Foundations of Data Science for Biologists

Working with files and text in Unix

BIO 724D

06-NOV-2023

Instructors: Paul Magwene, Jesse Granger, Greg Wray

History and philosophy of Unix

Unix history

Unix is an operating system, first developed at Bell Labs in the early 1970s. A few years after it's initial development, Bell Labs made the operating system and it's source code available to educational institutions. Because the source code was available, Unix became a popular platform for research and development in academia.

The history of Unix, its commercialization, various legal battles, etc are long and complicated. However, for our purposes it is sufficient to know that the most popular "descendant" of Unix used today is a free and open source operating system called Linux. Linux is built on the same principles as Uni, and taking advantage of many of the same tools and concepts which have been updated and improved on over the last 50 years.

Some key features that make Unix systems powerful

- Multi-user
- Supported on a wide array of hardware architectures
- Modular design
- Source code available, allowing fixes, improvements, derivations
- “Everything is a file” – documents, directories, system resources, physical devices, network interfaces, etc call all be accessed as if they were files. Shared tools and programmatic interfaces can be used across diverse resources.

Terminology: Kernel, Shell, Terminal

- Kernel – the kernel is the core or inner layer of an operating system. It controls all the tasks of the system such as managing memory and processes, communicating with attached hardware, providing useful abstraction through which other programs interact with the computer, etc
- Shell – a shell is program/environment that provides an interface between the kernel and the user. The shell interprets and translates user commands into underlying calls to the kernel.
- Terminal – A terminal is a text input/output environment for interacting with a computer. Originally, a terminal was a hardware device attached to a computer. Strictly speaking, these days we run “terminal emulators” – graphical programs that emulate terminals. We interact with the shell through the terminal.

Shells

- Bourne Again Shell (bash) – the default shell on our Linux VMs and probably the most popular terminal across different Unix operating systems. Derived from an older shell called the Bourne shell (sh).
- csh/tcsh – a shell with a C-like syntax; first developed by Bill joy at UC Berkeley as part of the Berkley System Distribution (BSD) Unix.
- Zsh – default shell on MacOS.
- … many others …

POSIX shell standard – an agreed upon set of standards for how shells should work. If you write shell scripts that follow the POSIX standard then in theory they should be easily portable across different shells that implement the standard.

The Unix Philosophy

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface. – Doug McIlroy

The Unix Philosophy, Expanded

from McIlroy, Pinson, and Tague, 1978

1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new “features”.
2. Expect the output of every program to become the input to another, as yet unknown, program. Don’t clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don’t insist on interactive input.
3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don’t hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you’ve finished using them.

