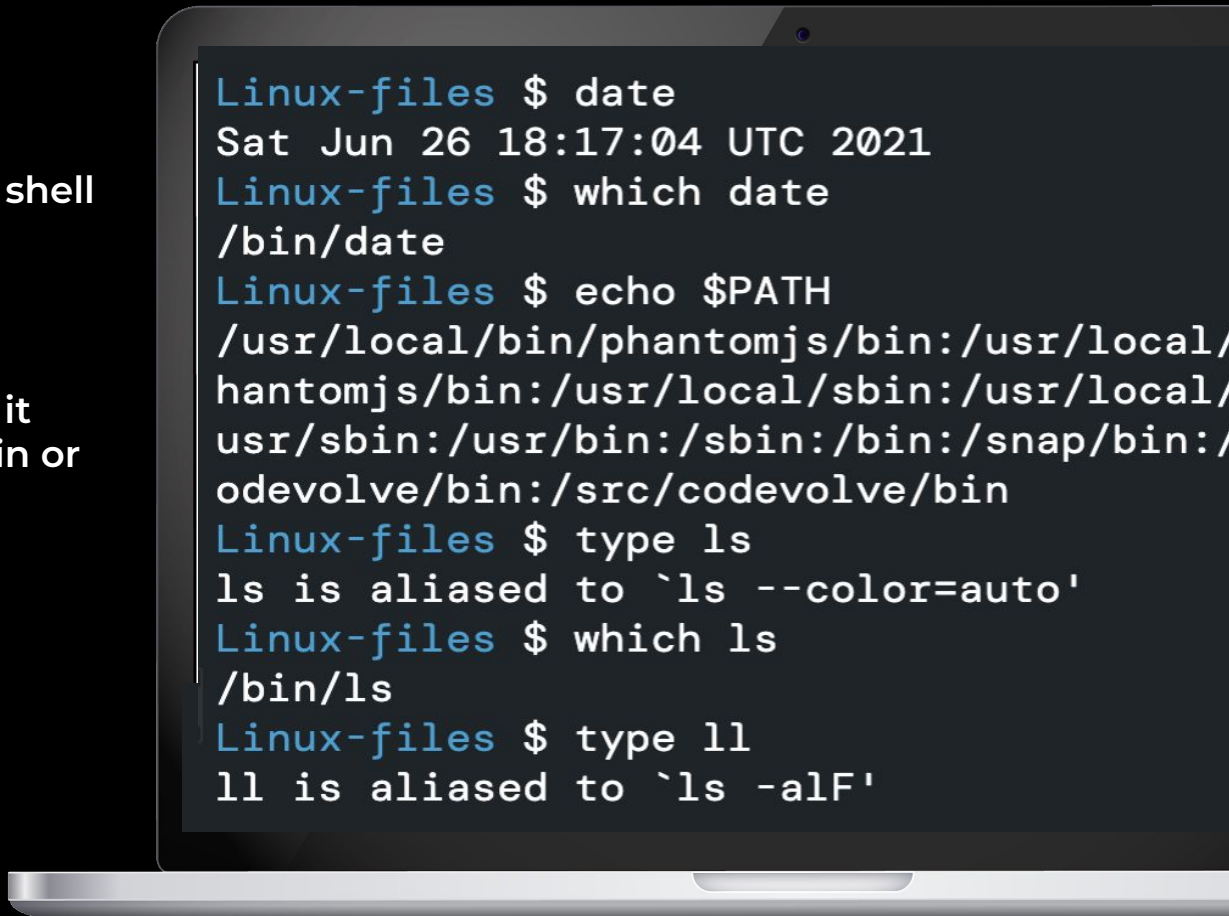
How does the shell find commands?

When you type a command, the shell looks thru a list of dirs to find it

This list can be found in ***$PATH***

If command is not in your **PATH**, it can't be found, unless it's a builtin or an *alias*

you can modify **PATH** ... later



The Linux Filesystem

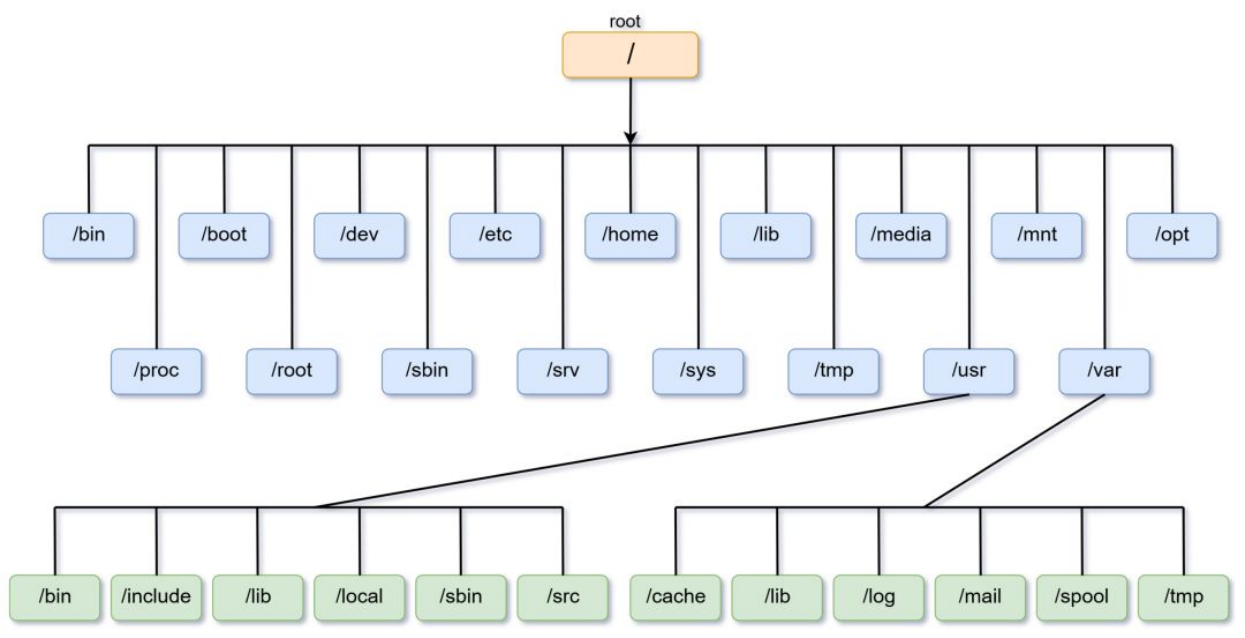
The Linux Filesystem

- hierarchical directory structure
- tree-like pattern of directories (or folders), which may contain files and other directories
- top level directory is denoted **/** and is called the root directory
- unlike Windows, which has a separate filesystem tree for each storage device, Linux has a single filesystem tree, regardless of how many drives or devices are attached
- devices are attached (or more correctly, *mounted*) in the tree, typically in **/dev** directory

The Linux Filesystem (cont'd)

- the command line has no graphical view so we'll have to think of the filesystem differently...
- ...imagine filesystem as upside-down tree and we're able to stand in the middle of it
- at any given time, we're inside a dir(ectory) and we can see the files contained in the dir and the pathway to the dir above us (called the parent dir) and any subdir(ectorie)s below us
- The dir we're in is the *current working directory*

The Linux Filesystem (cont'd)



Directories Common to Linux (pretty much)

