CS 45: Software Tools Every Programmer Should Know

*We're excited to announce that this class will continue as **CS 104: Introduction to Essential Software Systems and Tools** in the 2023-24 school year. CS 104 will be an upgraded version of CS 45 taught by Professor Achour in Winter Quarter. If what you see on this website looks interesting, you should check it out!*

Course Description

Classes teach you all about advanced topics within CS, from operating systems to machine learning, but there's one critical subject that's rarely covered, and is instead left to students to figure out on their own: proficiency with their tools. This course will teach you how to master the key tools necessary for being a successful computer scientist, such as the command line, version control systems, debuggers and linters, and many more. In addition, we will cover other key topics that are left out of standard CS classes, but that are essential to being a proficient computer scientist, including: security and cryptography, containers and virtual machines, and cloud computing.

General Information

This course meets in-person twice a week, Mondays and Wednesdays from 4:30 to 5:50 at 300-300. The course is offered for 2 units on a S/NC basis. For more information about the course structure, visit the Course Info page.

Computer Setup & Software Installation

This course will have about a 50/50 mix of conceptual background and hands-on practice with the tools we'll teach– this means you'll need to be able to download and install software onto your computer (either your personal computer, or a computer you have access to). Click here for more information about setting up your computer and the kinds of software we'll be using. (Let us know if this will present a challenge, e.g. if you're using a chromebook or a very old computer, or don't have access to a personal computer– we may be able to help 😄)

Course Staff

Staff Mailing List: cs45-spr2223-staff@lists.stanford.edu

Unless you're contacting us about OH or something else instructor-specific, please either use the staff mailing list or make a post on Ed.

Calendar

Week 1	**Topic**	**Materials**	**Assignments**
Mon, Apr 3	Lecture 1: Course Overview	slides lecture video assignment 0	
Wed, Apr 5	Lecture 2: The Shell and Shell Tools	slides notes lecture video	
Week 2			
Mon, Apr 10	Lecture 3: Data Manipulation	slides notes lecture video	
Wed, Apr 12	Lecture 4: Shell Scripting	slides notes lecture video assignment 1	Assignment 0 Due
Week 3			
Mon, Apr 17	Lecture 5: Text Editors	slides notes lecture video	
Wed, Apr 19	Lecture 6: Command Line Environment	slides notes lecture video assignment 2	Assignment 1 Due
Week 4			
Mon, Apr 24	Lecture 7: Compilers and Package Management	slides notes lecture video	
Wed, Apr 26	Lecture 8: Computer Networking	slides lecture video assignment 3	Assignment 2 Due
Week 5			

Mon, May 1	Lecture 9: Version Control I	slides notes lecture video	
Wed, May 3	Lecture 10: Version Control II	slides notes lecture video assignment 4	Assignment 3 Due
Week 6			
Mon, May 8	Lecture 11: Build Systems & DevOps	slides lecture video	
Wed, May 10	Lecture 12: Debugging and Profiling	slides lecture video assignment 5	Assignment 4 Due
Week 7			
Mon, May 15	Lecture 13: Security	slides lecture video	
Wed, May 17	Lecture 14: Cryptography	slides lecture video assignment 6	Assignment 5 Due
Week 8			
Mon, May 22	Lecture 15: Virtual Machines & Containers	slides lecture video final project	
Wed, May 24	Lecture 16: Cloud & Serverless	slides lecture video assignment 7	Assignment 6 Due
Week 9			
Mon, May 29	No Lecture: Memorial Day		
Wed, May 31	Lecture 17: Media Encoding	slides notes lecture video assignment 8	Assignment 7 Due
Week 10			
Mon, Jun 5	Lecture 18: Q&A and Conclusion		
Wed, Jun 7	No Lecture		Assignment 8 and Final Project Due

Attributions

Big thanks to the MIT CSAIL's The Missing Semester of Your CS Education taught by Anish Athalye, Jon Gjengset, and Jose Javier Gonzalez Ortiz– The Missing Semester was the original inspiration for CS 45.

Additionally, this website is based on versions of CS110 and CS111's website created by John Ousterhout and Jerry Cain– thank you!

Course Info

Lectures

Lecture attendance is optional, but highly recommended. Assignments will assume that you know the material from that week's lectures.

We will publish lecture notes as well as slides for each lecture for anyone who misses or cannot attend in-person class. We will try to post lecture recordings to Canvas on a best-effort basis, but we can't make any guarantees about their quality (or even existence).

