15.S60

Lecture 1: The Terminal

Bradley Sturt • January, 2016 • Massachusetts Institute of Technology

Last Revision: January 3, 2016

Table of Contents

1 Introduction

1.1 What is the terminal?

The **terminal** is a portal that allows us to interact with computers, exclusively by sending commands through the keyboard. For example, let's say we downloaded the entire "Romeo and Juliet" play by William Shakespeare, and wanted to find out the number of words in the play. In the terminal, we type

```
$ wc -w RomeoAndJuliet.txt
```

This command is in the format that is understood by the default **shell** (the "shell" is the program that receives and interprets the commands from the terminal). The command says to run the "Word Count" program, referred to as `wc`, on the file `RomeoAndJuliet.txt`. Since the extra argument `-w` was present, the terminal will only display the number of words.

When we click enter, we get the following:

```
$ wc -w RomeoAndJuliet.txt
25640 RomeoAndJuliet.txt
```

When we pressed enter, the shell received and interpreted the command. The computer then ran the built-in word-counter program. When it finished running, the program told the shell to output to the terminal the text `25640 RomeoAndJuliet.txt`. Thus, we now know there are 25640 words in Romeo and Juliet.

Terminals come default on just about every operating system. Just like there are many programming languages, there are many types of shells, each with their own set of built-in programs (like word-counter) and ways of specifying the commands (called the shell's syntax). For many decades, "BaSH" has been the built-in and default shell for most modern day operating systems (such as Macs, Linux, etc), and likely will continue to be the standard for many more decades to come. For these reasons, we will focus on the BaSH syntax. Microsoft Windows, unfortunately for us, uses a different shell, with a different syntax.

The focus of this lecture is on learning how to interact with computers through the terminal. In the process, we will learn some the basic and essential shell commands, and work through a number of examples common to researchers. This knowledge will form a foundation for the rest of the course. More generally, this introduction to the terminal will help us understand computers at a deeper level, an understanding that improves our productivity and capabilities when accomplishing tasks with computers.

1.2 Why learn the terminal?

Interacting with the computer through a terminal is quite a different interface for interacting with computers than we are used to (e.g. clicking on icons with a mouse, or, more recently, sliding and swiping with our fingers on a touch screen). Many newcomers to the terminal (myself included), initially find it unintuitive and obscure. Thus, let's briefly motivate why we are spending the first lecture learning the terminal.

- It's (sometimes) the only option
  - When consulting for a client, we often need to access their servers. Sometimes, the only way to work with client's data is to log in through the terminal.
  - There are many tasks, such as efficiently working with large data sets, that working with the terminal is the only feasible option
- It ain't goin' anywhere
  - The shell syntax has been unchanged over time. It is highly standardized, and the BaSH syntax is ubiquitous[1].
  - Since it is a stable tool, taking the time to learn it will payoff for years to come.
- It can be powerful
  - Once you get past the scariness of it, the terminal gives an abundance of new capabilities, allowing us to work with files, data, and code with unprecedented efficiency
- Documentation
  - Because BaSH is ubiquitous, it has great online documentation.

[1]Again, except for windows

2 Basic Commands

2.1 Directories & Files

Our first task is to understand **directories** and **files**, and we will start by comparing to folders and files.

In a Windows or Mac machine, our computer is comprised of files, which exist in folders (where "folders" are locations that store files, and possibly other folders). When we want to open a file, we traverse a list of folders, until the current folder we are in contains the file we are looking for.

In the terminal, the story is basically the same. The only difference is that "folders" are now called "directories", and the "current folder" is called the "current working directory".

At any given time, two natural questions may come to mind: (1) what directory am I in, and (2) what files are here?

To find out what directory you are in (i.e. the current working directory), we run `pwd` (i.e. **p**rint **w**orking **d**irectory). For example:

```
$ pwd
/foo/bar/
```

When we ran it, it gave us the **path** to the current working directory. Thus, we now know that the current working directory is `bar/`, which is a subdirectory of `foo/`.