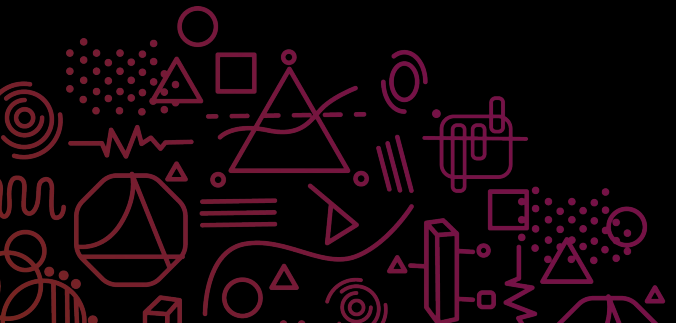
/ = root dir
/bin = binaries (i.e., programs) which must be present for Linux to boot and run
/dev = special dir that contains special files which represent the physical devices (disks, terminals, etc.)
/etc = contains systemwide configuration files
/home = home directories (may be elsewhere)
/proc = special files that give you a “window” into OS
/tmp = temporary storage, shared by everyone
/usr/bin = executables installed by your Linux (sometimes the same as **/bin**)

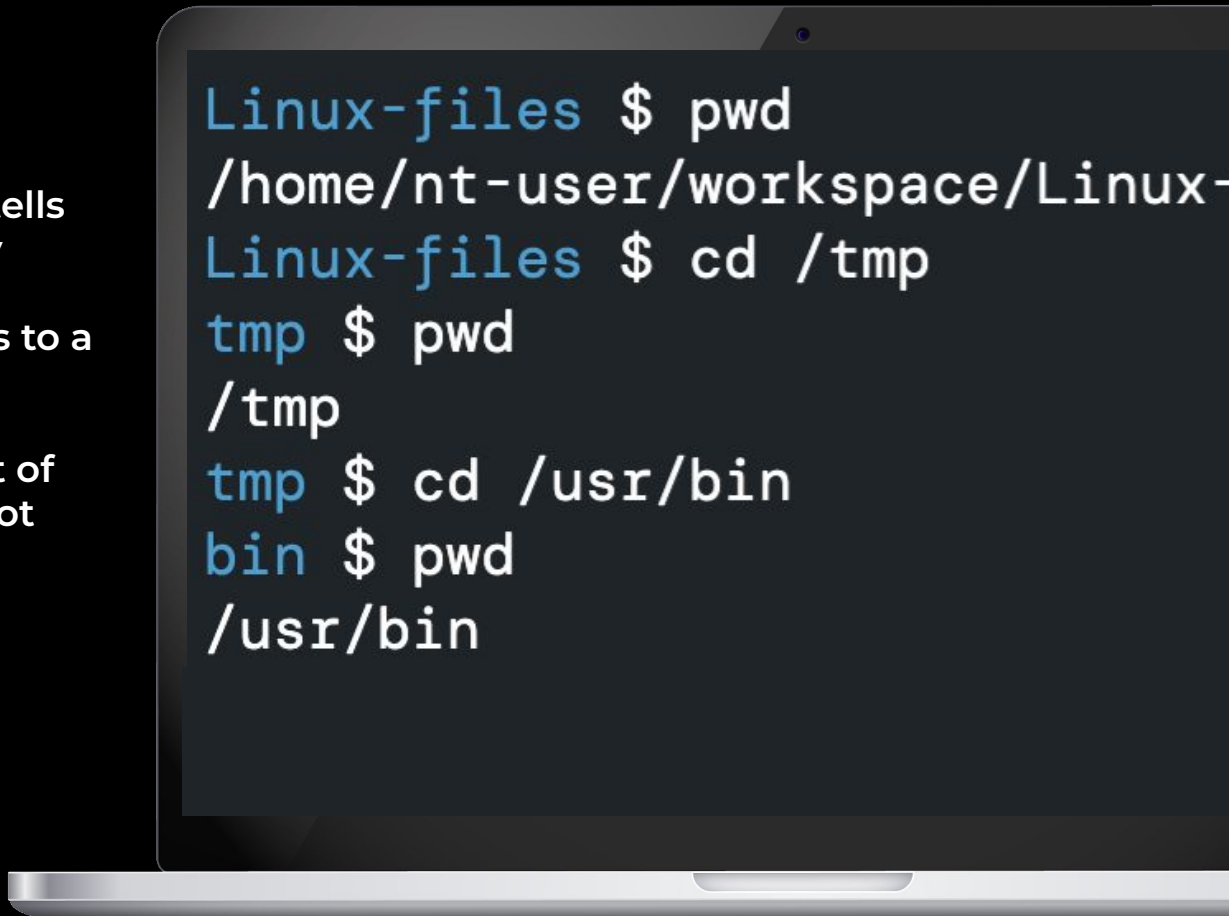
Linux Filesystem: Current working directory

pwd ("print working directory") tells us our current working directory

cd ("change directory") moves us to a (possibly different) directory

...notice your working dir (or part of it) is in your prompt, but that's not always going to be the case





Listing Directory Contents

the ***ls*** ("list") command allows us to view the contents of a directory...let's try a few variations:

```
ls
ls /tmp
ls -l /tmp
```

...how do we find out all of the options for ls?

`cd` (Change Directory)

as we've seen the ***cd*** ("change directory") command allows us to move around within the filesystem...

`cd /usr`
go to a specific dir an *absolute pathname*

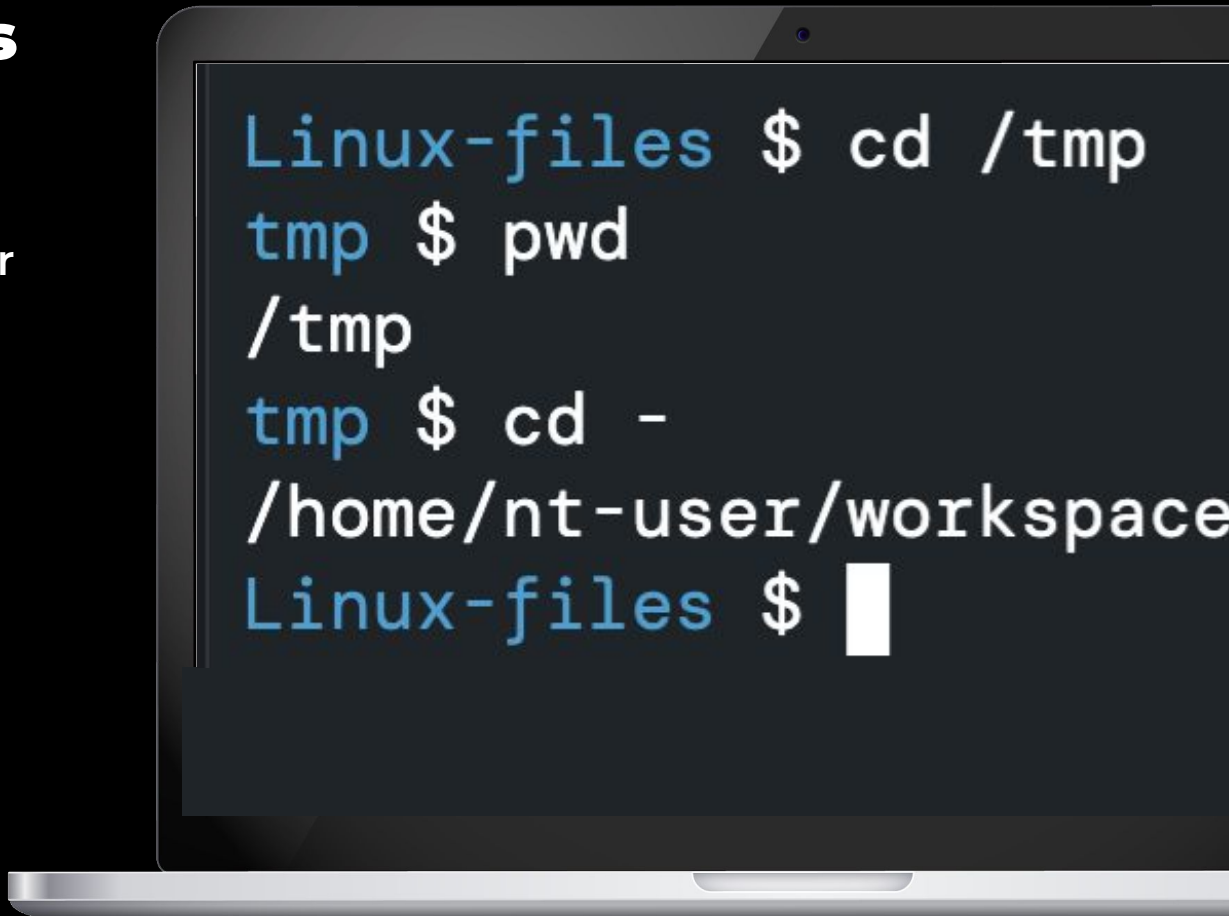
`cd bin`
go to a dir relative to the current working dir (a *relative pathname*)