Assignments

There will be 8 weekly assignments for the course as well as a final project. Assignments will usually be due on Wednesdays at 11:59 PM, unless indicated otherwise on the assignment handout or the course schedule. All assignments will be submitted on Gradescope. Each student will be allowed a total of 3 late days for assignments, which may be spent in units of one day (24 hours) on any assignment throughout the quarter. You do not need to inform us when taking late days; we'll use Gradescope submission times to figure them out. There will be no exams in this course.

Office Hours

Our office hours timings are listed on the homepage. We will try to maintain a regular weekly schedule, but we may have to make adjustments as the quarter progresses. They will be a mix of in-person and remote. You can also get help on Ed.

Grading Policy

CS45 is graded on a Satisfactory / No Credit basis. We expect that everyone should be able to earn a grade of satisfactory. Your course grade will be based on your assignment scores, final project score, and weekly surveys. You get an S if you earn 25 points in the class:

- Each weekly assignment (1 thru 8) is worth 3 points (Assignment 0 is worth 1 point).
- Each weekly survey is worth 0.5 points.
- The final project is worth 5 points

Honor Code

As in all Stanford classes, you are expected to follow the Stanford Honor Code. Work submitted for grading should not be derived from or influenced by the work of others. Do your own thinking, your own design, your own coding, and your own debugging. Any assistance you receive must remain within acceptable limits. Truthful citations must be made where required. All submissions are subject to plagiarism detection tools. Suspected violations are referred to the Community Standards office.

Students with Documented Disabilities

Students who may need an academic accommodation based on the impact of a disability must initiate the request with the Student Disability Resource Center (SDRC) located within the Office of Accessible Education (OAE). SDRC staff will evaluate the request with required documentation, recommend reasonable accommodations, and prepare an Accommodation Letter for faculty dated in the current quarter in which the request is being made. Students should contact the SDRC as soon as possible since timely notice is needed to coordinate accommodations. The OAE is located at 563 Salvatierra Walk.

CS 45, Lecture 2

Shell Tools

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Spring 2023

Outline

Contents

1 What is the Shell?

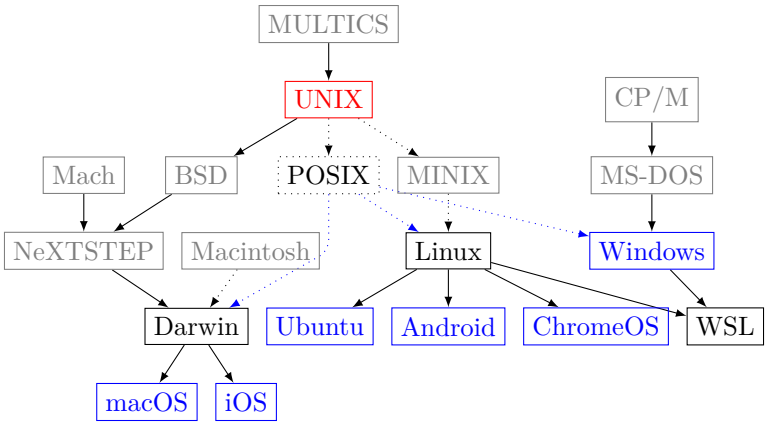
To understand what the shell is, we first have to understand the environment in which it was designed:

UNIX

- The shell (as we recognize it) began with the UNIX *operating system* in 1969.[1]
- UNIX was made at Bell Labs by Ken Thompson and Dennis Ritchie.
- UNIX introduced what is now called "the UNIX philosophy."
- Almost all modern computing is derived from the legacy of UNIX.

Modern Operating Systems

[1] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, jul 1974



In fact, every currently-mainstream operating system has some of UNIX in it. macOS and iOS are actual descendants of UNIX (in particular, the Berkeley Software Distribution, or BSD). Linux, the basis of ChromeOS, Android, and many "distributions" like Ubuntu, is designed to be UNIX-compatible (via the POSIX specification). Even Windows, which historically held an entirely different philosophy and design, has included some form of POSIX-compatibility for 30 years; nowadays, it takes the form of the Windows Subsystem for Linux.

Now, we said UNIX is an operating system, but what exactly does that mean?

UNIX Explained

Ken Thompson and Dennis Ritchie explain: `https://www.youtube.com/watch?v=JoVQTPbD6UY`.

