Course Notes: Command Line

These notes are new for the Fall 2020 semester, and they are very much a work in progress, and they'll likely be updated/augmented/refined throughout the semester. If you notice mistakes (big or small), if you have questions, if anything is unclear, if there are things not covered here that you'd like to see covered, or if you have any other suggestions; please get in touch either by posting to the course notes section of the forum, or by email to `6.009-staff@mit.edu`.

Table of Contents

1) Introduction

A *command line* is a text-based interface to your computer. You can think of it as being similar to the MacOS Finder or Windows Explorer in that it provides an interface for interacting with programs and other files on your computer; the main difference, though, is that the command line is entirely text-based. Rather than pointing and clicking on menus and buttons, one uses the command line by typing commands to perform specific tasks.

In part because it is text-based, working with the command line can feel difficult and intimidating at first, but it is a really awesome tool, and putting in the time and effort to develop a familiarity with the command line is well worth the time and effort. Not only can it look really cool, but using the command line can be a really efficient way of interacting with your computer. After getting used to it, some people even spend almost all of their time at a command line rather than using graphical programs!

But it's worth noting that when you see someone typing like the wind, flying around the terminal with ease, keep in mind that their fluency with those tools is often the result of years or practice (and years of accumulating neat "tricks" and shortcuts for various operations); just like learning to program, though, no one is born understanding how the terminal works, and it takes time and practice to develop that facility.

Part of our hope is that 6.009 can serve to introduce you to some of the basics of operating a computer via the command line, as a first step on that journey; and, while we can't cover everything here, we hope that this page can serve as a brief primer to help you get started by introducing a few of the most commonly used commands in general, as well as some of the commands you are likely to use frequently throughout 6.009.

If you get stuck, please also keep in mind that we're here to help (via office hours and/or via the forum).

2) Starting the Terminal

Before we can start typing our commands, we need to open a program that will accept those commands: the terminal. The exact details of how you open a terminal, as well as how you interact with it, will vary a little bit depending on the details of your setup[1]:

- On MacOS, you can open the "Terminal" program (by searching for "Terminal" in Finder, or from Finder->Applications->Utilities->Terminal).
- On GNU/Linux, you can open the "Terminal" program (sometimes a keyboard shortcut like `Ctrl+Alt+t` will open it, or you can search for it.)
- On Windows, we recommend installing Git for Windows, which includes a terminal called "Git bash." After you have installed Git for Windows, look for the Git Bash program and run it.

3) Commands

As the name implies, the command line lets you interact with your computer by running "commands." As a basic part of its operation, it also keeps track of the "working directory," the directory (folder) you're currently in; which will be important to many of the commands we will use.

When you first open your terminal, you are presented with a *prompt* (usually ending with a dollar sign `$` or an angle bracket `>`), followed by a blinking cursor. This is your indication that the shell is ready to receive a command from you.

We can think of most commands as consisting of three things:

- the name of the program we want to run,
- zero or more "options" that change the behavior of the program, and
- zero or more "arguments" on which the program should operate.

Options and arguments are generally separated by spaces.

Now let's try running our first command. We'll start small, with a command consisting just of a program (with no options or arguments). At the prompt, type the letters `pwd` and hit the enter key.

Your computer should respond by printing something like:

- `/Users/YOUR_USERNAME` on MacOS
- `c/Users/YOUR_USERNAME` on Windows
- `/home/YOUR_USERNAME` on GNU/Linux

This is the name of your current working directory (`pwd` stands for "**p**rint **w**orking **d**irectory"). This is a folder on your system (that you could navigate to using a graphical interface), and many commands we run will make use of the fact that we are currently in that folder.

Now let's run a second command (which again consists only of a program, with no options or arguments). Try typing `ls` (that a lower-case L followed by a lower-case S) and hitting Enter. As a result, you should see list of the files and folders contained within the current directory (`ls` is short for "list").

3.1) Notation

Often, when you see commands written out in an article, they will include a dollar sign to indicate the prompt. For example, a moment ago, we ran the `pwd` and `ls` commands. In many places, those would have been presented as:

```
$ pwd
```

and

```
$ ls
```

respectively. Note that when you see something like this, the dollar sign represents the prompt (it's your indication that what follows is a command to be typed into a terminal), so you should not type it into your terminal (only type the parts that follow the dollar sign).