`cd`
go to the user's home directory (technically same as **`cd ~`** or **`cd ~user`**)

Retrace your steps

`cd -`

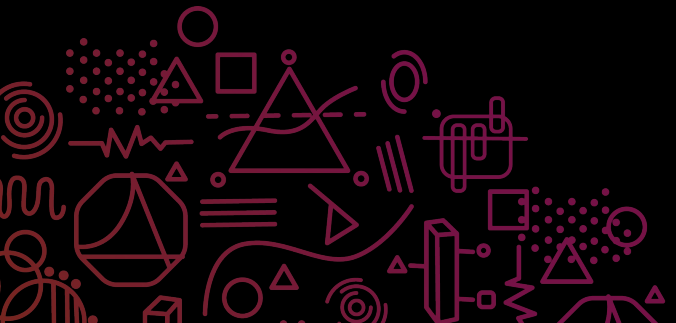
takes you back to the dir you were in before....

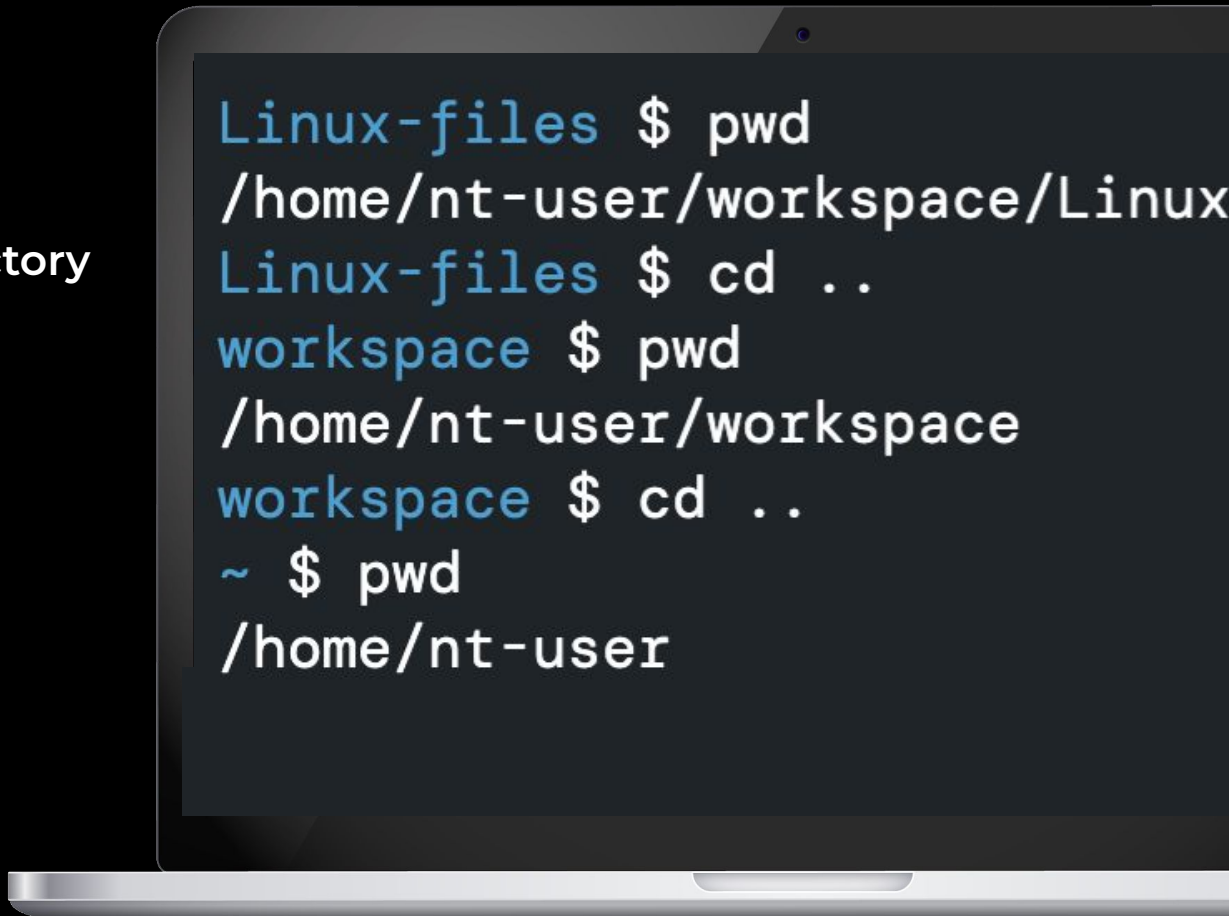


Go up!

`cd ..`

moves you UP one directory in the tree





Lab: Linux Filesystem

Enter these commands and make sure you understand what they do...

```
cd /tmp
pwd
ls
cd /
pwd
ls
cd -
pwd
ls
```

```
cd /usr/bin
pwd
cd ..
pwd
ls
cd bin
pwd
cd
pwd
```

Linux filesystem: Food for thought...

It's important to understand the difference between absolute and relative pathnames–which of the ***cd*** commands you entered specified absolute pathnames and which specified relative pathnames?

...is ***cd*** a standalone command, or a builtin?

.. is a special dir which is essentially a shortcut for the directory above

. is a special dir which is a shortcut for the current dir

What if we wanted to add a ... shortcut which means TWO directories above where we are?

Filenames

- filenames that begin with a dot (.) are so-called *hidden files*...
  - ...we can use **`ls -a`** to see them
- filenames are case sensitive, i.e., ***foo*** != ***Foo***
- extensions are not required as in Windows
- some programs expect to act on files with extensions (e.g., g++, the GNU C++ compiler), but Linux does not require them
- it's possible to embed spaces in a filename
  - shortly we'll see how to do this–and why it's not recommended

Options (redux)

multiple options can typically be combined, i.e.

