Foundations of Data Science for Biologists

Wrapping Up Unix

BIO 724D

2025-FEB-18

Instructors: Greg Wray and Paul Magwene

The `find` command

Introducing the `find` command

The `find` command:

- Locates files using one or more criteria
- Searches comprehensively from a specified starting directory
- Optionally performs an action on matching files

The only required argument is the starting directory

`find .`	list files in `pwd` and all sub-directories
`find ~/analysis`	list files in `~/analysis` and all sub-directories

Default behavior:

- Returns a list of all matching files and directories with their relative path
- Searches recursively (also searches sub-directories, their subdirectories, etc.)

Key concept: recursion

Recursion: when a procedure involves invoking itself

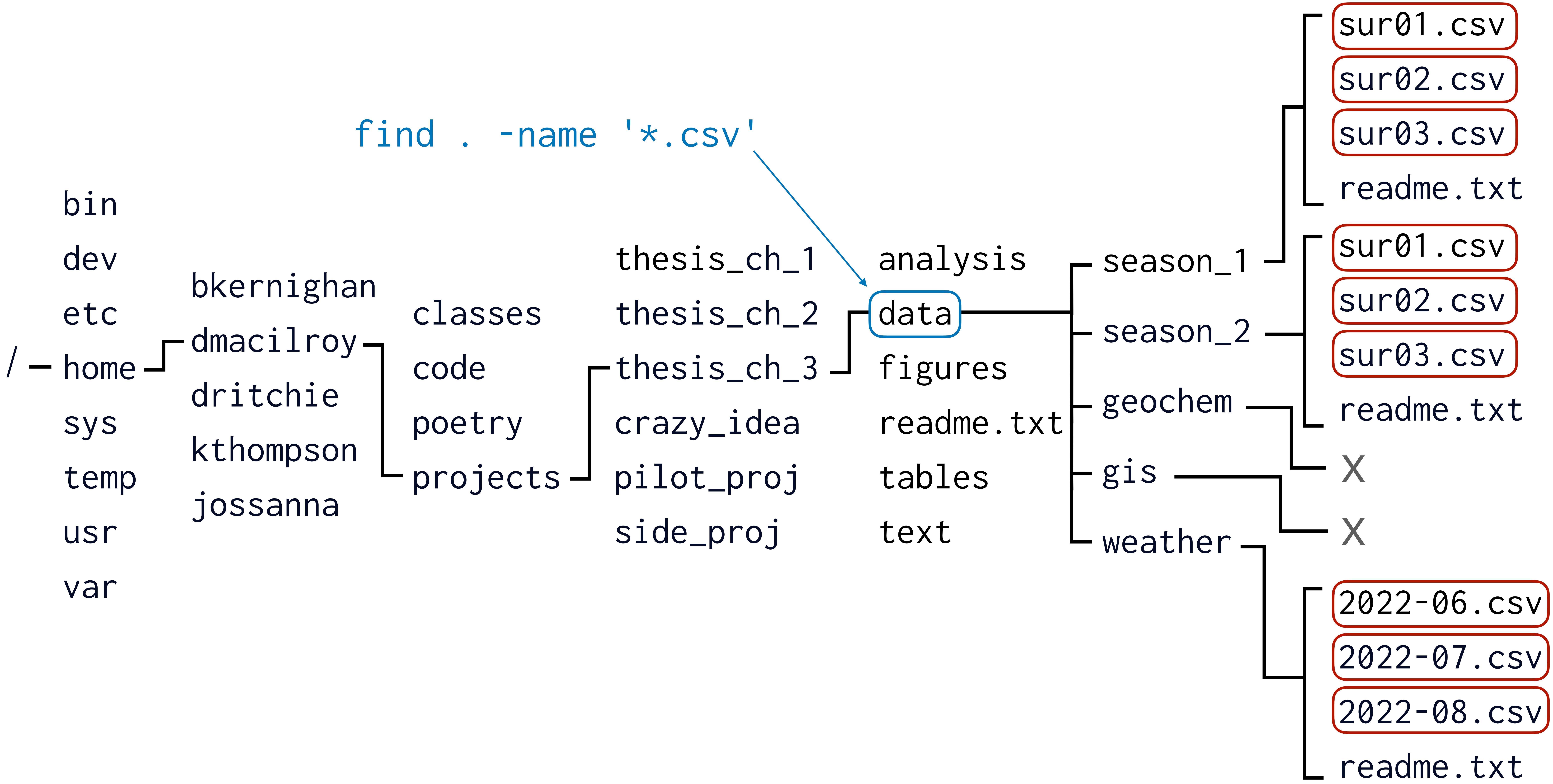
How `find` works:

- Searches within a directory: if it locates a directory it stops and calls itself
- Searches within that directory: if it locates a directory it stops and calls itself
- Eventually, it reaches a directory that contains no directories
- Then, it goes back up one level and completes searching that directory
- And so forth, until the original directory has been completely searched

This is called “walking” a directory structure: every file is checked

- Value of recursion: `find` doesn’t need loops or any information in advance

Walking a directory structure



```
find . -name '*.csv'
```

Specifying match criteria

To find files based on file **name**:

`find . -name '*.csv'`	list files with extension '.csv'
`find . -name 'test_4?.txt'`	list files that match 'test_42.txt', etc.
`find . -iname 'test_4?.txt'`	case-insensitive; finds 'Test_42.txt', etc.
`find . -regex '^data'`	searches **full paths**; does *not* find 'data.txt'

To find files based on relative **date** (note: also relative hours and many other options):

`find . -atime 3`	list files created 3 days ago
`find . -atime +365`	list files created more than 1 year ago
`find . -mtime 1`	list files created/modified yesterday
`find . -mtime -8`	list files created/modified within the past week

Specifying match criteria

To find files based on **size**:

`find . -empty`	list empty files and directories
`find . -size -50k`	list files smaller than 50 kilobytes
`find . -size +100M`	list files larger than 100 megabytes

To find files based on **type** (many more options exist):

`find . -type f`	list only files
`find . -type d`	list only directories
`find . -executable`	list directories and executable files

Combining match criteria

Criteria can be freely combined in any order (but may not be processed in that order):

`find . -name '*.csv' -mtime 1`	list `.csv` files created/modified yesterday

Using Boolean logic:

`find . -name '*.csv' -mtime 1`	AND is assumed with multiple criteria
`find . -name '*.csv' -a —mtime 1`	`-a` operator makes AND explicit (clearer)
`find . -name '*.csv' -o —mtime 1`	`-o` operator specifies OR
`find . ! -name '*.csv'`	`!` operator negates what follows

For complex conditions, use `\(` and `)\` to group criteria and/or force precedence

Actions

`find` can carry out one or more actions on files that match the search criteria:

`find . -name '*.csv' -print`	lists files with relative paths (default)
`find . -name '*.csv' -ls`	lists files with more information (= `ls -dils`)
`find . -name '*.csv' -delete`	deletes matching files (use with caution!!)
`find . -name '*.csv’ -fprint f`	prints file names to file `f`

It is possible to specify any valid command to be carried out on files that match:

`find . -type d –empty -exec rmdir {} \;`	deletes empty directories
`find . -type d –empty -ok rmdir {} \;`	asks permission for each deletion

It is also possible to pipe the list of matching files:

`find . -name '*.csv' | xargs cat`	concatenates matching files

Specifying how to search

Specify multiple start points for the search:

`find data/ code/`	specify two separate starting points

Specify the depth of search relative to the starting point (1 = pwd):

`find . -maxdepth 1`	limit search to the current directory
`find . -mindepth 2`	start searching in immediate subdirectories

Special search conditions:

`find . -L`	extends the search through symbolic links
`find . -mount`	don’t search on other filesystems
`find . -xdev`	same as above (for compatibility)

Useful examples of `find`

Display the length all .txt files and sort by length:

```
find .-name '*.txt' -exec wc -l {} \; | sort -n
```

List all .txt files in subdirectories that contain the string 'flamingo’:

```
find . -mindepth 2 -name '*.txt' | xargs grep -c 'flamingo'
```

Delete all regular files that are empty from the current directory:

```
find . -maxdepth 1 –type f -empty -delete
```

Delete all regular files that have not been accessed in >100 days, prompting for each:

```
find . -atime +100 –type f -empty -ok rm {} \;
```

Making bash scripts executable

Different contexts for using bash scripts

When you write a bash script, think about how it is likely to be used

1. Specialized for a particular project or task, you are keeping it as documentation

2. Generalized, you anticipate using it for other tasks

Using specialized bash scripts

Script is specialized for a particular project or task

Store the script with the project it supports

Use the `.sh` file extension to indicate it is a bash script (not required but recommended)

To run the script, use the `bash` command:

```
bash my_script.sh
```

Using generalized bash scripts

Script is generalized, you anticipate using it for other tasks

Be sure the shebang is on the first line and correctly formatted; optionally, remove `.sh`

