Intro to Command Line, Season 3

March, 2024

Table of Contents

# 1 About this Course

Fluency in programming and data science requires using computer software from the Command Line, a text-based way of controlling the computer. You will go on a guided under-the-hood tour behind the graphical interface we typically use: you will learn how to interact and manipulate files, folders, and software via the Command Line.

Targeted audience: Researchers who want to use scientific software launched from the command line, want to use a high-performance cluster computing environment, or want to use a cloud computing environment.

Commitment: A 1.5 hour workshop.

# 2 Command Line Workshop

## 2.1 What is the Command Line and why should I learn it?

Let’s think about the ways we can interact with a computer: keyboard and mouse, hand gestures on a smartphone, voice commands, AR/VR, etc. Most of these interactions are related to a Graphical User Interface (GUI), which centers on the interaction between a pointer and colorful windows and menus.

However, the way a computer interprets and executes instructions are based on text commands. Even graphical information, such as where the mouse is when it clicked a button, is converted into numbers and characters. That means to be an effective programmer and data scientist, we also need to learn how to interact with our computers in a text-based way. This text-based interaction is called the Command Line Interface (CLI).

Examples where we need to use the Command Line:

* Scalable manipulation of text, files, and folders: if we want to move all files that have the words “tax returns” to a new folder, it would probably not scale easily via a mouse, but it could be done in one command in the Command Line.
* Use of programming languages and scientific software tools often require Command Line knowledge: running Python and R programs, using Git, alignment and variant calling bioinformatics software. Although there are nice graphical user interfaces such as RStudio, Juypter Notebooks, and Galaxy, to have full flexibility of these languages you need to control them from the Command Line.
* Use to high performance computing systems and the cloud all require Command Line knowledge as they do not typically offer GUIs.

The most commonly used Command Line is the Shell. The Shell is a dynamic programming environment, similar to R and Python, but focuses on interacting with the operating system.

## 2.2 Getting Started

In this workshop, we are working on a virtual command line via Replit. After this workshop, if you want to use the command line locally from your computer, see the appendix.

To access the virtual command line on Replit, you should “fork” the following [workspace](https://replit.com/@clo22/CommandLineDaSL), and look at the “Shell” window.

You should see a single line of text, with a blinking cursor, such as this:

~/CommandLineDaSL$

The next piece of character, ~/CommandLineDaSL states that the currently directory is ~/CommandLineDaSL. The symbol ~ is a short-hand for the “home” directory for the user of a computer. In replit, the home directory is /home/runner.

This current directory is important: in the Command Line, you interact with the computer from a directory, similar interacting with the computer using a file system window graphically.

To see the full directory path, type in the command pwd and hit enter.

~/CommandLineDaSL$ pwd  
/home/runner/CommandLineDaSL

Unlike a GUI, the CLI does not provide immediate options to you to interact with. We have to know a learn a handful of vocabulary to interact with it well. But besides the vocabulary, we need to keep a mental model of a task we want to complete. In GUIs that that mental model is shown to us visually, such as a file browser.

We organizes our seminar by constructing several mental models and learning relevant commands related to each model.

## 2.3 Mental Model 1: Navigating a directory tree

On our computer, the **directory tree** organizes files and directories in an (upside down) tree-like structure. In each folder, there is a parental directory, and there can be files and directories within it. The root directory / does not have a parental directory and contains all directories and files in on the computer.

Below is a typical directory tree schematic, with the root directory being / and home directory being /home/runner. We refer to a particular directory by describing its **directory path** separated by / slashes.

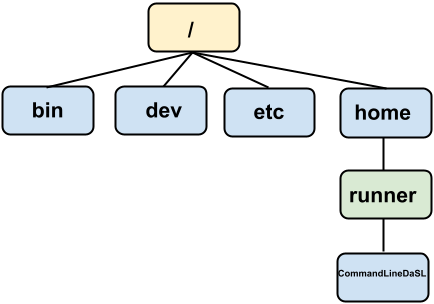


Figure : Replit directory tree

To interact with files and programs in the file system in the GUI world, we moved up and down the directory tree via point and clicking on windows. In the CLI world we can use the following commands:

* pwd prints out our current directory.
* cd / changes directory to the root directory.
* cd /home/runner changes directory to the home directory.