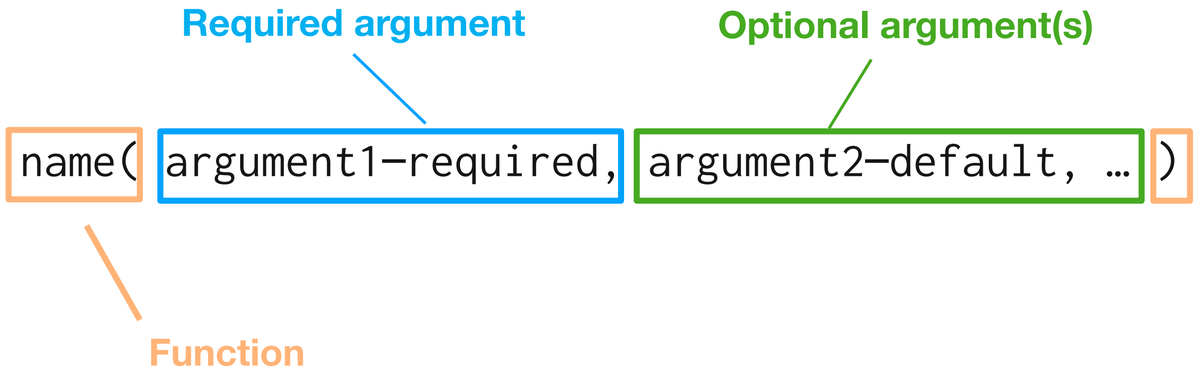
Commands and getting help

Goals for Today

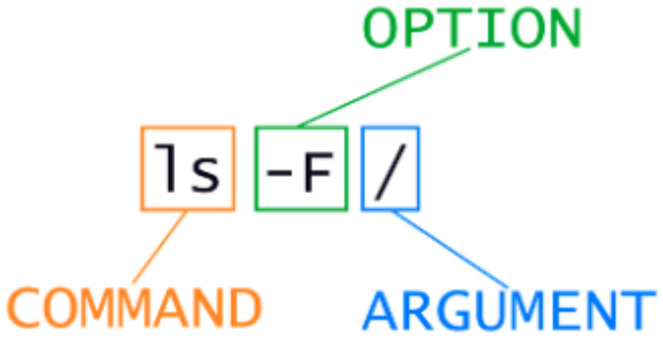
- Reviewing commands, options, and arguments
- Reviewing getting help: the man, and info pages and the --help option

The Anatomy of Commands in Unix vs Functions in R

In R, we had functions, which took arguments.
Some arguments were required, while others had a default option and so we didn't need to include anything.
Parentheses immediately follow the name of the function and encapsulate the arguments, arguments themselves are separated by commas



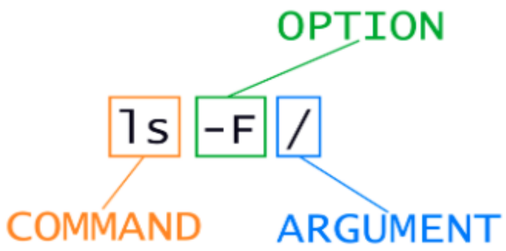
In Unix we have commands, which can take both arguments (also known as parameters) and options (also known as switches or flags).
Options change the behavior of the command, while arguments tell the command what to operate on.
Each part is separated by spaces



Arguments in Unix

Arguments help the command identify the data it needs to operate on.

For example, in the example to the right, / refers to the root directory, thereby telling ls to list the files in the root directory.



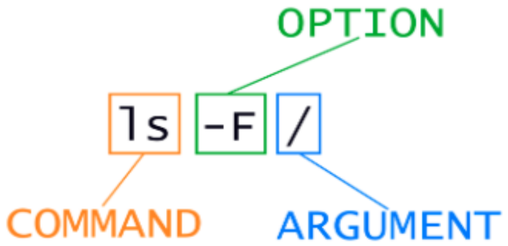
Some commands have obligatory arguments and will return an error if they are missing (i.e. cp which requires TWO arguments, a file and a place to put it).

Some commands can be called with more than one optional argument (i.e. echo or cat), while some commands require no arguments at all (i.e. date).

For commands that take multiple arguments, the order usually matters (ex: mv a.txt b.txt renames a.txt to b.txt, not the reverse)

Options/Switches in Unix

Options/Switches modify the behavior of a command. A command can be called with more than one option, or no option at all.



There are two kinds of switches (though not every command offers both):

1) The original Unix style (created back when keyboards were slow and clunky, and therefore prioritizes brevity):

- Uses single letters preceded by a single hyphen - (preferable to + because it didn't require holding down the shift button)
- Can be ganged together and the order doesn't matter, i.e. –a –b can also be written as –ab or –ba
- The argument (if any) follows the option, separated by a space.
- Lowercase options are generally preferred to uppercase (again due to the requirement of the shift key), so uppercase options (-F) are generally special variants of the lowercase option (-f). (i.e. capitalization matters)

2) The GNU style, developed later, which adds the use of keywords due to running out of single-letter option keys.

- Uses a word preceded by two hyphens
- CANNOT be ganged together, but the order still doesn't matter
- Following arguments (if any) can be separated by a white space or a single equal sign

A few other notes on commands

Exiting a command:

Ctrl+C will kill the current command that is running

Ctrl+D will exit the current shell

So, how am I ever supposed to remember all the possible options and arguments for these commands?

Ans: YOU CAN'T!

Just like in R, every command has infinite possible arguments (and options in this case).

Some options are generally all used in the same way (i.e. –a usually means all) and you can find a list of some of the common use cases here http://catb.org/esr/writings/taoup/html/ch10s05.html

However, just because that's how they're often used, doesn't mean that's how they're always used…

Getting Help (RTFM)

To figure out what arguments or options a particular command uses, we take advantage of the man and info commands, or the --help switch

- man short for manual, provides a short and succinct description of various commands and programs. Use the arrow keys to move up/down, the space bar or b to move forward/back a page, and q to quit.