`ls -la` (same as `ls -l -a`)

how do we know which options a command accepts?

what's important is not memorizing all of the possible options for each command, but rather, knowing how to find the information you need

Lab: *ls*

as we've noticed, ls has many options

take a look at man for ***ls***

try these (look up those which you don't understand, but not all will make sense yet)

```
ls -l
ls -a
ls -i
ls -S
ls -lrt /
```

Manipulating Files and Directories

cp = copies files and directories

mv = renames or moves files/directories

mkdir = creates directories

rmdir = removes directories (if empty)

rm = removes files and directories

cp: Copy Files or Directories

`cp a b`	"copy **a** to **b**"
what if **b** exists?	
`cp a somedir`	"copy **a** into dir **somedir**"
`cp a b c d somedir`	"copy files into **somedir**"
`cp dir1 dir2`	"?"

won't work unless you add **-r** (recursive) option
...if **dir2** exists, then copy **dir1** into **dir2**, otherwise **dir2** will be a copy of **dir1**

mv: Move or Rename Files/Directories

`mv a b` — "rename **a** to **b**"

what if **b** exists?

`mv a somedir` — "move **a** into dir **somedir**"

`mv a b c d somedir` — "move files into **somedir**"

`mv dir1 dir2` — "renames **dir1** to **dir2**"

...or moves **dir1** into **dir2** if **dir2** already exists

mkdir: Create Directories

`mkdir a` "create directory **a**"

what if **a** exists?

`mkdir a b c` "create three new directories"

`mkdir e/f/g` "create **g** in subdir **f** of dir **e**"

solution: **`mkdir -p e/f/g`**

rmdir: Remove Directories

`rmdir a` "remove directory **a**"

...only works if **a** is empty

`rmdir -p e/f/g` "remove **g** and all parents"

...in other words **e/f/g**, **e/f**, **e**

rm: Remove Files/Directories

`rm a`	"remove **a**"
`rm a b c`	"remove three files"
`rm a/b/c`	"removes **c** inside dir **a/b**"
`rm -rf file dir a/b/c`	"remove files and/or dirs"

-r = recursive

-f = force (used to avoid prompts and missing files)

rm: Remove Files/Directories (cont'd)

be careful, there is no undo for any ***rm*** command!

...there is a **-i** (interactive) option, but I suggest you don't use it, except in a special case we will see later

Lab: *cp/mv/mkdir/rmdir/rm*

1. Make copies of the file **abc** named **a**, **b**, and **c**
2. Change the name of **a** to **x**
3. Create a directory called **stuff**
4. Move **b** and **c** into **stuff** using a single command
5. Make a copy of **x** and put it into the **stuff** directory using a single command
6. Try to remove the **stuff** directory using ***rmdir***
7. Remove **x** and the **stuff** directory using ***rm -rf***

Manipulating Files (redux)

touch = create a new file or update modification date on an existing file

ln = create links

touch: Create/Modify Files/Directories

`touch a`

if **a** does not exist, it is created (empty)
if **a** does exist (file or dir), then its modification time is updated to the current time

Try...

```
touch newfile
touch poem
```

Links

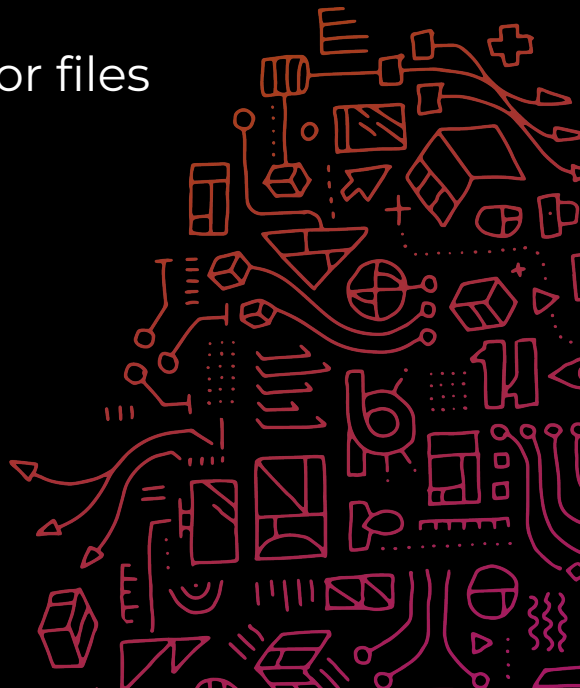
Links

A tricky but important concept in Linux...

Links are actually two different things:

1. shortcuts (a la Windows) or aliases (a la Mac) for files
   ...these are called *soft* or *symbolic* links

2. additional names for a file
   ...these are called *hard* links



ln: Create Links

`ln -s a b` "create a soft link to **a** called **b**"

a must exist

...what if **b** exists?

a is called the source, **b** is the target/destination

use `ls -l` to "see" the link

what happens if the source of the soft link is removed after the link is created?

compare that to hard links...

ln: Create Links (redux)

`ln a b` "create a *HARD* link to **a** called **b**"

a must exist

...what if **b** exists?

creates an alternate name or link for the existing file which is indistinguishable from the original name

`ls -l` will not tell you that it's a hard link except for one small clue, in the second field

try **`ls -il`** which will give you the inode number

inodes

...are data structures containing information about files

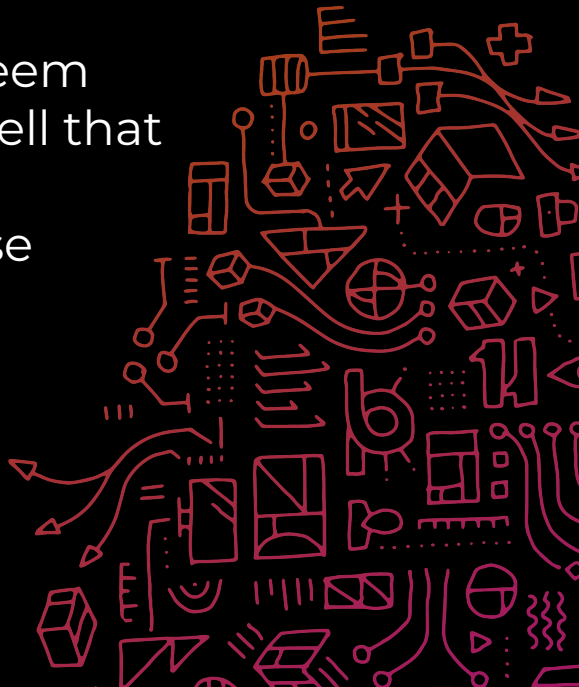
- owner
- size
- type
- permissions
- modification time
- ...but NOT the name (directories contain file names and inode numbers, nothing more)

Lab: Links

1. Create a new file
2. Create a soft/symbolic link to the new file
3. Use **ls** to verify that it's a soft link
4. Create a hard link to the new file
5. Use **ls** to verify the hard link, ideally in two ways

Lab: *touch*/inodes

1. Look at the man page for **touch** and figure out how to set the modification date of the file to an arbitrary date (try, for e.g, 'last month')
2. using the above command you can make it seem like a file is older than it really is–how can we tell that something like this was done?
3. hint, check out the **stat** command, or pay close attention to the man page for **ls**



Inodes: Food for thought...

when a hard link is deleted, the file is not affected until all links to the file are deleted

every file has at least one link

therefore, how does the rm command work?

... what does **rm** inspect in the inode?