Consider:

~$ cd /home/runner  
~$ pwd  
/home/runner

So far, we have changed the directory by specifying the **absolute directory path**, which means specifying the directory from the root directory /. Often, we can specify a path by the **relative directory path**, which is a path *relative to our current directory*.

Consider:

~$ cd CommandLineDaSL/  
~/CommandLineDaSL$ pwd  
/home/runner/CommandLineDaSL  
~/CommandLineDaSL$ cd ..  
~$ pwd  
/home/runner  
~$

We started in /home/runner, and then changed to /home/runner/CommandLineDaSL$ by specifying cd CommandLineDaSL$/ as a relative directory path. Finally, we changed the directory to the parent directory of /home/runner/CommandLineDaSL$ via cd .., back to /home/runner. The special directory symbol .. specifies the parent directory.

How did we know that the CommandLineDaSL$/ folder is in /home/runner? We can list all the files and directories within the current directory:

~$ ls  
CommandLineDaSL$

It is very common to navigate a directory tree via iterations of ls and cd on relative paths.

### 2.3.1 Exercise: explore the project folder.

~/CommandLineDaSL$ cd project/  
~/CommandLineDaSL/project$ ls  
analysis input\_files output\_files

To examine a text file completely, use the cat command with the first argument be the text file name, such as:

~/.../project/input\_files$ cat sample1\_case.fastq   
@HWI-ST745\_0098:1:1101:1097:2038#0/1  
CNTTTTTTACTTTTTTTCCAATCAAGGTACATTCANGATCCTAATAAAATTCCANNAACACTGGACANTGATACNA

To examine the first few lines of a text file, use the head command with the first argument to be the text file name. To examine the last few lines of a text file, use tail. To scroll through a text file, use less, and press q to quit.

Hint: to save typing, use the tab button twice to complete filepaths you don’t know existed.

## 2.4 Mental Model 2: Treat text-based commands as functions

The commands you have been using, pwd, cd, ls, and cat are actually computer programs! They are completely text-based: the program take in some text input, do something with the input, prints out or save something, and quits. There are other text-based programs that are more interactive while it is running, which we will see later: but for now, we will consider this schema for our programs. (We will use programs and commands interchangeably.)

If you have done some programming yourself, you use functions to create programming expressions. A function has a name, takes in inputs, and then does something before optionally returning a output.



Figure : Source: Wellesley CS 110: <https://cs.wellesley.edu/~cs110/lectures/L16/images/function.png>

Similarly, when using a command from the command line, we should treat it as a function: a command has a *name*, inputs in terms of *options* and/or *arguments*, and optionally *returns* something. See below for an example of running the ls command with some options and arguments.

A command’s usage can have any of the following combination:

* Arguments
* Options
* Options with its own required arguments

Source: Software Carpentry: https://swcarpentry.github.io/shell-novice/fig/shell_command_syntax.svg

Figure : Source: Software Carpentry: <https://swcarpentry.github.io/shell-novice/fig/shell_command_syntax.svg>

We have been calling ls with no argument and options, and it outputs the files and folders in the current working directory.

The command can take an optional argument of a folder path (full or relative), and it outputs the files and folders in that directory:

~/CommandLineDaSL$ ls /  
bin boot dev etc home inject io lib lib32 lib64 libx32 media mnt nix opt proc repl root run sbin srv store sys tmp usr var

We add the option -F:

~/CommandLineDaSL$ ls / -F  
bin@ dev/ home/ io/ lib32@ libx32@ mnt/ opt/ repl/ run/ srv/ sys/ usr/  
boot/ etc/ inject/ lib@ lib64@ media/ nix/ proc/ root/ sbin@ store/ tmp/ var/

This displays a slash (‘/’) immediately after each pathname that is a directory, and (‘@’) after a symbolic link (not important to know right now).