To list all of the files in the current directory, we run `ls`:

```
$ ls
file1.txt        file2.txt        foobar/
```

The output tells us that, in the current working directory, there are two files (named `file1.txt` and `file2.txt`), and another subdirectory contained in `/foo/bar/` called `foobar/`.

Section 2.1 – Key Ideas

- Files are contained in directories
- At any given time, we are in a single directory, called the current working directory
- `pwd` tells us the location (i.e. path) of the current working directory
- `ls` tells us what files and subdirectories are in the current working directory

2.2 Changing Directories

Now that we know what directory we are in, and how to find out what files and subdirectories are in the current directory, the next task is to be able to move directories (in other words, change the current working directory). This is analogous to moving to a different folder on a Windows or a Mac.

The command `cd` gives us everything we need to change directories. We will show how it can be used a few different ways. Let's suppose, once again, that the current working directory is `/foo/bar`, which contains the files `file1.txt`, `file2.txt`, and a subdirectory `foobar/`.

First, suppose we want to move into a subdirectory of the current working directory. For example, suppose our current working directory is `/foo/bar`, and we want to move to the subdirectory `foobar/`. We can move to that directory with

```
$ cd foobar
```

We also verify our new current working directory:

```
$ pwd
/foo/bar/foobar/
```

To recap: by running `cd foobar`, we moved into the subdirectory `foodbar`. We then used `pwd` to verify that we changed our current working directory from `/foo/bar/` to `/foo/bar/foobar/`.

Next, suppose we want to move to the **parent directory** (i.e. the directory that contains our current working directory as its subdirectory). For example, suppose our current working directory is `/foo/bar/`, and we want to move to our parent directory, `/foo/`. For parent directories, the shell let's the double period, `..`, always refer to the parent directory. So, we can move to the parent directory with `cd ..`. For example:

```
$ pwd
/foo/bar/
$ cd ..
$ pwd
/foo/
```

We have now seen how to change to a subdirectory or the parent directory. But, a more general way to change the current working directory is to give the full (path) name of the directory we want to move to. Suppose we are in `/foo/bar/`, and want to move to `/foo/bar/foobar/`. Then, we can do

```
$ pwd
/foo/bar/
$ cd /foo/bar/foobar/
$ pwd
/foo/bar/foobar/
```

Finally, there is a directory called a **home directory**. This is the directory that we start in, whenever we first open up the terminal. The home directory does not behave any differently than a normal directory; however, for convenience, the shell lets ~ (the tilde character), be hardcoded to refer to the home directory. So, if we wanted to go to the home directory, we can run

```
$ cd ~
```

We stop to notice that the command `cd` is always followed on the same line by some text (namely, the directory we want to move to). `pwd`, on the other hand, is always on a line by itself. From here on, we will refer to that additional text as the `argument` of the command. `pwd` thus has zero arguments, and `cd` has one argument.

Section 2.2 – Key Ideas

- `cd` allows us to change the current working directory
- The parent directory is referred to by `..`
- The home directory is the directory that we start in when we first log in, and can be referenced by ~
- `pwd` and `ls` typically have zero arguments, and `cd` has one argument

2.3 Creating Directories and Files

On a Windows or Mac computer, we are familiar with creating a new folder, or deleting a folder. In the terminal, the same is true.

Suppose we are back in the directory `/foo/bar/`, and want to make a new subdirectory `barfoo`. This is accomplished with `mkdir`. For example,

```
$ ls
file1.txt      file2.txt      foobar/
$ mkdir barfoo
$ ls
file1.txt      file2.txt      foobar/      barfoo/
```

So, `mkdir`, followed by the name of the new directory, makes a new subdirectory.

Similarly, let's say we wanted to make a new, empty file, in the current working directory. This is accomplished with `touch`. For example,

```
$ ls
file1.txt      file2.txt      foobar/
$ touch file3.txt
$ ls
file1.txt      file2.txt      foobar/      file3.txt
```

The new file, `file3.txt`, is empty, which makes it kind of useless. We will see in a few sections from now how to edit it.