Their breakdown is a good one (obviously, since they invented the thing), but it's explained pretty quick. Let's go into it piece-by-piece in more detail.

1.1 What is an Operating System?

An operating system, at its core, is an abstraction layer. A computer contains many different parts, each of which has a different purpose:

Anatomy of a Computer

Input Keyboards, Mice, *Serial Ports*, etc.

Output Screens, *Serial Ports*, Speakers, etc.

Storage Memory (RAM), Disks, Disc Readers, etc.

Compute CPUs (math), FPUs (math with decimals), GPUs (math with matrices)

Networking Ethernet, Wi-Fi, *Serial Ports*, etc.

Misc. Fans, Power Supplies, Sensors, etc.

Different computers have different parts serving these purposes. My laptop, for example, has very different hardware than the computer running Carta. However, they both use the same operating system kernel (i.e., Linux), and I can therefore run the same programs on both.

One recent example you might be familiar with—while older Macs used CPUs made by Intel, newer Macs use CPUs made by Apple. These CPUs are incompatible with each other; essentially, they speak different machine languages, and can each only understand programs written in their native language. However, since they both have the same operating system (macOS), programs written for an Intel Mac can run on an ARM Mac; macOS automatically translates programs written in the Intel language to the ARM language, hiding the low-level details from both the programs and their programmers.

Definition 1.1 (kernel)**.** An Operating System Kernel is a program that abstracts over different hardware, allowing the same software to run on different computers.

You might notice that serial ports show up quite a lot on this list. At the time UNIX was designed, serial ports were ubiquitous—any time you wanted to connect two devices together, you'd use a serial port. Serial ports at the time were like USB ports today—you'd use them for everything. Even today, you can still see the legacy of serial ports hiding in the design of most operating systems.

As an aside, computers didn't have screens at one point, not long before UNIX. Instead, they were connected (via a serial port) to a *typewriter*. When a computer "printed" output, it literally printed it out onto a piece of paper. Over time, these were replaced with "video terminals"; these things had a video screen instead of a piece of paper, usually tall enough to show 25 lines of text. While computer scientists still called it "printing", any output sent by the computer was instead displayed on the screen. Eventually, we progressed to graphical displays like the one you're reading this on. Instead of a dedicated video terminal, you can now open a "terminal emulator" (usually just called a "terminal"), which is a window that pretends to be a video terminal.

Userspace

- A kernel by itself is kind of useless.
- Abstractions are great, but we want to *do* something with our computers.
- This is where userspace comes in.

Definition 1.2 (userspace)**.** Userspace is the set of programs that come bundled with an OS kernel, which allow a user to perform various tasks.

Almost everything we'd call an "program" (or an "app") is a userspace program. While we can install programs/apps after-the-fact, every operating system must come with some set of them installed; otherwise, the operating system wouldn't be able to do anything.

An operating system's userspace almost always contains the following:

- A text editor.
- A clock.
- A calendar.
- A calculator.
- An email program.
- A web browser.
- A file browser.
- An "app store" (package manager).

Running Programs

- Now that we have a bunch of programs installed, we want to run them.
- We need something that wraps up all the programs and provides a common interface to them.

This common interface is what we call the shell.

Definition 1.3 (shell)**.** A shell is the outermost layer of an operating system; it lets a user run userspace programs, which in turn let a user interact with their computer's hardware.

Definition 1.4 (operating system)**.** An Operating System is the combination of a kernel, a set of userspace programs, and a shell.

Operating System	Shell	Type	How you start programs
Windows	`explorer.exe`	Graphical	Start Menu, Desktop
macOS	Aqua	Graphical	Dock, Launchpad
iOS, Android	Home Screen	Graphical	Tap icon
Linux	GNOME, KDE, XFCE, …	Graphical	Various
Windows	`cmd.exe`	Text	Type name of `.exe` file
UNIX	`sh`	Text	*The rest of this lecture.*
Linux	`bash`	Text	Same as `sh`
macOS	`zsh`	Text	Same as `sh`

Table 1: Shells across common operating systems

Modern operating systems have many different kinds of shells, depending on what they're used for. Some are graphical, some are text-based. While graphical ones are far more common in general nowadays, text-based ones are far more common for developers. Given that you're taking this class, you probably already know how to use graphical ones (you're using one right now to read this), and you probably want to know how to use a text-based one (which is the rest of this lecture!).