Try it: type man ls to read the manual for ls or man man to read the manual for man

- info is another helpful command that works similar to man but is usually wordier and more detailed, and may include examples of usage cases.

Try it: compare info ls to man ls

- --help many commands have a --help switch that prints out a summary of the standard usage pattern along with a list of available options

Try it: try ls --help and compare it to the above use of man and info

- explainshell.com is a web site that shows you help text for common command line tools. Type a command line into the "search bar" and see explanations for various flags, etc.

Viewing and processing files

Goals for this part of class

Learn how to explore text files from the command line: `cat`, `less`, `head`, `tail`, and `wc`

Learn how to edit files using `nano`

Learn how to process text and tabular files: `grep`, `sort`, `uniq`, `cut`, and `tr`

Basic points about text-processing commands

Text-processing commands return results but do not alter the original file

- To **see** effects at the command line (default), provide 1 file name
- To **save** effects to a file, use the > operator or provide 2 files names (input, output)
- To **pipe** effects to the next command, use the | operator

Text-processing commands are designed to work in pipes

Many text-processing commands can work with specified columns or positions

View text files

`cat` and `less` are commonly used to view the contents of files:

- `cat` sends entire file contents at once
- `less` displays contents in a full-screen format that allows forward/backward scrolling

`>cat test.txt`	sends contents of file to the terminal
`>cat test.txt more.txt > new.txt`	joins contents of two files and writes to file
`>less test.txt`	displays contents in full-screen format

Notes:

- `cat` is useful for viewing small files and for concatenating multiple files before output
- `cat` is useful for sending output to files or pipes (more about this later)
- `less` is useful for viewing larger files; press `q` to quit when done
- `man` pages are displayed using `less`

View part of a text file

Use `head` and `tail` to view the beginning or end of a large file

```
>head test.txt
>head -n30 test.txt
>tail test.txt
```

sends first 10 lines to the terminal

sends first 30 lines to the terminal

sends last 10 lines to the terminal

View basic statistics about text files

Use `wc` to retrieve the number of words, lines, and characters in a text file

```
>wc test.txt
>wc -w test.txt
>wc test.txt more.txt
```

returns all three statistics

returns just the number of words

returns a table of statistics for multiple files

Notes:

- To return only number of lines, words, or characters, use `-l`, `-w`, or `-m` (respectively)
- Output is not labeled, always in this order: line, word, character, filename
- Above order is followed, regardless of order in which options are presented
- If multiple files are specified, totals will also be returned

Edit a text file

Use `nano` to edit the contents of a text file

```
>nano test.txt
```

opens a file for full-screen editing

Notes:

- If file does not exist, it will be created in working directory
- Use arrow keys to move the cursor and edit freely
- To page up and down, use `^Y` and `^V`
- To save and exit, use `^O` (output to file), then `<return>`, then `^X` (exit)
- Commands displayed at bottom of screen for reference

Filter lines in a text file

Use `grep` to keep or remove lines containing a specified search string

`>grep hello test.txt`	retains lines containing `hello` and saves to file
`>grep -v hello test.txt`	removes lines containing `hello`
`>grep -i hello text.txt`	makes search case-insensitive
`>grep -w hello text.txt`	matches only whole words
`>grep h?llo text.txt`	matches `hallo`, `hello`, `h9llo`, etc.
`>grep -n hello text.txt`	includes line numbers with filtered lines
`>grep -C2 hello text.txt`	include 2 lines before and after each match
`>grep -c hello text.txt`	returns the number of matching lines

And more… including regular expressions and filtering zipped files, directory listings, etc.