6.009's assignments will also generally follow this convention when indicating commands that you should run at your command line.

4) Arguments

Now let's see an example of what happens when we modify the behavior of a command by specifying an option. Options generally start with a hyphen `-`, and they modify the behavior of a program.

To see an example, let's try running the following command (remember to type only the piece after the dollar sign! and note that the `-l` is a hyphen followed by a lower-case L):

```
$ ls -l
```

The `-l` is an option that tells `ls` to produce more detailed output. Here, instead of seeing only the names of the files and folders, we see some information about them. We'll not worry about interpreting that output yet (feel free to ask on the forum if you are interested), but includes information about how large each file is, when it was modified, etc. For our purposes now, though, we just want to note that that option changed what the `ls` program reported back to us.

5) Navigating with `cd`

Another command that you will likely use a lot is `cd` (which stands for "**c**hange **d**irectory"). This command is used to move around your computer by changing the working directory.

For example, it's likely that the output of the `ls` program indicated a subdirectory called `Downloads`. If so, and if we run the `cd` program with `Downloads` as an argument, we should see that our working directory changes:

```
$ cd Downloads
```

Depending on your setup, it is likely that the prompt changed to indicate the current working directory (which is a really nice feature!). If it didn't, you can run `pwd` to see what directory you are in (and maybe consider posting a message to the forum so we can help you configure your shell to display the working directory as part of the prompt).

Note that if we now run `ls`, we should see a list of all of the files and folders contained within `Downloads` (with no arguments specified, `ls` will always show us the files that are in the current working directory).

You can run the following command to move back up the directory hierarchy:

```
$ cd ..
```

(the `..` is a special name that refers to the *parent* of a directory)

You can also move more than one directory at a time. For example, if we had a folder called `Music` inside our `Downloads` folder, we could get to it by running `cd Downloads` followed by `cd Music`, but we could also get there with a single command `cd Downloads/Music` (note the `/` which is used as a separator). Similarly, if we were in that directory, running `cd ../..` would bring us back (by moving us up two levels in the filesystem).

If you ever get lost in your filesystem, you can also just run `cd` with no arguments to return to your "home" directory:

```
$ cd
```

6) Quick Overview of Commands Related to Navigation

- `cd` ("**c**hange **d**irectory")

  Changes the working directory. If an argument is given, that argument (if it refers to a valid directory) becomes the working directory. If no argument is given, the user's "home" directory becomes the working directory.

  `cd ..` moves up one level in the directory hierarchy.

- `pwd` ("**p**rint **w**orking **d**irectory")

  Prints out the current directory, if you're not sure where you are.

  (On a well-configured system, this command is probably not used very often, as your shell's prompt will typically include information about the working directory)

- `ls` ("list")

  Lists files and folders. If no argument is given, lists the files in the current directory. If an argument is given, lists the files in that location instead.

  The `-l` option causes `ls` to print more information (a "long" listing) for each file. The `-a` option ("all") causes `ls` to show hidden files (files or directories whose names begin with a period `.`, which are normally not shown).

- `mv` ("move")

  Moves or renames a file. Typical usage involves two arguments, a source (the file to be moved) and a destination (either a new name for the file, or a irectory into which the file should be moved).

- `rm` ("remove")

  Deletes a file given as an argument.

- `mkdir` ("make directory")

Creates a new directory. For example, to create a directory called `hello` in the current working directory, run `mkdir hello`.

7) Running Python

Of course, we are not limited to using the terminal only for navigation! We can use it to interact with other programs as well[2]; and in 6.009, one of the most common things we will do in the terminal is to ask Python to run a program we've written. `python` can be specified as the first element of a command just like any other program.

In a typical lab directory, we have a file called `lab.py` that contains your work for the lab. If you have used `cd` to navigate to the directory containing your file, you can run the following:

```
$ python lab.py
```

Note

On some setups, you may need to type `python3` instead of `python`, to differentiate it from older versions of Python.

This command will cause Python to run the program in `lab.py`, including printing the results of any `print` statements to the terminal.

Specifying the `-i` flag will cause Python not to exit after evaluating the code in the given file, but rather to enter into a "REPL" (**R**ead-**E**valuate-**P**rint-**L**oop), usually prefaced with `>>>`, where you can type Python commands to interact with objects your program created.