`chmod +x my_script`	change permission to executable
`./my_script`	run the script (`./` is usually required)

Optionally, store the script in a folder where it is accessible from anywhere:

`mkdir ~/scripts`	create a directory for your scripts
`mv my_script ~/scripts`	move the script into that directory
`export PATH=$PATH:/home/gwray/scripts`	add the directory to your `PATH`
`my_script`	run the script from anywhere

File permissions and ownership

Every file and directory is assigned a set of **permissions**

- Permissions determine who can **r**ead, **w**rite, and e**x**ecute the file or directory
- **r** = view contents, **w** = change contents, **x** = run (programs) or view (directories)

Every file and directory has an **owner** and a **group**. By default, you are:

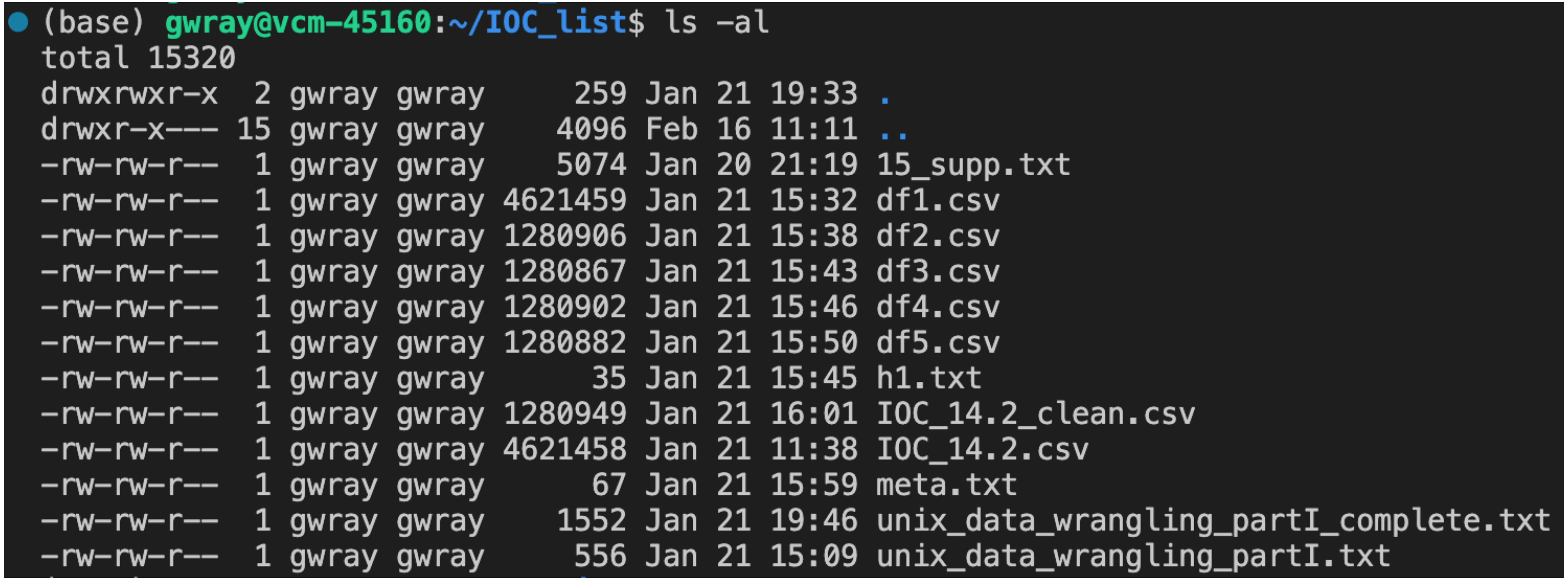
- Owner and group member of every file in your home directory and subdirectories
- The only member of your group (you can designate others)

Permissions are managed separately for owner, group, and other

- The owner can read, modify, delete, move, copy, execute, and change ownership
- Members of the group and everyone else typically have limited or no access
- The root (superuser) has complete access to every file and directory