It is sometimes easy to overlook that the text printed from a command like ls is indeed the returned output from the program. It is important to keep this in mind when we start to use multiple commands together later in this seminar.

### 2.4.1 Subcommands

Sometimes, a piece of software have many uses, like a swiss army knife. The software might organize its use by using subcommands. For instance, the software git has several subcommands such as git clone, git commit, and so forth. The usual options and arguments follow the subcommand.

### 2.4.2 How do we know what options and arguments to use for a command?

Often, there is a --help or -h option that tells you how to use the software:

ls --help

Online resources: <https://explainshell.com/>

### 2.4.3 Exercise: options for ls

In the maze, try out a bunch of ways to list files and directories using various options of ls. Some questions to explore:

* Can you sort by last modified?
* Can you show the long format? What is the long format?
* What are hidden files? Are there any hidden files in the maze? (This requires some googling. The manual is not clear on this.)
* How can you use multiple options at once?
* Can you print out the entire maze directory tree by using a recursive option?

## 2.5 Putting the two mental models together: file manipulation

Here are some commands that allows you to create, move, copy, and delete files and folders. All of these commands have no return value.

* cp [from] [to] copies a file or folder from the [from] path to a [to] folder.
* mv [from] [to] moves a file or folder from the [from] path to a [to] folder.
* mkdir [folderPath] creates a new folder at the path specified by [folderPath].
* rm [path] deletes a file at [path]. rm -r [folder] deletes a folder and its subcontents. Cannot be undone unless there is a backup system (usually not on personal computers, but available on FH’s computing cluster.)

There is a another special directory symbol that come up often in copying and moving. If you want to copy a file to the current directory, specify the working directory via .. For example,

cp my\_folder/file.txt .

copies the file to the working directory.

Can you explain what I am doing below?

~/.../project/input\_files$ mkdir case  
~/.../project/input\_files$ mkdir control  
~/.../project/input\_files$ cp sample1\_case.fastq case  
~/.../project/input\_files$ cp sample5\_control.fastq control  
~/.../project/input\_files$ cd control/  
~/.../input\_files/control$ ls  
sample5\_control.fastq

### 2.5.1 Wildcards to access multiple files

Suppose that I want to move all the files starting with the characters orca in the maze/west directory. I could run mv command multiple times, but that is time consuming if I have a lot of such files. In the Shell, there are special wildcard symbols that allows you to access multiple files of a specific pattern:

The \* wildcard represents zero or more characters of any form. Therefore, \*case\* will specify all files that starts with anything, have the word case in the middle, and ends with anything.

~/.../project/input\_files$ ls -l \*case\*  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:19 sample1\_case.fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:20 sample2\_case.fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:21 sample3\_case.fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:22 sample4\_case.fastq  
  
case:  
total 0

Similarly, if we want anything that ends in fastq,

~/.../project/input\_files$ ls -l \*fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:22 sample10\_control.fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:19 sample1\_case.fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:20 sample2\_case.fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:21 sample3\_case.fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:22 sample4\_case.fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:22 sample5\_control.fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:22 sample6\_control.fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:22 sample7\_control.fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:22 sample8\_control.fastq  
-rw-r--r-- 1 runner runner 1934812 Oct 23 23:22 sample9\_control.fastq

The input argument for ls is now a list of files.

We move it:

~/.../project/input\_files$ cp \*case\* case/

The wildcard can be used in different part of the filename to specify different files:

We can use \* to specify *all* files in a directory (undoing what we did before):

~/.../project/input\_files$ cd case/  
~/.../input\_files/case$ ls  
sample1\_case.fastq sample2\_case.fastq sample3\_case.fastq sample4\_case.fastq  
~/.../input\_files/case$ rm \*

## 2.6 Using a text editor in CLI

A commonly used task in CLI is to edit text files. nano, vim, and emacs are the three most popular ones, in increasing learning curve but also complexity in the tasks you can perform. We will start with nano today.

