Lecture 3B: History, Variables, Quotes

History and a Word About Quotes

So to review: we showed how you can use arrow keys to browse your history, and use Ctrl+r to search your history. So it shouldn't a surprise to you that the commands you type in are saved. To view your complete history, you.... `history` .

```
2075  ll
2076  cd courseNotes/
2077  ll
2078  rm friendly*
2079  head -3 Acceptable-Use-Policy
2080  tail -3 Acceptable-Use-Policy
2081  ll *.png
2082  ll *.jpg
2083  file lecture2a.md
2084  file lecture2a-nautilus.png
2085  cp lecture2a-nautilus.png test.txt
```

you can call any of these commands using the **!** and the number from history:

```
 !2079
```

```
head -3 Acceptable-Use-Policy
```

```
--------------------------------------------------------------------------------
Policy :  Information Technology Acceptable Use
```

Another incredibly useful shortcut is `!!` . This will repeat the last command you typed in. This doesn't seem too useful until you start knowing something about permissions.

Review: The Three Types of Filepaths

- **Relative**: This is a filepath that is *relative to your current location*. For example: `cd cambridge` , `cd ../gen_ed` .
- **Relative-to_home**: This is a filepath that is *relative to a home directory* (usually yours). For example: `cd ~/Documents` , `cd ~uli101/a1` .
- **Absolute**: This filepath isn't *relative* to anything. It begins from root and never changes. For example: `cd /etc/shared` .

Let's assume that your filesystem looks like this:

```
/
├── etc
│   ├── network
│   │   └── interfaces
│   └── selinux
```

```
│       └── config
├── example.html
├── example.md
├── home
│   └── student
│       ├── Documents
│       │   └── resume.docx
│       ├── Music
│       └── Projects
│           └── Seneca
│               └── final-project.c
└── usr
    └── local
        ├── bin
        │   └── program.py
        └── lib
            ├── node
            └── python
```

...and you are given a question such as this:

> You are logged in as Student. Your current location is `/usr/local/lib`. Copy `program.py` into the directory called `Seneca`.

You will be required to use the shortest possible filepath for both the *target* and the *destination*.

Our target is `program.py`. Our current location (`.`) is `/usr/local/lib`. Our home (`~`) is `/home/student`. We have three possible filepaths.

The Target

- **Relative**: start from current location and "travel" to the target. `..` gets us to /usr/local. `bin` gets us into the proper subdirectory. `program.py` identifies the target. So the relative path is: `../bin/program.py`. (3 steps).
- **Relative-to-home**: start from `~`. Unfortunately our target isn't in a subdirectory of home, so this is going to get complicated! The relative-to-home path is: `~/../../usr/local/bin/program.py` (7 steps).
- **Absolute**: start from `/`. `usr` is a subdirectory of `/`. Then `bin` is a subdirectory of `usr`. So the absolute path is: `/usr/local/bin/python.py` (4 steps).

We can see that the relative path is the shortest, most efficient path to `program.py`.

The Destination

Again, for the destination we have three options.

- **Relative**: start from our current location again. `../../..` gets us to root, so already we can see this isn't a very efficient route. `home/student/Projects/Seneca` will get us to the destination. The complete relative path is `../../../home/student/Projects/Seneca` (7 steps).
- **Relative-to-home**: start from `~`. From here, we type `Projects` to get into a subdirectory, and `Seneca` is our destination. The relative-to-home filepath is `~/Projects/Seneca` (3 steps).

- **Absolute**: start from root (`/`). The absolute filepath is `/home/student/Projects/Seneca` (4 steps).

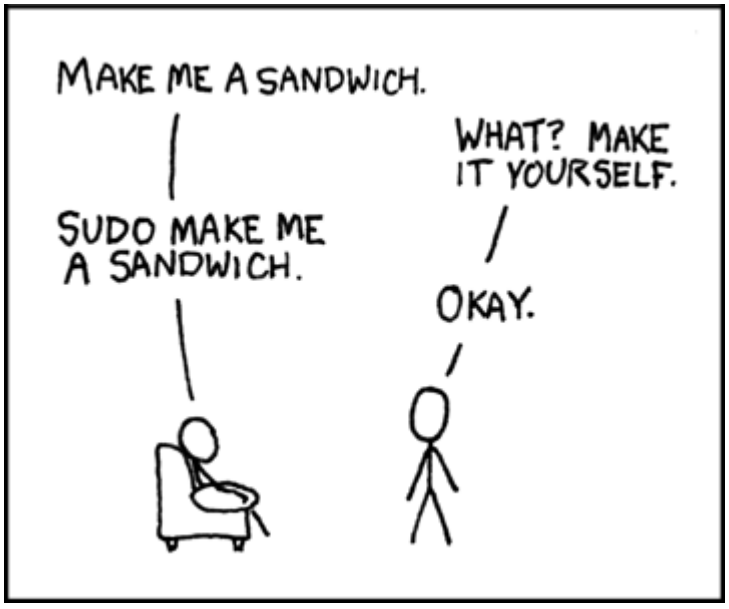
So for the destination, we should use a relative-to-home filepath.