Inspecting Files

The *cat* command

cat (short for "catenate") is a Linux command which dumps out the contents a file

many commands let you look at a file, but ***cat*** is the simplest

if you want to look at a file in a paged manner, i.e., one page at a time, use ***more*** or ***less***

The *file* command

given a file or filenames as its argument(s), ***file*** tells you what type of files they are...try:

```
file /etc/passwd
file /bin/ls
file /dev/null
```

Lab: *file*

1. Use the ***file*** command to identify your soft link
2. Try the ***file*** command on your hard link
3. Did it identify your hard link? Why not?
4. Use the ***file*** command on itself

Wildcards

Wildcards

special characters/sequences to rapidly specify a group of *filenames*

***** = any characters, e.g, **foo***, ***.txt**, **a*b**
? = any single character, e.g., **file?**, **a?c**, **foo.???**
[chars] = any character in *chars*, which is a set of characters, e.g., **[aeiou]**
[!chars] = any character NOT in *chars*,, e.g., **[!aeiou]**
{g1,g2,g3,...} = matches any or all of the terms, e.g.,
foo{1,2,3} matches **foo1**, **foo2**, and **foo3**
...but not **foo4**, **foobar**, etc.

Wildcards (cont'd)

[[:*class*:]]	any character that is a member of the class, e.g.,
[:alnum:]	any alphanumeric character
[:alpha:]	any alphabetic character
[:digit:]	any numeral
[:lower:]	any lower case character
[:upper:]	any upper case character

Lab: Wildcards

1. Which wildcard would you use to find all 3-character filenames in the current dir?
2. How about all files that have a least one dash in their name?
3. How about 3-character filenames that have no vowels in them?
4. How about filenames consisting of any characters, followed by **.txt** or any 3-character extension where the middle character is an **a**

Scripting: Building Blocks

Building Blocks

Linux provides a number of tools/programs which can be combined in powerful and interesting ways

Bash is not as powerful as Python, Perl, or other more modern scripting languages, but it is often the better tool for the job, especially when you want to run a bunch of Linux commands

We'll start with the building blocks, see what they can and cannot do, and move up to building complex shell scripts...

The Exit Status

Every Linux command conveys the status of its execution (i.e., success or failure) to the shell via an integer in the range 0-255

0 generally means *success*

non-zero can mean success or failure, depending on the conventions of the command

inspecting the exit status: **`echo $?`**

later we'll see how to branch based on exit status

Lab: Exit Status

Determine the exit status of the following commands:

```
grep zzz poem

grep undergrowth poem

cat non-existent-file

rm -f non-existent-file
```

Redirection

`date > foo`

…overwrites foo, then executes date

`set -o noclobber (set +o noclobber)`

…I don't recommend it

`cat < foo`

runs ***cat***, using file **foo** as input

Redirection (cont'd)

`cat < foo > foo`

"works", but probably not what you expected

`sort < foo > foo.sorted`

cf. `sort foo -o foo`

Standard {input, output, error}

By default, every Linux command...

reads from the *keyboard* (“standard input”)

writes to the *screen* (“standard output”)

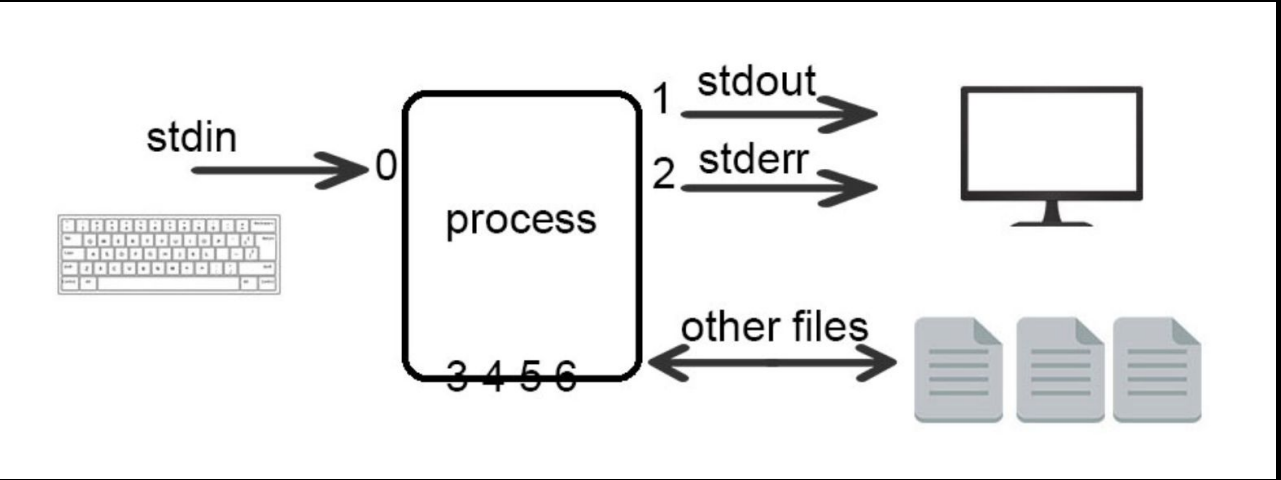
...and writes errors to the *screen* (“standard error”)

These special “channels” are really files that are opened automatically by the OS

...represented by numbered file descriptors 0, 1, 2

Standard {input, output, error}

By default, every Linux command
reads from the *keyboard* (0)
writes to the *screen* (1)
...and writes errors to the *screen* (2)



Redirection (redux)

`sort somefile 2>foo`

redirects error messages, but not normal output

`sort somefile > out 2>err`

redirects normal output to **out**, and errors to **err**

`cat poem non-existent-file >& foo`

`cat poem non-existent-file > foo 2>&1`

both of the above cause standard error to be merged with standard out, and then the merged stream goes into the file **foo**

Redirection (cont'd)

`date >> out`

redirects output to file **out**, but APPENDS the output instead of overwriting the file anew each time

if **out** does not exist, it is created, just as with **>**

Lab: Redirection

1. Demonstrate that output redirection occurs before the command is run by using **`ls`** and redirecting its output into a file in your directory
2. You can create a new empty file as follows:

   `> filename`

   ...how does the above differ from **`touch filename`**?

3. Will the following command work? Why or why not?

   `sort filename >> filename`

Food for thought...

What's the difference between these pairs of commands?

```
cat < somefile
cat somefile

sort somefile > someotherfile
sort somefile -o someotherfile
```

Connecting Commands Together

How to designate output of one command as input of another?

Our first attempt...

```
ls > foo
wc < foo
rm foo
```

Does it work?

Any problems?

Pipes

Instead, we'll use a *pipe*, which is a Bash operator for connecting the output of one command to the input of another...

```
ls | wc
ls | tee file
(date; ls) | wc
```

Let's write our first shell script:

```
echo 'ls | wc -l' > nu
bash nu
```

Processes

Every instance of an external command which is ‘running’ on a Linux system is called a *process*. (In fact, only one process per core can be running, and very few processes are even “runnable”; many are waiting for some event to occur)

Processes (cont'd)

Each process has a unique ID (or PID), an integer

Use ***ps*** to examine processes

...can also look in **/proc** "directory"

Your shell, the program you 'talk to' to execute commands has a process ID: **`echo $$`**

If you want to create a unique temporary file, which doesn't conflict with an existing file in the current dir, you can use of the process ID: **`ls > temp$$`**

More Linux Commands

More Linux Commands

Linux has so many useful commands, it's difficult to be familiar with them all

Our goal is to become familiar w/ a useful subset which can be used alone, or combined together (using pipes) to perform more complex tasks