Pick a file of interest, and run nano [filename]. You will see a new screen of the filename’s contents. You can move around via the letter keys and make edits as needed. On the bottom of the screen are commands you use to manage the file, such as saving, opening, and quitting. The ^ symbol refers to the Control key on your keyboard. To quit, hit Ctrl-x, and you may have to hit y to confirm. You should get comfortable using a text editor in CLI as it is a common task.

## 2.7 Applying what we just learned: running bioinformatics software

A popular DNA sequence alignment tool is called “BWA”, and here we present a toy version of it to simulate what it like to use a bioinformatics command line tool. It takes in an unaligned sequence file and a reference genome, and then pretends to align the unaligned sequence to the reference genome.

~/CommandLineDaSL/project$ cd software/  
~/.../project/software$ python aligner.py --help  
usage: aligner.py [-h] --reference REFERENCE --input INPUT  
  
options:  
 -h, --help show this help message and exit  
 --reference REFERENCE, -r REFERENCE  
 Reference genome file  
 --input INPUT, -i INPUT  
 Input fastq file

You should use a fastq file from the input\_files folder and the reference genome file from the input\_files/reference\_genome folder.

If it works, you will get a wall of text ending with something like:

Aligned sequence 9996 : ATATTAAATTCTAATATCTGGATCCTTTTGTAGTTCATGAGCGTGATGATTGGGTGTTTCACGCATGTGTGTGCAA  
Aligned sequence 9997 : ATTTTCTCTTTTTATTTATTATTTATTGGTTTCATAGAGCTGAGTGGAGTACGAATGTCTTAACTCTGATATCATT  
Aligned sequence 9998 : TGGTCTCCATTCACACCAGCCTCTTGTCACTGCTCTGTTCTTGTCTCTGGCTTAGAGCTACTTCCTGCTATGGTCC  
Aligned sequence 9999 : ATTCTTCCACCTCTGAAAAACCCCTACCCTCACAGACACTCTAGCCCACAGTCCAACCAGTCCCCAGCCCTCTCTG  
Aligned sequence 10000 : ATCTGAGGGGACGAGAGGGTAAGATGATTGATGGAGGGGAAATCCACAGAGCCTCAGGCACCAAATATGTAGCAAA

Nice job!

## 2.8 Redirects and Pipes

The python aligner.py command seems pretty messy - it just dumps an aligned sequencing file in your command line console - what do you do with it? If you look at its manual, there is no option or argument to specify an output file name, which is a bit not user-friendly at first.

However, in the CLI world, this is a perfectly normal way to output a result from a program. We simply need to **redirect** that output to a file ourselves. The output of ./bwa mem and the output of ls can be redirected to a file, via the > operation:

python aligner.py --reference [reference genome fasta file] --input [unaligned sequences fastq file] > [output file]

Remember when we looked at the output of the ls command, an emphasis is placed that the files and directories listed are considered the output of the program. We can redirect that output into a file too:

ls > ls\_output.txt

Another reason why in the CLI world the output is often dumped to the console is that it gives the user an option to **pipe** it to the next program that takes the first program’s output as input, using the symbol |. This allows the user to chain together several programs together, each performing a modular task. For instance, if we want to align the sequences, and then look at the first few aligned sequences, we can pipe the output of bwa mem with tail:

python aligner.py --reference [reference genome fasta file] --input [unaligned sequences fastq file] | head

and then save it:

python aligner.py --reference [reference genome fasta file] --input [unaligned sequences fastq file] | head > [output file]

This encourages the development of modular, flexible programs that can be connected together via pipes and saved via redirecting.