Section 2.3 – Key Ideas

- `mkdir` let's us create a new subdirectory
- `touch` let's us create an empty file in the current working directory

3 More Basic Commands

3.1 Copying Directories and Files

On a Windows or Mac computer, we are familiar with the notion of copying a file or folder. We usually right-click on a file or folder, click "copy". Then, somewhere else, right-click again, and click "paste". Here, we learn how to do the same tasks in the terminal.

First, copying a file. `cp` is the command for "copy". It takes two arguments: the name of the original file, and the name for the copy. For example,

```
$ ls
file1.txt       file2.txt       foobar/
$ cp file1.txt file1_copy.txt
$ ls
file1.txt       file2.txt       foobar/         file1_copy.txt
```

Often, when we want to make a copy of the file, we want to place the copy in a different directory. This can be done similarly. For example, suppose we wanted to make a copy of `file2.txt` in the `foobar` subdirectory:

```
$ ls
file1.txt       file2.txt       foobar/
$ cp file2.txt foobar/file2_copy.txt
$ cd foobar
$ pwd
/foo/bar/foobar/
$ ls
file2_copy.txt
```

If we wanted to make a copy of the file in the parent directory, we could similarly run

```
$ ls
file1.txt       file2.txt       foobar/
$ cp file2.txt ../file2_copy.txt
$ cd ..
$ pwd
/foo/
$ ls
bar/    file2_copy.txt
```

Now, what about copying directories? On Windows or Mac, when we copy a folder, we also copy everything in that folder. So, here, when we copy a directory, we also want to copy everything in that directory in (its files, its subdirectories, the files and subdirectories in its subdirectories, etc).

To copy directories, we will use the same `cp` command. But, since we want to copy everything in that directory, we need to add an additional argument, `-r`, which is called a **flag**. Flags are (usually optional) arguments to a command, that allow us to specify additional settings for the command. Here, the `-r` flag to `cp` should be thought of as "copy the directory, all the files in the directory, all the subdirectories in the directory, all the files in those subdirectories, all the subdirectories in the subdirectories, etc". In computer science, we refer to this sort of thing as "recursion", hence the `r` in the `-r`. Let's see this in action:

```
$ ls
file1.txt       file2.txt       foobar/
$ cp -r foobar/ foobar_copy/
$ ls
file1.txt       file2.txt       foobar/         foobar_copy/
```

Section 3.1 – Key Ideas

- `cp` let's us copy files and directories, and takes two arguments: the name of the original, and the name of the copy
- Flags are optional arguments for commands
- The flag `-r` is used with the command `cp` to recursively copy directories

3.2 Deleting and Moving Directories and Files

Our final category of basic commands is on deleting and moving directories and files. On a Windows or Mac computer, we delete folders and files, as well as move them from one location to another (which is equivalent to copying to a new location, and deleting the original).

The command to delete is `rm` and the command to move is `mv`. These commands behave quite similarly to the commands we have seen thus far. `rm` requires one argument of text, referring to the file or directory that is to be deleted. If the text refers to a directory, the `-r` flag isn't needed. Sometimes, the shell wants to make sure that we actually want to delete a directory and everything in it, so we add the additional flag `-f` to force everything to be deleted.

Similarly, the move command is `mv` command operates exactly like `cp`.

Unlike Windows and Mac computers, the terminal does not have a 'Recycle Bin' or 'Trash'. In other words, there is no temporary directory that stores files or directories that are deleted by `rm`, or removed in the process of a `mv`. Thus, we take great caution before using either command, because we don't want to accidentally lose files or directories that we might need later.

Section 3.2 – Key Ideas

- `rm` deletes files and directories. Sometimes, both of the flags `-r` and `-f` are needed to delete a directory and everything in it.
- `mv` is identical to `cp`, except it deletes the original file or directory
- `rm` and `mv` can't be reversed, so use with caution.

4 Working with Files

4.1 What are files

As we previously alluded to, a computer is comprised of only two containers that store information: files and directories[2].

This section is about files. We all have a basic notion of what files are, but before moving forward, let's briefly formalize our understanding.