Types of Shell

Modern UNIX-like operating systems use more modern shells, like `bash` and `zsh`, but they're all backwards-compatible with the classic `sh`. While Table 1 lists the default on each platform, some platforms let you change the default shell, and there are many `sh`-compatible shells available.

Windows also has a more modern UNIX-shell derivative called PowerShell, which is designed to be a hybrid of the traditional Windows command line, the UNIX shell, and Windows' native object-oriented interfaces. It's not fully compatible with the UNIX shell, and is much less widely used, but it's worth checking out if you're curious about alternative shells; it has some cool features not found on traditional UNIX shells.

While all of these shells can *start* programs, only the UNIX shell (and its derivatives) can *combine* them.[2]

This unique property, especially in combination with The UNIX Philosophy, has resulted in the UNIX shell growing into a powerful software ecosystem.

1.2 The UNIX Philosophy

The UNIX philosophy is a way of thinking about what a program should do.

Original

As described in the Bell System Technical Journal in 1978 [3]:

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.

That's written very confusingly, but it's all just a really wordy way of saying this:

[2] Okay, technically the other shells *can* also combine programs to some extent, but the UNIX shell can do so to a much greater extent.

[3] M. D. McIlroy, E. N. Pinson, and B. A. Tague. UNIX time-sharing system: Foreword. *The Bell System Technical Journal*, 57(6):1899–1904, July 1978

Simplified

The UNIX Philosophy

Build lots of small tools, each of which does exactly one thing well, but which can be combined to do more powerful things.

The UNIX Philosophy is also deeply tied to the UNIX file abstraction. Remember how we said that the job of an OS kernel is to abstract over different hardware? Well, in UNIX (and UNIX-derived OSes):

1.3 The UNIX File Abstraction

The UNIX File Abstraction

- In UNIX, *everything is a file* (including hardware!).
- Most files are text.
- Programs which operate on text can operate on almost everything.

The idea that *everything is a file* is not an exaggeration. Want to write to your screen? Write to `/dev/fb` (the "framebuffer"). Want to get input from the mouse? Read from `/dev/input/mice`. Want to play a beep from the speaker? Write to `/dev/snd/pcm*`. While this has reduced a bit on modern OSes, and there are now higher-level abstractions for many things, UNIX-based operating systems model an insane number of different things as files.

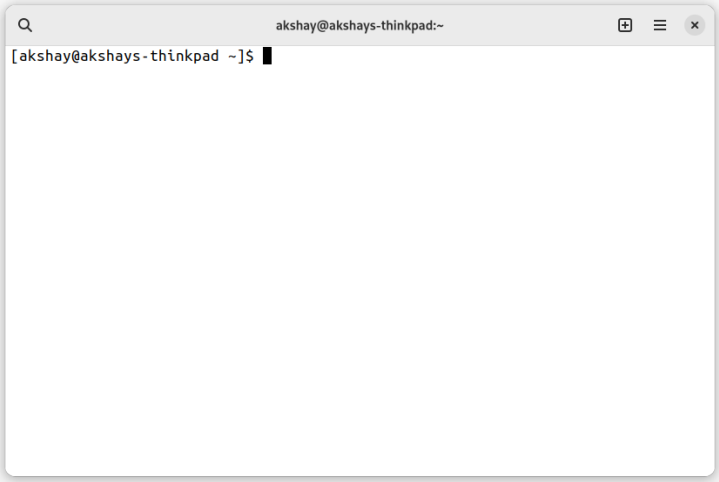
The combination of the UNIX Philosophy and the UNIX File Abstraction means we have hundreds of small, single-purpose programs which operate on text files, and we have every possible thing the computer could do represented as text files. The UNIX shell lets us combine these programs and these devices/text files to do… whatever we want, really.

2 The UNIX Shell

When you open the "Terminal" application on your computer, you'll be greeted by the UNIX shell (from now on, just "the shell"').[4]

The UNIX Shell

[4] You can open the "Terminal" application the same way you'd open a normal application; on macOS it's located in the "Utilities" folder of "Applications", on Windows it's in your start menu (it might be called "Ubuntu"), and on Linux it'll be in your desktop environment's normal app launcher.



In particular, the shell will give you a "prompt"; a sequence of information, followed by a cursor where you can type commands.