We will look at some in detail:

sort, diff, grep, sed, tr, head, tail, awk

these Linux commands are known as *filters*

sort

many useful options

-c check whether file is sorted and exit accordingly
-f treat upper and lower case equally (‘fold’)
-n numeric sort
-r reverse sort

```
sort -c sorted-file; echo $?
sort -c unsorted-file; echo $?
sort num
sort -n num
sort -rn num
sort file -o file
```

grep

find lines in files which match a pattern

-q quiet, suppress normal output
-s suppress error messages

```
grep 10 num; echo $? (cf. -q)
grep foo numX; echo $? (cf. -s)
```

grep (cont'd)

-c count matches
-i ignore case
-l list filenames which contain matches
-n prefix match with line number
-v invert match (i.e, find lines which DO NOT match)

```
grep -c 1 num
grep -v 1 num
grep -l the *
grep -n the poem
```

Lab: *grep/pipes/sort*

1. use ***ls***, ***grep***, and a pipe to find all of the directories in the current directory

2. use ***ls***, ***grep***, and a pipe to find all of the non-directories in the current directory

3. use ***ls***, ***sort***, and a pipe to sort the long directory listing by the size of the file

tr

transl[iter]ator

...used to translate characters in the input

```
tr aeiou x < poem
tr aeiou 12345 < poem
tr 'a-z' 'A-Z' < poem
tr '[:lower:]' '[:upper:]' < poem
tr -c aeiou x < poem
tr -d ' ' < poem
tr -s ' ' < poem
```

diff

displays differences between files

-w ignore white space
-B ignore blank lines
-i ignore case
-q quiet (report diff or no diff, but not details)

```
diff num num; echo $?
diff sorted-file unsorted-file
diff sorted-file unsorted-file
                    >/dev/null; echo $?
```

Lab: *diff/tr*

Use ***diff*** and ***tr*** to determine if two files have the same content, irrespective of line breaks

(In your materials, the files **1a1** and **1a2** have the same words and even punctuation, but different link breaks.)

head/tail

show first/last lines of a file

```
head poem
tail poem
head -1 poem
tail -1 poem
tail -n +3 poem
```

for interactive use, **-f** (“follow” = output appended data as file grows) is indispensable

Putting it all together!

```
tr A-Z a-z |
tr -sc a-z '\n' |
sort |
uniq -c |
sort -n |
tail
```

Let's try it:

```
./piat < poem
./piat < hamlet.txt
```

sed

"stream editor" / filter

used to ‘make changes’ to files using pattern matching and substitution commands

-n = suppress printing of each line

```
sed ‘s/[aeiou]/x/’ < poem
sed ‘s/[aeiou]/x/g’ < poem
sed ‘s/ //g’ < poem
sed '/roads/d' < poem
sed -n '/roads/{n;p;}' poem
```

Lab: *sed*

1. use ***sed*** to change all instances of 'foo' in a file to 'bar'
2. use ***sed*** to delete all lines containing 'bar'
3. what else can we do with ***sed***?

awk

pattern scanning/processing language (awk = Aho, Weinberger, Kernighan)

supplanted by ***Perl*** and ***Python***, now sort of a one-trick pony for breaking input into fields

```
awk '{print $2}' < poem
awk -F: '{print $1}' < /etc/passwd
```

cut

remove ("cut") sections (fields, characters) of each line of a file

-d = use specified delimiter instead of TAB
-f = choose fields to cut
-s = skip lines which don't contain delimiter
--complement = invert set of fields

```
cut -f1,5 -d ' ' -s poem
cut -f2 -d: /etc/passwd --complement
```

colrm

removes columns starting at first and ending at last (if supplied)

```
colrm [first [last]]
```

handy for removing fixed length parts of lines, either at beginning or middle

```
colrm 1 6 < poem
colrm 10 < poem
```

Shell Scripting: More Pieces of the Puzzle

Variables

Just like programming languages, Bash has local and global (environment) variables

set is used to inspect local variables

env is used to inspect global or "environment" variables

```
lvar=1; set; env
gvar=foo; export gvar; set; env
```

Note: that there must be no spaces around =

Position Variables

Arguments to scripts are called "position variables"
...analogous to ***argv*** in C/C++ or ***sys.argv*** in Python

$# = number of args passed to script
$1, $2, $3... are the args themselves
$0 is the name of the script

```
./da
./da 1 2 3
./da "1 2" 3 4
```

Notice that args which have not been passed are simply empty (actually unset). It is not an error to refer to $5 if only 4 args were passed

Quoting: Backslash + Single Quotes

backslash prevents the next character from having any special meaning

```
echo ls \> foo
```

single quotes prevent shell from interpreting the text in between them

```
x=abc; echo '$x'
echo 'Is a > b?'
```

Quoting: Double Quotes

bash peeks inside double quotes to perform variable expansion (**$**), and **** escaping:

```
x="a b"; echo "$x"
echo \\; echo "\\"; echo "\$50"
echo *; echo "*"
```

Use double quotes to protect variable expansion in the event that the variable has embedded spaces (we'll see why shortly)

Simple Shell Script

```
cat $1 | wc -l
```

$1 = the first argument sent to the shell script

Try `./lc poem`

What if you wanted to send 2 arguments?

```
cat $1 $2 | wc -l
```

Try `./lc2 lc lc2`

Clumsy!

Simple Shell Script (cont'd)

Solution...

```
cat $* | wc -l
```

$* expands to the value of all arguments

Try `./lc3 lc lc2 lc3`

Shebang

if the first line of your script is **#!** (pronounced "shebang"), the rest of the line tells Linux which interpreter should be used to *run* the script

Bash is the default so you technically don't have to include a Bash shebang, but it's best practice to do so, and it will avoid problems on older systems

```
#!/bin/bash
```

you could actually write your own command interpreter, and then write scripts in your own language and have your interpreter run them
might be a good lab!

The *source* command

Sometimes you want to include the code from one file in another (like #include in C/C++)

Try **`source source1`**

source is actually the C shell syntax which has been co-opted by bash

Original way to include a file was to use '.', e.g.,

```
. source1
```

The *source* command (cont'd)

What is the difference between these two commands?

```
source source1
bash source1
```

Aliases

aliases provide a way to create a new "command" or change the behavior of an existing command

```
alias l=less
alias ll='ls -l'
alias ls='ls -F'
alias
```

to turn off an alias temporarily, preface the command with a \, e.g., **`\ls`**

to turn off an alias permanently, use ***unalias***

Lab: *alias*

1. Enter the following alias
   `alias hi='echo Hello, how are you?'`
2. Test it
3. Type exit to terminate your shell and then click *click here to reset* to get a new shell
4. Try your alias–what did you notice?