For example, the following would run the code in your `lab.py` and then put you into a Python REPL where all definitions from your `lab.py` are available:

```
$ python -i lab.py
```

8) Installing Python Packages with Pip

While most 6.009 labs will not rely on any software that is not included as part of Python's "standard library" (i.e., software that is included with a typical Python installation), we will occasionally make use of additional packages. The `pip` program (included in most Python installations) is used to install additional packages/modules to your Python environment.

Perhaps most importantly, all of our labs require having the `pytest` package installed, for running the test cases we distribute with each lab. You can install this with a command like the following:

```
$ pip install pytest
```

Note

Depending on your environment, you may need to type `pip3` instead of `pip`, and/or you may need to preface the command with `sudo`. If you have trouble with commands like this, please let us know; we're happy to help!

After running this command, running a Python file containing an `import pytest` statement should work without error.

9) Running Pytest

We will be using `pytest` as a means of testing the behaviors you implement in your `lab.py` files for correctness. While this can be done from within Python, it is also useful to learn some things about using `pytest` as a command to run the test cases for a given file.

The most direct way to do this is to use `cd` to navigate to the directory containing your `lab.py` (and the associated `test.py`) and to run the following:

```
pytest test.py
```

This will display some information about which test cases passed and which failed (a dot indicates a test case that passed, and an F indicates a test case that failed).

However, `pytest` is very flexible, and you can customize some details of its behavior by supplying additional options/arguments. Some useful examples:

- `pytest -v test.py` will display more information about the test cases as they are being run.
- `pytest -x test.py` will cause execution to stop after the first failed test case (if any), rather than running all test cases
- `pytest -s test.py` will cause print statements to print to the terminal immediately as test cases are run (without `-s`, the output from print statements is collected and only displayed after running all test cases)
- `pytest -k PATTERN test.py` will run only the test cases that contain `PATTERN` in their name. For example, in lab 0, running `pytest -k echo test.py` will cause only the test cases with "echo" in their name to run (in this case, running only the tests corresponding to that portion of the lab).

These options can also be combined, so, for example, `pytest -s -x -k echo test.py` will only run the echo-related tests, exiting after the first error and displaying the output from `print` statements during execution rather than at the end.

`pytest` is a fairly complicated and flexible program, so there are additional options you can provide; but the examples above are likely to be some of the most useful as you work on 6.009.

10) Other Tips

This last section does not introduce any new commands, but it contains some advice for efficiently navigating using the terminal, as well as some other hints

10.1) Pro-tip: "Tab Completion"

Most command lines support a useful feature called "tab completion" that helps us avoid some of the monotony of typing out long arguments to programs (often, long filenames).

In the example above, if we had typed only `cd Dow` and hit the tab key, your shell will likely fill in the rest of the word `Downloads` for you! Making use of tab completion is a good habit to get into, as it really makes navigating using the terminal a lot more pleasant!

10.2) Pro-tip: Up-arrow and Down-arrow for Navigating History

Often, we want to run the same (or similar) commands repeatedly, one after the other. It can be tedious to type the same command multiple times, so

When faced with an empty prompt, you can use the "up arrow" key to put the command you just ran back on the command line. After having done so, you can edit the command if you want (to fix typos, change arguments, etc), and you can hit Enter to run it (regardless of whether you edited it).

Hitting the up arrow multiple times will bring up older and older commands. You can use the up- and down-arrow keys to navigate through the history of commands you have run, so you won't have to re-type a long command if you don't want to, even if you ran it a long time ago.

10.3) Filenames with Spaces

As we mentioned above, options and arguments to a program are separated by spaces. As such, it is usually conventional to avoid putting spaces in filenames, since this can confuse the terminal (for example, `My Documents`, when typed directly into the terminal, would be interpreted as two separate arguments (`My` and `Documents`), rather than as a single entity.

But luckily, there are options for working with arguments that contain spaces. One way to handle this is to wrap the argument in quotes `"`, for example: `cd "My Documents"`.

Another way is to add a backslash `\` in front of each space, for example: `cd My\ Documents`. The backslash tells the comand line that the following space should be treated as a literal "space" character, rather than as a separator between two separate arguments.

Footnotes

[1] There are also some differences in how these programs behave (especially Windows' Powershell versus the terminal in MacOS or GNU/Linux), but all of the commands we're describing in this document should work the same in all of these environments.

[2] Some people rarely leave the terminal and use various text-based programs for text editing, e-mail, web browsing, etc!