In the previous section, we came to understand that directories are basically there to organize files. Files, in contrast to directories, are containers of information, in the form of text, like numbers, letters, and other symbols[3]. Files could contain data, code, programs. Simply put: all real information is stored in files: plain and simple.

By convention, files end with a **suffix** (also called the file extension), which gives us an indication of how to interpret the information stored in the file. A very brief list is below:

.txt	a file containing just simple text. May be data
.csv	a file containing data, where each line is a row of data, and each column of a row is separated by a comma
.py, .cpp, .jl	a file containing code in Python, C++, and Julia, respectively
.bin	a binary file, meaning that the information should be interpreted as just 0's and 1's
.pdf	a *portable document format*, is interpreted by a PDF reader

We remember, though, that the subscript on the name of the file doesn't change the fact that all files are just containers of information. Regardless of the subscript on the name of a file, a file should be thought of as just information in the form of text. The subscript is just a convention that gives us and programs an understanding of how to interpret the information stored in the file.

Section 4.1 – Key Ideas

- A computer is comprised of files and directories, which store files and other directories.
- A file should always be thought of as a container that stores information, regardless of the subscript in its name.
- Depending on the circumstance, the information in the file is interpreted as data, code, an application, etc

[2]This is not totally true, but is the right insight to have at this point. The actual story, of course, is more involved: directories themselves are files, albeit special ones. There are also other sorts of structures, like symbolic links in a file system and environment variables in the shell. However, the vast majority of the time, it is best to think of a computer as comprised of only files and directories, nothing else

[3]More correctly, a file is a container of a sequence of 1's and 0's (i.e. bits). Humans are bad at making sense of long sequences of 1's and 0's, so when the terminal shows us the contents of the file, it maps the bits into numbers, letters, and other characters. Different mappings can be used; traditionally, ASCII is used, which maps tuples of 8 bits, called a byte, into one of $2^8 = 256$ different symbols. In other circumstances, an even larger superset of ASCII, called Unicode, is used.

4.2 Looking into files

Let's return back to the terminal. In this section, we want to be able to see the information stored in the files. First, we recall our directory: `/foo/bar/`.

```
$ pwd
/foo/bar/
$ ls
file1.txt        file2.txt        foobar/
```

First, we use the `cat` command to see the contents of a file:

```
$ cat file1.txt
E to the U du dx, E to the X dx!
Cosine! Secant! Tangent! Sine!
3 point 1 4 1 5 9!
Integral, radical mu dv
Slipstick, slide rule, M.I.T.!
```

We now see that `file1.txt` contains the lyrics to an MIT song! The song is quite short, so the entire file fit on the screen. However, some files can be much longer. For example, a data file with millions of lines of data will overwhelm the terminal if we try displaying it with `cat`. Thus, `cat` is not always the best way to look at a file. When we are dealing with files with many lines, two other commands come in handy: `head`, which shows the first several lines, and `tail`, which shows the last several lines. Let's see them in action:

```
$ head -n 2 file1.txt
E to the U du dx, E to the X dx!
Cosine! Secant! Tangent! Sine!
$ tail -n 2 file1.txt
Integral, radical mu dv
Slipstick, slide rule, M.I.T.!
```

As we see with `-n`, sometimes flags take arguments themselves. In this case, the flag `-n`, followed by a number, tells `head` and `tail` how many lines to display from the beginning and from the end of the file, respectively.

In the beginning of the lecture, we came across the command `wc`. This command gives us some basic statistics about a given file. We now can understand everything from the beginning of the lecture.