The Prompt

Example 2.1 (prompt). An default shell prompt might look like this:

```
[akshay@akshays-thinkpad ~]$
```

This PROMPT will print every time the shell is ready to accept another command. It probably looks different on your computer, since different shells have different defaults. Regardless of what it looks like right now, you can customize it to look like whatever you want. In most cases, you can get back to it by pressing CTRL-C on your keyboard.

Definition 2.2 (prompt)**.** The prompt is a sequence of information printed by the shell when it's ready to accept a new command.

Let's look at the pieces of information in more detail:

The Prompt

Username

Example 2.3 (prompt: username). This part of the prompt is your username:

```
[akshay@akshays-thinkpad ~]$
```

This is probably the same as the username you use to log into your computer. By default, you're auto-logged into the shell using your normal user account.

The Prompt

Hostname

Example 2.4 (prompt: hostname). This part of the prompt is your computer's hostname:

```
[akshay@akshays-thinkpad ~]$
```

This is your computer's name on whatever network it's connected to. Generally you don't really care about this unless you have multiple computers.

The Prompt

Current Directory

Example 2.5 (prompt: current directory). This part of the prompt is your current directory.

```
[akshay@akshays-thinkpad ~]$
```

This is your WORKING DIRECTORY; "directory" is just a fancy name for "folder", like you'd have in Windows Explorer or macOS Finder.

Definition 2.6 (directory)**.** A DIRECTORY is a folder. It can contain files and other directories.

Definition 2.7 (working directory)**.** Your WORKING DIRECTORY is the directory you're "in"; until you change your working directory, you can see only the files and folders in this directory. This is also often called your CURRENT DIRECTORY. It can be abbreviated as `..`

Just like in Windows Explorer or macOS Finder, you're "in" a particular directory at any point in time, in which case you can see the files in that directory.

By default, you start in your HOME DIRECTORY, which is the folder that contains `Documents`, `Downloads`, `Pictures`, `Videos`, etc. The home directory is abbreviated as a tilde (~) because it's so common.

Definition 2.8 (home directory)**.** Your HOME DIRECTORY is the directory all your files are in. Everything "under" your home directory belong to you, while everything "above" it belongs to someone else (probably the OS itself).

The shell is a command interpreter: you type in commands, and it does them (or gives you an error). Unfortunately, the commands were all named half a century ago when bytes were expensive, so the names cryptic and short. Let's go through a few useful commands to get started.

3 Basic Commands

3.1 Directories

Listing Files

In Windows Explorer or macOS Finder, the current directory is always visible. In the shell, you have to ask for the list of current files manually.

The LIST command is called `ls`.

Example 3.1 (ls). On my computer, the results look like this:

```
[akshay@akshays-thinkpad ~]$ ls
Desktop    Downloads  Music     Public     Videos
Documents  Dropbox    Pictures  Templates
[akshay@akshays-thinkpad ~]$
```

These are all the SUBDIRECTORIES of my home directory.

Definition 3.2 (subdirectory)**.** A SUBDIRECTORY is a directory contained within another directory.

You are here

- Before we go exploring, let's do one more command in the home directory.

The PRINT WORKING DIRECTORY command is called `pwd`.

Example 3.3 (pwd). Print the current working directory:

```
[akshay@akshays-thinkpad ~]$ pwd
/home/akshay
[akshay@akshays-thinkpad ~]$
```

Note that `~` is actually short for `/home/akshay`. Its PARENT DIRECTORY is called "home". The parent directory of "home" doesn't have a name at all!

Definition 3.4 (parent directory)**.** The PARENT DIRECTORY of a subdirectory is the directory which contains the subdirectory.

Definition 3.5 (root directory)**.** The ROOT DIRECTORY is the topmost directory on the filesystem. It's often called `/`.

Definition 3.6 (filesystem)**.** The FILESYSTEM is the hierarchy of directories which contain every file on your computer. Some of these files are "real", i.e., they exist on your hard drive, and some are "virtual", i.e., they're made up by the OS kernel to represent hardware.

N.B.: your computer might say something slightly different depending on your OS. I think macOS would say `/Users/<username>`.

Changing Directories

- The home directory isn't too interesting on its own.
- Let's go somewhere you probably know well: the Desktop!
- In Explorer or Finder you could just click on a folder to enter it.

The CHANGE DIRECTORY command is called `cd`.