Putting It All Together

Our answer for this question is: `cp ../bin/program.py ~/Projects/Seneca`

Sudo

So as a normal user, you don't have permission to install applications or change much of the system configuration. This is because you don't have *root privileges*. Oftentimes in the real world, you will type in a command, and have the shell tell you that permission denied. `sudo` is a command that invokes your root privileges for a command. So you have situations like this:



Relevant XKCD

This is a command to check for new software updates in Ubuntu:

```
apt update
```

```
E: Could not open lock file /var/lib/apt/lists/lock - open (13: Permission denied)
```

```
sudo apt update
```

```
sudo apt update
[sudo] password for eric:
Hit:2 http://ca.archive.ubuntu.com/ubuntu bionic InRelease
  Get:3 http://security.ubuntu.com/ubuntu bionic-security InRelease [83.2 kB]
  Hit:4 http://archive.canonical.com/ubuntu wily InRelease
  Reading package lists... Done
Building dependency tree
Reading state information... Done
All packages are up to date.
```

Instead of typing in 'apt update' twice, the same thing can be accomplished with:

```
sudo !!
```

This isn't too relevant for you yet, since in this course you won't be given root privileges. But it's a common issue with Linux users. And it will become relevant in the following situation:

```
echo "Hello User !!"
```

```
echo "Hello User sudo apt update"
Hello User sudo apt update
```

What just happened?

Well, the shell saw this: `!!` and performed the substitution that it thought was appropriate. Namely, it substituted `!!` with the last command I ran, which in this case was `sudo apt update`. **This is an important concept to keep in mind.** Putting a string in double quotes (") did *not* prevent the substitution from happening.

Variables And Quotes

Here's another example of the same thing. We've talked a bit about Bash being a programming language, and programming languages make use of *variables*. We have variables in Bash, and here's how they work.

```
school=Seneca  # variable name of school has just been given a value of "Seneca"

echo $school

Seneca
```

Finally, the `echo` command is becoming useful. To assign value to a variable, we use an equal sign (=). But to call that variable, we start with a dollar sign ($).

```
school=Seneca College

College: command not found
```

This throws us an error, because the shell is *interpreting* the second word 'College' as a command. Let's try again by putting the string in double quotes.

```
school="Seneca College"

echo "I have enrolled at $school!"
I have enrolled at Seneca College!
```

Double quotes indicate to the shell the boundaries of our string. You can think of them as a *weak* fence, but also notice how the dollar sign still gets *interpreted* not as a dollar sign, but something that identifies a variable.

Now let's use single quotes (').

```
 echo 'I have enrolled at $school !!'
I have enrolled at $school !!
```

Single quotes are a *strong* barrier. No variable or command substitution is allowed. Everything is very *literal*: ! are exclamation points, and $ are dollars signs.

Note: If you only wish to escape *one* character such as a dollar sign, use the escape character (₩).

```
 cost=100
echo $cost
100
echo "Wow, I can't believe this textbook costs \$$cost."
Wow, I can't believe this textbook cost $100.
```

If there is only one golden rule for Bash scripting, it is this: **use quotes**, and use the quotes appropriate for what you want to do.

Backstory: Open Source Software and GPL License

The concept of open source software (or free software) is a very important part of Linux. But what exactly does it mean? I'll pass along the definition of 'free software' from Richard Stallman. Free software follows four freedoms:

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help others (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

One important distinction you hear made is that free software is "free" as in "free speech", but not necessarily "free" as in "free beer." You also get into distinction between 'free software' and 'open source software', but for the most part we use the two terms interchangeably. Like a lot of things in the Linux world, there's a lot of politics and backstory if you feel like reading up on it!

What this means is that a lot of the software provided with your version of Linux is free: you can use it without cost, you can read its source code and choose to contribute to the project if you wish. And open source has had a huge impact on how computing has developed. The Internet would look very different (or would still be in its infancy!) if there had never been the LAMP stack, which is all open source.

- Linux
- Apache
- MySQL
- PHP

The GNU General Public License

Let's say you work on a project that you think might benefit other people, and decide to open source your project. There are several licenses available that you might consider. One of these is the *GPL*. In a nutshell, under a GPL license (version 3), anyone who modifies your code must also release their modified code under the GPL license. This would prevent, say, a company from taking your code, modifying it a bit, and then making it proprietary.

Software Licensing and ownership is a huge but interesting topic. I encourage you to read about it further.

Summary

Shortcuts

- !!: Repeat last command
- &&: Combine two commands; if the first is successful, run the second.
- !####: Repeat a command from history
- ₩: Escape character in Bash
- "": Weak Quotes. Allows Bash to expand $ and !
- '': Strong Quotes. $ and ! will be taken literally

Introduction to Variables and Quoting

- To assign a value to a variable: `var=value`
- To print the value of a variable: `echo $value`

Review For Quiz 1

- define *absolute/ relative/ relative to home* filepaths.

absolute: /home/eric/document. relative: ../document. relative-to-home: ~/document:

- What is *redirection*?

Redirection is when you *redirect* STDOUT or STDERR from your display to another location, like a file.

- you are in ~/cpr/lectures. You want to cp a file called ~/cpr/fall2018/gradebook to ~/Desktop. Do this with relative filepaths.

cp ../fall2018/gradebook ../../Desktop

- There are reports called 'report-YYYY-MM.pdf'. Use a find command to list all reports from 2017 and 2018.

```
find / -iname report-201[78]*.pdf 2> /dev/null
```

- how can you delete all hidden files that end with a number?

```
rm .*[0-9]
```