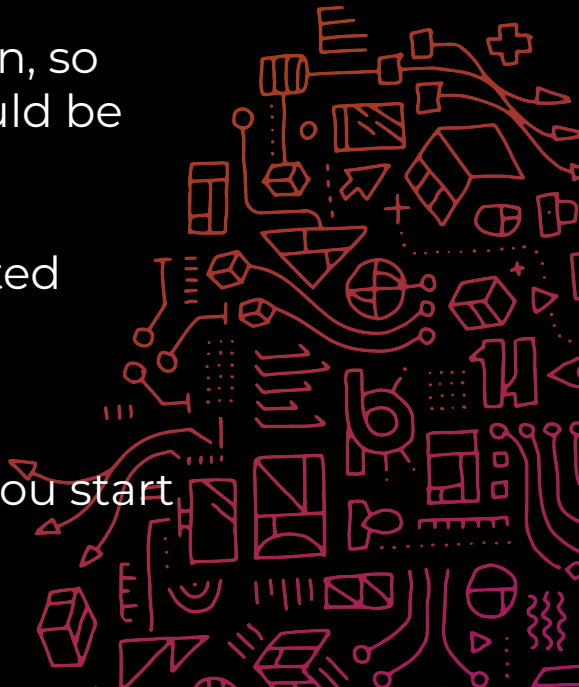
Startup Files

cd into your home directory and remember **`ls -a`** will show you the hidden files

.bash_profile - this file is sourced when you first log in, so anything that you want to happen at login time should be placed in this file (not present in the sandbox)

.bashrc - this file is sourced when a new shell is created

- aliases should go in here
- modifications to your PATH
- anything that you want to happen EVERY time you start a new shell



Backquotes

Bash runs the command inside backquotes and replaces the backquoted string with the output of the command:

```
x=`date`; echo $x
```

Very powerful construct; can be nested

```
thisdir=`basename \`pwd\``
```

Interpreted inside double quotes

```
ip="IP address is `curl https://ipinfo.io/ip`"
```

Backquotes (cont'd)

"modern day" equivalent of backquotes is **$(...)**

```
date=$(date)

thisdir=$(basename $(pwd))

ip="$(curl https://ipinfo.io/ip)"
```

The *test* command

Originally a program, now a shell builtin, used to test the status of files, and also do string/numeric comparisons

test produces no output; instead it sets its exit status to success (0) or failure (>= 1)

```
which test
type test
```

The *test* command in action

`test -f da; echo $?`

Does the file da exist? Yes: exit status is 0

`test -f dax; echo $?`

Does the file dax exist? No: exit status is 1

`x=1; test $x = 1; echo $?`

(set x equal to 1) Is x equal to 1? Yes: exit status is 0

Note: what is actually run is test 1 = 1

The *test* command in action (cont'd)

```
x=1; test $x = 2; echo $?
```
(set x equal to 1) Is x equal to 2? No: exit status is 1

```
test $x -gt 0; echo $?
```
Is x greater than 0? No: exit status is 1

```
y=1; test $x -gt 0 -a $y -gt 0; echo $?
```
(set y equal to 1) Is x > 0 and y > 0? Yes: exit status is 0

If statements

```
if command; then
    some cmd        # This branch is taken if exit
    more cmd        # status of command is 0
else
    other cmd       # This branch is taken if exit
    more cmd        # status of command is not 0
fi
```

Note that ***then*** is a separate command, and must be on a line by itself or separated from ***if*** part by a semi-colon

command is an actual Linux command or a shell builtin (e.g., ***test***) which returns an exit status

If statements (cont'd)

```
if test $x = 1; then
    echo "x = 1"
else
    echo "x != 1"
fi

if grep roads poem; then
    echo "found 'roads'"
fi
```

‘[‘ is a synonym for *test*

in order to make shell scripts look more like a standard programming language, you may use a **[** in lieu of test, but you must include a trailing **]** to close off the test:

```
if [ $x = 1 ]; then  # note spaces around [ ]
    echo "x = 1"
else
    echo "x != 1"
fi
```

Lab: *if* statements

write a shell script which takes a single command line argument representing a filename and checks to see if that file exists

it should be silent if the file exists but should complain about a file that does not exist

the script should also complain if no arguments are provided

for loops

we use a for loop when we want to do something repeatedly for a known number of times

```
for var in ...; do
    cmd
done
```

for loops (cont'd)

```
for x in foo bar baz; do
    echo $x
done

# iterate through command line arguments
for arg in $*; do
    echo $arg
done

# same as above but preserve spaces in args...
for arg in "$@"; do
    echo $arg
done
```

Lab: *for* loops

modify the shell script you just wrote so that it takes multiple command line arguments and checks for existence of all of them, rather than just one

while you're at it, append a line to each file that says "inspected by <name>"

...and remove files that are completely empty

Lab: shell scripting

Suppose a company just changed its name from **Foo** to **Bar**...

Write a script to change the text "Foo" to "Bar" for all files in a directory (Hint: ***sed***, ***for*** loop, ***man***)

Additional wrinkles if you have time...

1. back up each changed file to *filename.bak*
2. only make the changes for files ending in .txt
3. don't change words like Foot to Bart
   (hint: Google!)

Using *set* to load position variables

this a nice trick for doing ***awk***-like splitting in Bash:

```
set foo bar baz
echo $1             # foo
echo $*             # foo bar baz
set $(wc poem)
echo $*             # 17      115      679 poem
echo $2             # 115
set $(ls)           # (all files in directory)
set $(date)         # (output of date command)
echo $3             # (day of the month)
```

while Loops

we use a ***while*** loop when we want to do something repeatedly, but don't know a priori how many times

two forms...

```
while command; do
  cmd
done

until command; do
  cmd
done
```

while Loops (cont'd)

```
# check for existence of file
while test ! -f blah; do
    echo -n .
    sleep 3
done

until test -f blah; do
    echo -n .
    sleep 3
done
```

case Statements

Only builtin way to perform pattern matching in the shell (except for new extensions in bash)

Can be used instead of an ***if*** statement when you need pattern (wildcard) matching

```
case word in
    pattern1) commands;;
    pattern2) commands;;
    …
esac
```

;; is required and is used to break out of the case (like a break statement in C/C++/Python/etc.)

case Statements (cont'd)

```
read thing

case $thing in
     a*b) echo 'a*b';;
     ???) echo '3 letters';;
    *foo) echo 'ends in foo';;
 [aeiou]) echo 'single vowel';;
       *) echo 'something else';;
esac
```

"here" Documents

Suppose you want a script to print a lot of text

```
echo This is the first line…
echo This is the second line…
```

…tedious, and also hard to gauge it so the text fills the width of the page

Instead, use a here document:

```
cat <<END      (END is arbitrary)
…
END            (pattern must match one above)
```

"here" Documents (cont'd)

Not just for ***cat***–here documents are used any time you want the input to a program to come from subsequent lines of the script

Here is a simple phonebook looker upper:

```
echo -n "Enter a name: "
read name
grep $name <<END
Dave W-S 720-555-1212
Mary Smith 303-555-1212
Alice Smith 415-555-2063
END
```

“here” Strings

stripped down version of here documents

also allows you to send text to a command:

```
wc -w <<< $var

tr A-Z a-z <<< $var

lcvar=$(tr A-Z a-z <<< $var)
```

More Shell Scripting

what do you need to do on a daily basis?