Section 4.2 – Key Ideas

- `cat` displays the contents of the file
- `head` and `tail` display the first and last several lines in a file, and are useful when the file has many lines
- Flags can sometimes take arguments themselves

4.3 Text Editors

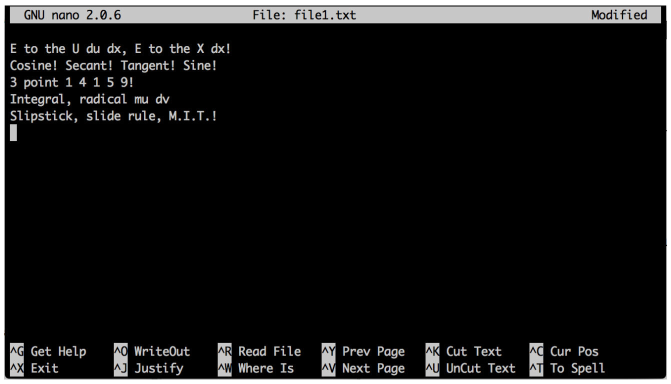
Now that we can see the contents of the file, we want to be able to edit the file (i.e. change the contents of the file). Thus, we need a program like Microsoft Word or TextEdit on Macs, that are specially designed to editing files.

These programs that allow us to edit files are called **Text Editors**. As we will see shortly, our shell has some text editors built in to it, so we can edit files without ever leaving the terminal. Two popular text editors are `Vim` and `emacs`. Both of these are powerful, and we highly recommend learning one of them at some point. However, both take some time to learn, so in the interest of time, we will just cover a simple editor, called `nano`.

We open the text editor with

```
$ nano file1.txt
...
```

Then, the terminal will open nano. The terminal will look something like:



```
  GNU nano 2.0.6                File: file1.txt                       Modified

E to the U du dx, E to the X dx!
Cosine! Secant! Tangent! Sine!
3 point 1 4 1 5 9!
Integral, radical mu dv
Slipstick, slide rule, M.I.T.!


^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Text  ^T To Spell
```

Here, we can navigate through the file with the arrow keys, and edit it with the keyboard. We save the changes with Ctrl-o, and exit nano with Ctrl-x.

Section 4.3 – Key Ideas

- A text edit is a program for editing files
- Vim and emacs are two popular text editors
- For these examples, we use nano for its simplicity

5 SSH

So far, we have been running a terminal to work on our own computer. However, there are many situations where we want to work on a remote computer. For example, if we are doing consulting work, we may want to log onto to the client's server to access their data or software. Or, if we are running a time-consuming computation, we often want to run it on a university server or on the cloud.

For these reasons, we want to be able to log on to another computer's terminal. This can be done from the terminal with the command `ssh`. SSH, also called a **Secure Shell**, is "an encrypted protocol to log onto a remote machine to communicate securely over an unsecure network". In simpler terms, SSH is a safe way to work on a remote terminal, as if it were our local computer.

To show how to use SSH, we will use the example of logging on to the Athena cluster. At MIT, every student has an account on the Athena cluster. To log onto Athena, we run the following command:

```
$ ssh athena.dialup.mit.edu -l mymitid
```

(note that `mymitid` should be replaced with your MIT username)

The first argument is the IP address for the remote machine. According to the MIT Athena website, `athena.dialup.mit.edu` is the IP address to SSH onto the Athena cluster. The `-l` flag says that the next argument will be the username.

Once logged in, our terminal will behave as if it is on the remote machine. That is, once we have SSH-ed into the remote machine, we won't be able to access anything on our local computer. When we are done on the remote machine, we can exit with the command `exit`.

Section 5 – Key Ideas

- SSH is used to work on a remote computer
- The first argument is the IP address of the remote computer, followed by `-l` and our username on the remote computer.
- To return to our local computer, we use the command `exit`.

6 Conclusion

We have now covered many of the fundamentals of the terminal. This is, of course, just the beginning. There are many more commands, flags, and programs like Text Editors in the shell, which expand the capabilities and increase productivity. So, we conclude with some accumulated knowledge and insights.