Interpreting “long” file listings

`ls -al`



```
(base) gwray@vcm-45160:~/IOC_list$ ls -al
total 15320
drwxrwxr-x  2 gwray gwray     259 Jan 21 19:33 .
drwxr-x--- 15 gwray gwray    4096 Feb 16 11:11 ..
-rw-rw-r--  1 gwray gwray    5074 Jan 20 21:19 15_supp.txt
-rw-rw-r--  1 gwray gwray 4621459 Jan 21 15:32 df1.csv
-rw-rw-r--  1 gwray gwray 1280906 Jan 21 15:38 df2.csv
-rw-rw-r--  1 gwray gwray 1280867 Jan 21 15:43 df3.csv
-rw-rw-r--  1 gwray gwray 1280902 Jan 21 15:46 df4.csv
-rw-rw-r--  1 gwray gwray 1280882 Jan 21 15:50 df5.csv
-rw-rw-r--  1 gwray gwray      35 Jan 21 15:45 h1.txt
-rw-rw-r--  1 gwray gwray 1280949 Jan 21 16:01 IOC_14.2_clean.csv
-rw-rw-r--  1 gwray gwray 4621458 Jan 21 11:38 IOC_14.2.csv
-rw-rw-r--  1 gwray gwray      67 Jan 21 15:59 meta.txt
-rw-rw-r--  1 gwray gwray    1552 Jan 21 19:46 unix_data_wrangling_partI_complete.txt
-rw-rw-r--  1 gwray gwray     556 Jan 21 15:09 unix_data_wrangling_partI.txt
```

owner grp size kb last mod. file name

file type:
– file
d directory
l sym link

link count: number of hard links

other
group
user

r = read: can view
w = write: can modify
x = execute: can run (program) or view (directory)

The chmod command

The chmod command is used to change permissions:

```
chmod u+x my_script

chmod -w final_text.txt
```

change to executable by owner

change to read-only by owner, group, other

chmod provides fine-grained control over permissions

- We won’t cover options in this class
- However, it’s good to be aware of this command and what it is used for

The `.profile` and `.bashrc` files

Every time you log into your account on a Unix-like system, these files are run

- They are scripts that set up and customize your environment

`.profile` is run at log-in

- Contains commands not specific to bash (i.e., it is shell-agnostic)
- Place to set environment variables, including `PATH`
- Anything available to `sh` (the command interpreter)
- If logging into a bash shell, it calls `.bashrc`

`.bashrc` is run at log-in and whenever a new interactive shell is invoked

- Contains set-up specifically related to bash
- Defines how you interact with the prompt
- Contains aliases, choice of editor, customized prompt, etc.

The PATH variable

The $PATH variable is a colon-delimited list of paths:

- The shell searches this list to find commands / executable files
- Allows you to use commands / executables from anywhere without specifying the path

Order matters: paths are searched in order until a matching file name is found

- If two executable files have the same name, the first one encountered will be run
- In general, the most commonly searched directories should appear early in the list

Using the $PATH variable:

`echo $PATH`	view the current list
`export PATH=$PATH:/home/gwray/scripts`	add new path at the **end**
`export PATH=/home/gwray/scripts:$PATH`	add new path at the **beginning**

Parallel processing

Passing arguments in pipes

The `xargs` command is used to pass **arguments** rather than output in pipes

Consider the following example:

```
ls *.txt | head
ls *.txt | xargs head
```

returns the first 10 matching file names

returns the first 10 lines of each file

First case: a single list is passed as input to head, which runs once

Second case: arguments are passed one at a time, and head runs once for each argument

Note: `xargs` can also pass arguments to a set of commands that run in parallel

We won’t cover this, but you may encounter it in bash scripts

Introducing the `parallel` command

The `parallel` command lets you to run similar jobs on multiple cores at once

Not standard with most Unix / Linux distributions, but is pre-installed on your VMs

Basic syntax:

```
parallel command {} ::: inputs
```

Example:

```
parallel echo "This is job {}" ::: 1 2 3 4 5 6 7 8
```

Returns:

This is job 1
This is job 2
etc.

Using the $RANDOM environment variable

The $RANDOM variable generates a pseudo-random integer in the range 0…32767

To return an integer in a specified range, use arithmetic substitution:

```
rand=$(($RANDOM % 100 + 1))
rand=$(($RANDOM % 10 + 1))
```

returns 1…100

returns 1…10

To increase randomization, first “seed” with a unique value using command substitution:

```
RANDOM=$(date +%s)
```

uses date/time to ensure a unique seed

Passing data from a file to parallel

Typically, you will want to pass input and arguments to parallel from pipes or files

To pass input from a pipe:

```
ls *.txt | parallel wc -l {}
```

returns word count and file name

To pass input from a text file:

```
parallel wc -l {} :::: input.txt
```

returns word count and file name

To pass input from a .csv file:

```
parallel --colsep ','  echo {1}{3} :::: input.csv

parallel --colsep ',' --header 1 echo {1}{3} :::: input.csv
```

Numbers in curly braces refer to columns; number after - - header are lines to remove