Regular Expressions

similar to wildcards, regular expressions are used to describe a whole class of patterns

best demonstrated with ***grep*** and ***grep -E***, the extended version of grep

. = any single character (like the ? wildcard)
***** = 0 or more of the preceding pattern
a* = a, aa, aaa, ... (and "")
(ab)* = ab, abab, ababab, ... (and "")
+ = 1 or more
a+ = a, aa, aaa, ..., but not ""
? = match 0 or 1 times
a? = a or *nothing*

Regular Expressions (cont'd)

^ = anchor pattern at beginning

```
grep '^foo' /usr/share/dict/words
```

$ = anchor at end

```
grep 'arbor$' /usr/share/dict/words
```

[] = character class, ^ inside means invert/complement

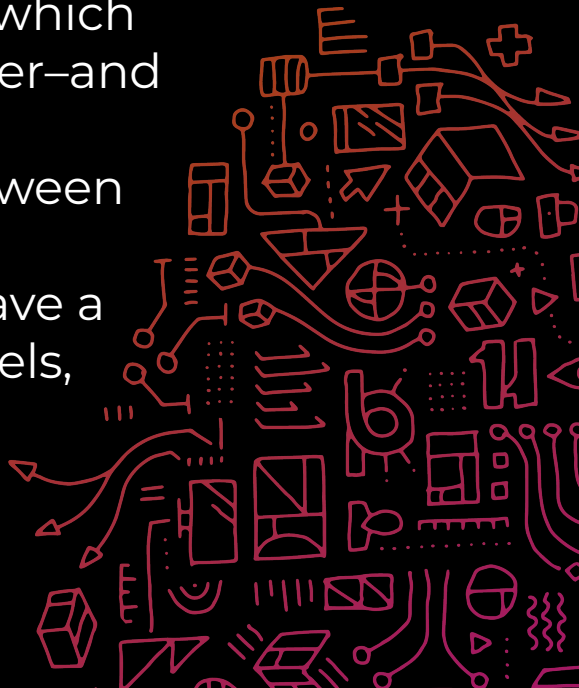
```
grep
'^[^aeiou]*a[^aeiou]*e[^aeiou]*i[^aeiou]*o[^aei
ou]*u[^aeiou]*$' /usr/share/dict/words
```

{} = repetition (with grep -E only)

```
grep -E '[aeiou]{4}' /usr/share/dict/words
```

Lab: Regular Expressions

1. find all of the words in **/usr/share/dict/words** which start with ***a*** and end with ***z***
2. find all of the words in **/usr/share/dict/words** which start with ***a*** and also contain ***b***, ***c***, and ***d*** in order–and have no ***a*** between the ***a*** and the ***b***, no ***a*** or ***b*** between the ***b*** and the ***c***, and no ***a***, ***b***, or ***c*** between the ***c*** and ***d***
3. find all words in **/usr/share/dict/words** that have a vowel (not including y) followed by 8 non-vowels, followed by another vowel



Parting Words

know where to find answers

- man/info pages
- stackoverflow
- your peers

it's not about knowing, but rather, it's about knowing how to know

Thank you!

PLURALSIGHT

Additional slides you may find useful but were removed after teaching this several times...

Digression: The Bash Prompt

You can change your prompt by typing

```
PS1="new prompt"
```

e.g.,

```
PS1="What next? "
```

we will revisit this...

Digression: How does the shell execute commands?

When you type ***ls***, the shell forks (an OS call which creates a new process which is a clone), creating a parent & child shell

The parent has control of the child and directs it to exec (a system call which overwrites the current process with the image of a new program) ***/bin/ls***

When the child process completes, the parent "reaps" the finished process (causing it to disappear from the system) and then displays your prompt so you can type again

Data in child process cannot affect parent (but child could e.g., alter files outside of its and parent's process)

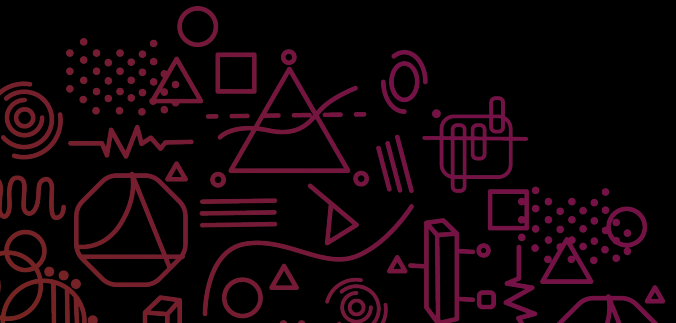
Digression: History (never repeats...)

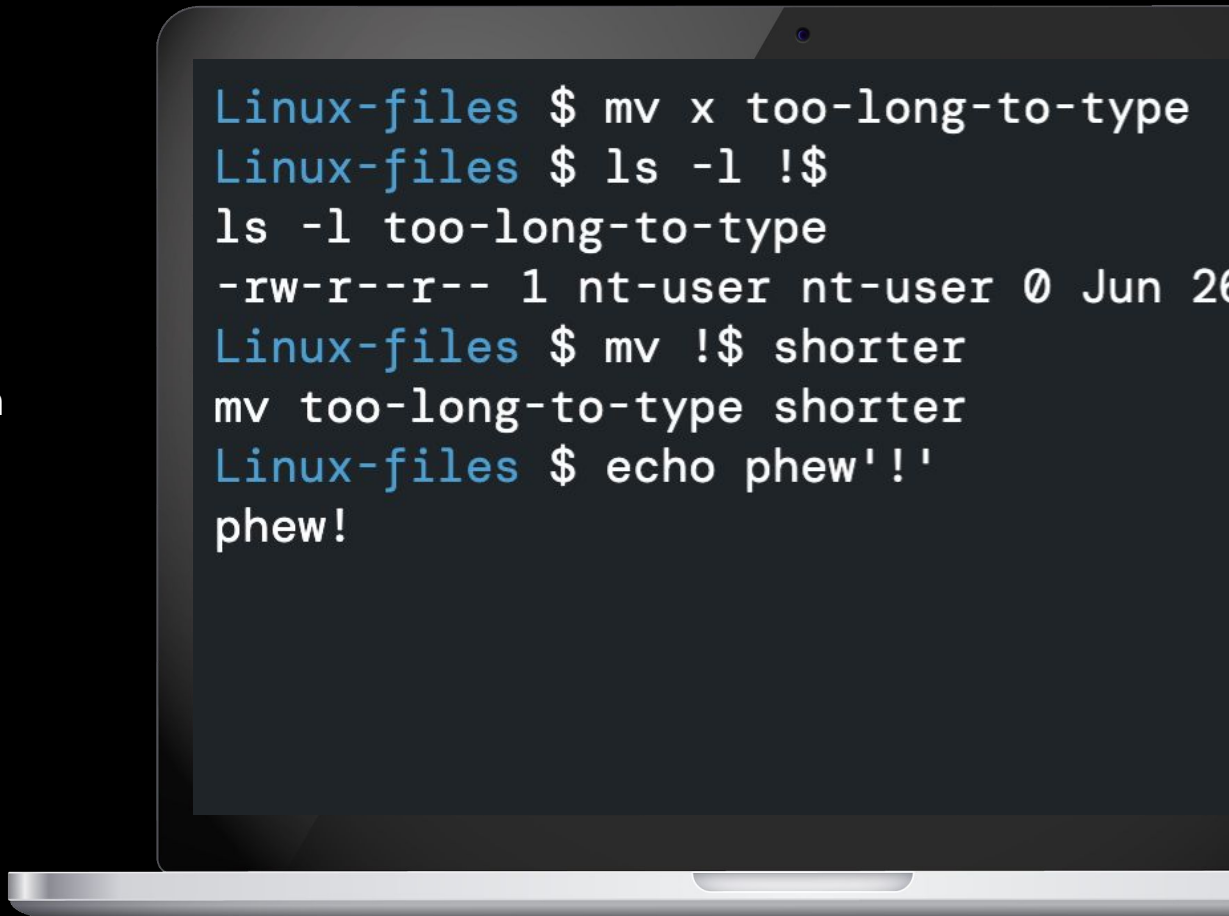
“Bang dollar”

`!$`

shorthand for the last *word* of last command

old school, but comes in handy occasionally–of course you can simply edit previous command





Bang (!) = History

actually a more general shorthand:

!! = execute last command (same as up-arrow)

!:0 = last command without any arguments

!:1 = first argument of last command

...

!* = all arguments of last command
!$ = last argument of last command

use **history** command to see previous commands as a list