## 2.9 Appendix: Starting the command line locally on your computer:

### 2.9.1 For Mac users

Open up “Terminal” Application.

### 2.9.2 For Windows users

* Open Windows Powershell.
* Type in: wsl --install -d ubuntu, and hit enter.
* You will might be asked to enter a new username and password. You can use the same as you have for your computer.
* A shell terminal should show up. If it doesn’t show up, look in your search bar for “Ubuntu on Windows”, and open it.

## 2.10 Appendix: Special symbols for directories:

* . the current directory.
* .. the parent directory.
* / the root directory.
* ~ the home directory.

## 2.11 Appendix: Why do we run some programs using ./?

When you type in a command, such as ls in the command line, the command line looks to see whether the command belongs to a list of commands it knows how to run. This list of approved commands is stored in an environmental variable. We can see it via:

andrew@MGQQR2YQRT9 ~ % echo $PATH  
/opt/homebrew/bin:/opt/homebrew/sbin:/usr/local/bin

In each folder (using : to distinguish folders apart) are the programs to run commands such as ls. If we want to run a program that is not in the $PATH environmental variable, there could be security issues. For instance, perhaps someone created a program called ls that does something bad to your computer, and you run it, thinking that you are running the program ls you are familiar with located in the $PATH environment variable. Or someone creating a malicious program called sl that takes advantages of typos.

To protect that, to run programs not in $PATH, we use: ./program\_name. This is a short-form of referring to the current directory ./ and running program\_name.

For more info, see this [post](https://unix.stackexchange.com/questions/4430/why-do-we-use-dot-slash-to-execute-a-file-in-linux-unix).

# About the Authors

These credits are based on our [course contributors table guidelines](https://www.ottrproject.org/more_features.html#giving-credits-to-contributors).

| Credits | Names |
| --- | --- |
| **Pedagogy** |  |
| Lead Content Instructor(s) | Chris Lo |
| Lecturer | Chris Lo |
| Content Author(s) (include chapter name/link in parentheses if only for specific chapters) - make new line if more than one chapter involved | If any other authors besides lead instructor |
| Content Contributor(s) (include section name/link in parentheses) - make new line if more than one section involved | Wrote less than a chapter |
| Content Editor(s)/Reviewer(s) | Checked your content |
| Content Director(s) | Helped guide the content direction |
| Content Consultants (include chapter name/link in parentheses or word “General”) - make new line if more than one chapter involved | Gave high level advice on content |
| Acknowledgments | Gave small assistance to content but not to the level of consulting |
| **Production** |  |
| Content Publisher(s) | Helped with publishing platform |
| Content Publishing Reviewer(s) | Reviewed overall content and aesthetics on publishing platform |
| **Technical** |  |
| Course Publishing Engineer(s) | Helped with the code for the technical aspects related to the specific course generation |
| Template Publishing Engineers | [Candace Savonen](https://www.cansavvy.com/), [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/) |
| Publishing Maintenance Engineer | [Candace Savonen](https://www.cansavvy.com/) |
| Technical Publishing Stylists | [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/), [Candace Savonen](https://www.cansavvy.com/) |
| Package Developers ([ottrpal](https://github.com/jhudsl/ottrpal)) [Candace Savonen](https://www.cansavvy.com/), [John Muschelli](https://johnmuschelli.com/), [Carrie Wright](https://carriewright11.github.io/) |  |
| **Art and Design** |  |
| Illustrator(s) | Created graphics for the course |
| Figure Artist(s) | Created figures/plots for course |
| Videographer(s) | Filmed videos |
| Videography Editor(s) | Edited film |
| Audiographer(s) | Recorded audio |
| Audiography Editor(s) | Edited audio recordings |
| **Funding** |  |
| Funder(s) | Institution/individual who funded course including grant number |
| Funding Staff | Staff members who help with funding |