Filter with logic

It is possible to construct more complex searches with `grep`

`>grep 'hello|bye' test.txt`	OR: retains lines with `hello` or `bye`
`>grep hello test.txt | grep bye`	AND: retains lines with both `hello` and `bye`
`>grep hello test.txt | grep -v bye`	AND NOT: with `hello` but not `bye`
`>grep -v hello test.txt | grep bye`	NOT AND: not `hello` but with `bye`

Notes:

- To use OR operator `\|` enclose in quotes (single or double)
- OR filtering may differ according to shell (try adding `-e` if above doesn't work)
- No AND operator exists; use a pipe to filter in two steps
- No NOT operator exists; use -`v` option to exclude rather than include

Passing strings as arguments on the command line

Single words can be passed without quotes (quotes can be included and are innocuous)

Quotes are needed when passing strings that contain:

- Spaces
- Quotes as characters
- Non-printing characters such as <TAB> (requires escaping with `\` character)

```
>grep 'hello' test.txt
>grep 'hello world' test.txt
>grep '"hello"' test.txt
>grep "'hello'" test.txt
>grep '\t hello' test.txt
```

search string is `hello`

search string is `hello world`

search string is `"hello"`

search string is `'hello'`

search string is `	hello` (preceded by TAB)

Sort lines in a file

Use `sort` to reorder lines in a text or tabular file according specific criteria

`>sort test_01.txt`	sorts using entire lines (default)
`>sort test_01.txt -r -n`	sorts in reverse order based on integers
`>sort test_07.csv -t',' -k3,3`	sorts CSV file based on column 3
`>sort test_07.tsv -k3,5`	sorts TSV file based on columns 3 through 5
`>sort test_07.tsv -k4,4 -k1,1`	sorts TSV file based on columns 4 then 1

Notes:

- Sorts using ASCII encoding: white space, numbers, capital letters, lower case letters
- Default separator for tabular data files is `<TAB>`

Encodings

Computers store 0s and 1s; **encodings** specify how to store human-readable information

ASCII — first widespread encoding and basis for most later ones

- Name is an acronym for American Standard Code for Information Interchange
- Based on 7 bits, specifies 128 distinct printing and non-printing characters
- E.g., `1001000` indicates `<tab>` and `1100100` indicates `A`

Unicode — family of encodings used by internet and nearly every current computer

- First 128 characters identical to ASCII
- Based on 1-4 bytes, potentially specifying >1,000,000 characters
- **UTF-8** is most widespread encoding by far; specifies >100,000 characters
- Includes mathematical symbols, arrows, scripts for many languages, emoji, etc.

ASCII table

$b_7 b_6 b_5$ → / Bits				0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓	Column → / Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	—	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

sort order:

<tab>

<return>

<space>

symbols/puncutation

numerals

more symbols/punct.

upper letters

more symbols

lower letters

more symbols

US-ASCII 1967, the most widely adopted encoding during the early days of computers

Filter identical lines in a text file

Use `uniq` to remove or extract duplicated (identical) lines

`>sort test.txt sorted.txt`	sort the file before filtering with uniq
`>uniq sorted.txt`	removes copies of identical lines
`>uniq -u sorted.txt`	returns unique lines only (no identical lines)
`>uniq -d sorted.txt`	returns identical lines only (no unique lines)
`>uniq -i sorted.txt`	ignore case when sorting
`>uniq -c sorted.txt`	include the number of duplicates
`>uniq -f2 sorted.txt`	ignore the first 2 columns when sorting
`>uniq -s5 sorted.txt`	ignore the first 5 characters when sorting

Note: remember to sort before using `uniq`; no error will be returned if you don't

Replace, delete, or compress characters in a text file

Use `tr` to carry out simple text manipulations

```
>cat test.txt | tr Aa aa
>cat test.txt | tr -d hello
>cat test.txt | tr -s Aa
>cat test.txt | tr -ds hello A
```

replaces each `Aa` with `aa`

deletes each `hello`

compresses each `AaAa`, `AaAaAaAa`, etc. to `Aa`

deletes each `hello` and compresses to `Aa`

Notes:

- Input must come from a pipe
- Replacement is character-for-character; use only equal-length strings

Using character classes

Use `tr` with character classes to carry out powerful text manipulations

```
>cat test.txt | tr [:upper:] [:lower:]
>cat test.txt | tr -cd [:print:]
```

replaces uppercase with lower case

removes non-printing characters

Character classes are defined sets of characters:

`[:upper:]`	26 upper case letters of English alphabet: A . . . Z
`[:lower:]`	26 lower case letters of English alphabet: a . . . z
`[:alpha:]`	upper + lower
`[:digit:]`	10 decimal numerals: `0` . . . 9
`[:alnum:]`	alpha + digit
`[:punct:]`	punctuation characters (e.g., ! ? - ,)
`[:print:]`	printable characters; alnum + punct
`[:blank:]`	whitespace characters (e.g., <TAB> <RETURN>)