6.1 Getting unstuck

When using the terminal, most of us find ourselves frequently in the following situations:

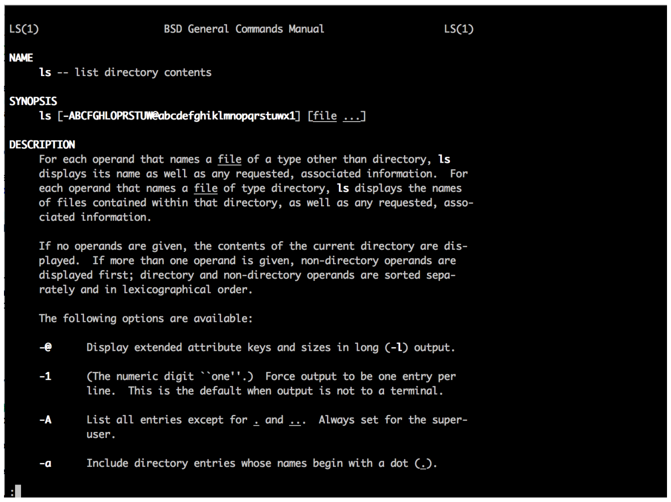
- We forget what flags exist for a command.
- We forgot the name of a command
- We find ourselves repeating the same tasks over and over, and want to be more efficient

When we get stuck, there are a number of resources at our disposal.

First, when we are unsure what flags exist for a command, or what exactly it does, we often check the **man pages**, which is a set of documentation on shell commands that comes built-in to the shell. For example, to see the flags associated with `ls`, running

```
$ man ls
```

opens up the man page for `ls`, which looks something like



```
LS(1)                     BSD General Commands Manual                    LS(1)

NAME
     ls -- list directory contents

SYNOPSIS
     ls [-ABCFGHLOPRSTUW@abcdefghiklmnopqrstuwx1] [file ...]

DESCRIPTION
     For each operand that names a file of a type other than directory, ls
     displays its name as well as any requested, associated information.  For
     each operand that names a file of type directory, ls displays the names
     of files contained within that directory, as well as any requested, asso-
     ciated information.

     If no operands are given, the contents of the current directory are dis-
     played.  If more than one operand is given, non-directory operands are
     displayed first; directory and non-directory operands are sorted sepa-
     rately and in lexicographical order.

     The following options are available:

     -@      Display extended attribute keys and sizes in long (-l) output.

     -1      (The numeric digit ``one''.)  Force output to be one entry per
             line.  This is the default when output is not to a terminal.

     -A      List all entries except for . and ...  Always set for the super-
             user.

     -a      Include directory entries whose names begin with a dot (.).

:
```

We can navigate the man pages by the arrows, and exit with the character `q`.

Unfortunately, the man pages can often be hard to understand. Then, making heavy use of Google is an essential. As we mentioned, the shell has been around for many decades. Therefore, when you get stuck, it is likely that other people have gotten stuck in the same way, so it is likely that an answer can be found by searching online.

Section 6.1 – Key Ideas

- The man pages are useful for learning the use and capabilities of a given command
- Google is an invaluable resource for finding more information on the terminal

6.2 'rm' is forever

We pause here to comment on `rm` and `mv`. When using either of these commands, the file or directory that is deleted really is gone; there will be no (easy) way of recovering them. Most seasoned terminal users have unfortunately learned this lesson the hard way. So, it is widely accepted that both commands should be used with caution, and should never be used hastily. Measure twice, cut once.

Section 6.2 – Key Ideas

- `rm` and `mv` are irreversible. Use with caution

6.3 Other information

The information we covered in this lecture sets the foundation for working in the terminal. However, after becoming comfortable with the basics, there are many commands, techniques, and tools to learn, that can greatly increase productivity and effectiveness in the terminal. We briefly mention a highly-incomplete subset of these further topics:

tmux This is a program that is very useful to use while SSH-ing into a remote machine. It let's us have multiple terminals open at once, allows the terminal to keep running even if our computer gets disconnected, and contains many other useful features.

Piping and redirection Piping in the shell let's us use the output of one command as the input to another command. Redirection let's us save the output of a command as a file, as well as use a file as an input to a command.

grep There is a set of commands, such as grep, that allow us to search in directories for files that contain some text in its name, or files that contain some text inside of them. These are invaluable timesavers.

Scripting When we want to automate a sequence of commands, we can save them as a "script". This can be done more simply with a `.sh` file. More powerful and popular scripting languages include Perl, AWK, and Python.