## ─ Session info ───────────────────────────────────────────────────────────────  
## setting value   
## version R version 4.0.2 (2020-06-22)  
## os Ubuntu 20.04.5 LTS   
## system x86\_64, linux-gnu   
## ui X11   
## language (EN)   
## collate en\_US.UTF-8   
## ctype en\_US.UTF-8   
## tz Etc/UTC   
## date 2024-03-14   
##   
## ─ Packages ───────────────────────────────────────────────────────────────────  
## package \* version date lib source   
## assertthat 0.2.1 2019-03-21 [1] RSPM (R 4.0.5)   
## bookdown 0.24 2024-03-13 [1] Github (rstudio/bookdown@88bc4ea)   
## cachem 1.0.8 2023-05-01 [1] CRAN (R 4.0.2)   
## callr 3.5.0 2020-10-08 [1] RSPM (R 4.0.2)   
## cli 3.6.2 2023-12-11 [1] CRAN (R 4.0.2)   
## crayon 1.3.4 2017-09-16 [1] RSPM (R 4.0.0)   
## desc 1.2.0 2018-05-01 [1] RSPM (R 4.0.3)   
## devtools 2.3.2 2020-09-18 [1] RSPM (R 4.0.3)   
## digest 0.6.25 2020-02-23 [1] RSPM (R 4.0.0)   
## ellipsis 0.3.1 2020-05-15 [1] RSPM (R 4.0.3)   
## evaluate 0.23 2023-11-01 [1] CRAN (R 4.0.2)   
## fastmap 1.1.1 2023-02-24 [1] CRAN (R 4.0.2)   
## fs 1.5.0 2020-07-31 [1] RSPM (R 4.0.3)   
## glue 1.4.2 2020-08-27 [1] RSPM (R 4.0.5)   
## htmltools 0.5.7 2023-11-03 [1] CRAN (R 4.0.2)   
## knitr 1.33 2024-03-13 [1] Github (yihui/knitr@a1052d1)   
## magrittr 2.0.3 2022-03-30 [1] CRAN (R 4.0.2)   
## memoise 2.0.1 2021-11-26 [1] CRAN (R 4.0.2)   
## pkgbuild 1.1.0 2020-07-13 [1] RSPM (R 4.0.2)   
## pkgload 1.1.0 2020-05-29 [1] RSPM (R 4.0.3)   
## prettyunits 1.1.1 2020-01-24 [1] RSPM (R 4.0.3)   
## processx 3.4.4 2020-09-03 [1] RSPM (R 4.0.2)   
## ps 1.4.0 2020-10-07 [1] RSPM (R 4.0.2)   
## R6 2.4.1 2019-11-12 [1] RSPM (R 4.0.0)   
## remotes 2.2.0 2020-07-21 [1] RSPM (R 4.0.3)   
## rlang 1.1.3 2024-01-10 [1] CRAN (R 4.0.2)   
## rmarkdown 2.10 2024-03-13 [1] Github (rstudio/rmarkdown@02d3c25)  
## rprojroot 2.0.4 2023-11-05 [1] CRAN (R 4.0.2)   
## sessioninfo 1.1.1 2018-11-05 [1] RSPM (R 4.0.3)   
## stringi 1.5.3 2020-09-09 [1] RSPM (R 4.0.3)   
## stringr 1.4.0 2019-02-10 [1] RSPM (R 4.0.3)   
## testthat 3.0.1 2024-03-13 [1] Github (R-lib/testthat@e99155a)   
## usethis 1.6.3 2020-09-17 [1] RSPM (R 4.0.2)   
## withr 2.3.0 2020-09-22 [1] RSPM (R 4.0.2)   
## xfun 0.26 2024-03-13 [1] Github (yihui/xfun@74c2a66)   
## yaml 2.2.1 2020-02-01 [1] RSPM (R 4.0.3)   
##   
## [1] /usr/local/lib/R/site-library  
## [2] /usr/local/lib/R/library

# 3 References