Example 3.7 (cd). Change (`cd`) into the `Desktop` directory:

```
[akshay@akshays-thinkpad ~]$ cd Desktop
[akshay@akshays-thinkpad Desktop]$
```

Note that my prompt changed to say `Desktop` instead of `~`.

Not Changing Directories

- There's a special name that always means "the current directory": `.`
- `cd .` says "change directory to the current directory".

Example 3.8 (cd .). Don't change directories:

```
[akshay@akshays-thinkpad Desktop]$ pwd
/home/akshay/Desktop
[akshay@akshays-thinkpad Desktop]$ cd .
[akshay@akshays-thinkpad Desktop]$ pwd
/home/akshay/Desktop
```

Shortcut	Name
`~`	Home Directory
`/`	Root Directory
`.`	Current Directory
`..`	Parent Directory

Table 2: Directory Shortcuts

Command	Description	Argument	Required
`ls`	List Directory	Directory Name	No, defaults to `.`
`cd`	Change Directory	Directory Name	No, defaults to `~`
`pwd`	Print Working Directory	N/A	N/A
`mkdir`	Make Directory	Directory Name	Yes
`rmdir`	Remove Directory	Directory Name	Yes

Table 3: Directory Commands

From now on, I'm going to omit the part of the prompt before the `$` because it's not very useful for our purposes, and it varies from computer to computer anyway.

Making Directories

- Now that we're on the desktop, let's create a new directory to do some experiments in.
- This is the equivalent of right-clicking and selecting "New Folder".

The MAKE DIRECTORY *command is called* ***`mkdir`****.*

Example 3.9 (mkdir). Create a directory called "cs45-test-directory" and `cd` into it:

```
$ mkdir cs45-test-directory
$ cd cs45-test-directory/
```

You can go back to the Desktop by typing `cd ..` or `cd ~/Desktop`.

In general, `..` is a special name which always refers to the parent of the current directory. Regardless of where you are, you can always go to the parent directory by typing `cd ..`.

Now that we can make directories, you might be wondering how to delete them. The process is actually almost the same, just with a different command.

The REMOVE DIRECTORY *command is* ***`rmdir`****.*

This is used almost the same way as mkdir; one thing to note is that `rmdir` can only be used on an empty directory.

Let's review what we've covered about directories:

Directory Review

Definition 3.10 (argument)**.** An ARGUMENT is an input provided to a command. This may be a filename or arbitrary text, depending on the command. It is passed to a command by specifying it after the command name. By default, arguments are separated by spaces; if you want to provide an argument with a space, use double- or single-quotes around the argument.

3.2 Files

Now that we can create, delete, and move around directories, let's start working with files. As mentioned in subsection 1.3, everything in UNIX is a file, so (in a way) we're actually learning to work with *everything*.

However, before we work with files, let's take a brief diversion to look at something a little more basic: input/output.

Output

- Sometimes we just want to print something out, like "hello, world".

The PRINT *command is called* `echo`.

Example 3.11 (echo). Print the text "hello, world":

```
$ echo "hello, world"
hello, world
$
```

The name "echo" is a bit less intuitive than the ones we've seen before, but it makes sense: it "echoes" back anything you tell it.

If you're familiar with programming in C, you might also like the PRINT FORMATTED command, `printf`. You can use it the same way as the C `printf` function, except you don't need parentheses.

Example 3.12 (printf). Print the text "hello, world" using `printf`:

```
$ printf "%s, %s\n" "hello" "world"
hello, world
$
```

Input

- Sometimes we want to read input from the user.
- Unfortunately, this is trickier: there are multiple commands which can be used for input. Let's use the simplest one.

The CONCATENATE *command is called* `cat`. It can also be used for input.

Example 3.13 (cat). Read text from the user:

```
$ cat
this is a test
this is a test
line 2
line 2
$
```

The `cat` command will print out whatever you type into it… forever. To get it to stop, you can either "kill" it by pressing CTRL-C, or tell it "end of file" by pressing CTRL-D. Note that this is actually the CONTROL key, even on Macs. Using the COMMAND key will not work.

Aside: Windows and Linux users might be wondering how you copy and paste from the terminal if CTRL-C kills programs instead of copying anything. The answer is: it's complicated. The UNIX shell predates

CTRL-C and CTRL-V for copy-and-paste, so they use those shortcuts for other things. Different terminal emulators (the program called "Terminal" on your computer) have different ways of doing copy-and-paste; mine uses CTRL-SHIFT-C and CTRL-SHIFT-V, but yours might use something else. Mac users don't have to worry about this, since they can use CMD-C and CMD-V as usual.