Extract specified portions of each line from text files

Use `cut` to extract specified parts of lines from text files

`>cut -f1 test.tsv`	extract column 1 from each line
`>cut -s -f1 test.tsv`	as above, but ignores lines with no delimiters
`>cut -f1-5 test.tsv`	extract columns 1-5 from each line
`>cut -f1,5 test.tsv`	extract columns 1 and 5 from each line
`>cut -d ',' -f5 test.csv`	extract column 5 from comma-delimited file
`>cut -c3 test.txt`	extract third character from each line
`>cut -w -f5 test.txt`	extract word 5 from each line

Notes:

- For tabular files, assumes `<TAB>` is the delimiter between columns
- Returns a blank line if there is no match (number of lines in input and output match)

Pipes in Unix

Unix standard streams

Programs that follow Unix conventions and that execute in a shell have associated with them a set of what are called “standard streams”, which you can think of as channels for communicating with the outside world. The three standard streams are referred to as:

- standard input (`stdin`)
- standard output (`stdout`)
- standard error (`stderr`)

By default, `stdin` is usually associated with keyboard input and `stdout` and `stderr` are the terminal display, but these can be changed.

Redirection, output operators

- `>` – redirect the output operator. Sends the output of the command on the left to the file, device, or stream on the right. If the file already exists, it will be overwritten. If the file doesn't exist it will be created.
  - `echo "Hello, World!"` – by default, stdout is associated with the terminal so executing this command command prints the result of the echo command to terminal display.
  - `echo "Hello, World!" > hello.txt` – We're now redirecting stdout to a file called `hello.txt` (open this file to confirm that the contents are what is expected)
- `>>` – append output operator. Like redirect operator but appends output to the specified file rather than creating/overwriting.

```
echo "first line" >> lines.txt
echo "second line" >> lines.txt
```

Redirection, input operator

- < – redirect input operator
  - The `tr` (translate) doesn't have a built in mechanism for reading from a file, only from stdio. However we can redirect a file to stdio use the < operator. For example, here we read text from the file `hello.txt` (created above) and translate all upper case letters to their lower case equivalents:

```
tr '[:upper:]' '[:lower:]' < hello.txt
```

We can even chain together redirect operators like so:

```
tr '[:upper:]' '[:lower:]' < hello.txt > lower_hello.txt
```

Pipes

- `|` – pipe operator. The output (`stdout`) of the command on the left is used as the input (`stdin`) for the command on the right.
- The left and right commands run concurrently
- The data is buffered – only some of it is loaded into memory at a time

Examples:

- `ls *.txt | wc -l` – list all files ending in `.txt` and count how many they are using the `wc` utility
- `cat file.txt | fold -w 1` – read the input of a file and create output with one character per line
- `seq 5 15 | shuf` (but see `shuf -i` in man pages) – generate numbers between 5 and 15 and randomly shuffle them

Extended example: Exploring genome annotation

Use `wget` to download the genome annotation for the *Saccharomyces cerevesiae* (budding yeast) genome onto your VM (see course wiki for link).

GFF feature format

GFF format – “Generic Feature Format”, a plain text format for cataloguing recognized features in genomes. This is a tab-delimited file with 9 columns, the details of which are described in the GFF3 Specification.

We’re going to be most concerned with these columns:

- Column 1: “seqid” – The ID of the landmark used to establish the coordinate system for the current feature. e.g. “chrom1”
- Column 3: “type” – The type of the feature, e.g. “gene” or “exon”.
- Columns 4 & 5: “start” and “end”– The start and end coordinates of the feature are given in positive 1-based integer coordinates.
- Column 7: “strand” – The strand of the feature. + for positive strand (relative to the landmark), - for minus strand, and . for features that are not stranded.
- Column 9: “attributes” – A list of feature attributes in the format tag=value. Multiple tag=value pairs are separated by semicolons. See the GFF3 spec for more info.

Let’s build some pipelines to answer the following Q’s

- How many annotated features are there in the yeast genome?
- What is the set of feature types?
- How many gene features are there?
- Which chromosome has the most genes?