Enough on I/O (as input/output is often called); let's get back to files.

Creating Files

- There are a few different ways to create files. Let's start with the simplest.

The TOUCH FILE command is called, ***`touch`****.* While we don't care about "touching" files as such, it has the handy side effect of creating files.[5]

Example 3.14 (touch). To create a file called "text.txt":

```
$ touch test.txt
$ ls
test.txt
$
```

We can also rename or move the file we just created.

Renaming Files

The MOVE FILE command is called ***`mv`****.* It can also be used to rename files.

Example 3.15 (mv). To rename a file called "text.txt" to "empty.txt":

```
$ mv test.txt empty.txt
$ ls
empty.txt
$
```

Deleting Files

The REMOVE FILE command is called ***`rm`****.*

This is irreversible!

This command is dangerous! It does **not** move the file to a "trash" folder; it permanently and irreversibly deletes it.

Example 3.16 (rm). To remove a file called "text.txt":

```
$ rm test.txt
$
```

[5] "Touching" a file means changing its last-modified timestamp to the current time; essentially like opening a file, saving it, and closing it again. This isn't useful very often, so most people think of `touch` as the "create file" command.

Writing to Files

- We have a problem though: the file we created is empty. We can't do much with a bunch of empty files.
- We can check this by running `ls` with a special FLAG asking for extra info (including the file size).

Definition 3.17 (flag)**.** A FLAG is a special argument to a function which configures its behavior. By convention, flags can be short (a single letter) or long (a word), and start with either one or two hyphens to distinguish them from normal arguments.

Example 3.18 (ls -l)*.* To print extra information about files:

```
$ ls -l
total 0
-rw-r--r-- 1 akshay akshay 0 Dec 18 11:59 test.txt
```

This prints out a bunch of information we don't really care about right now: the file permissions, the file's owner, the file's group, and the file's last-modified time. What we do care about is the "0" highlighted in red above: the file is zero bytes long.

As an aside, the real purpose of the `touch` command we just used is to update the modification time of the file. If you touch the file and run `ls -l` again, you should see the last-modified time update.

One very useful flag which is supported by almost every command is `--help`. This will usually print out information about how to use that command. On some commands the short form `-h` will also work, but that's sometimes used for something else.

Writing to Files

- Everything is a file, including the output of our commands.
- By default, this is called STANDARD OUTPUT, and goes to our terminal.
- The shell lets us REDIRECT standard output to go to a file instead.

Example 3.19 (output redirection)*.* To create a file called "hello.txt" with the contents `hello, world`:

```
$ echo "hello, world" > hello.txt
$
```

To append to an existing file, you can use `>>` instead of `>`.

Reading from Files

- Just like STANDARD OUTPUT, the input to our programs is also a file.
- By default, this is called STANDARD INPUT, and comes from our terminal.
- The shell also lets us REDIRECT standard input to come from a file.

Example 3.20 (input redirection)*.* To print a file called "hello.txt":

```
$ cat < hello.txt
hello, world
$
```

Operator	File	Overwrite?
`<`	`/dev/stdin`	
`>`	`/dev/stdout`	Overwrite
`>>`	`/dev/stdout`	Append
`2>`	`/dev/stderr`[7]	Overwrite
`2>>`	`/dev/stderr`	Append

Table 4: UNIX Shell Redirection Operators

This is actually slightly redundant, since the `cat` command lets us specify a specific file to use as input directly, so `cat hello.txt` would do the exact same thing. The redirection approach is more general though: it works with any command that accepts input.

When the shell runs a program, it sets up its input and output using special files.[6]

I/O(/E?)

Definition 3.21 (standard input)**.** STANDARD INPUT (`/dev/stdin`) is the file from which a program reads its input.

Definition 3.22 (standard output)**.** STANDARD OUTPUT (`/dev/stdout`) is the file to which a program writes its output.

Definition 3.23 (standard error)**.** STANDARD ERROR (`/dev/stderr`) is the file to which a program writes its error messages.

Since these files are so widely used, the shell provides easy ways to redirect them. We've already seen a few of these.

Redirection Operators

You might have seen the `>` operator before, in the form `> /dev/null`. `/dev/null` is a special file that discards anything written to it, so it's common to use it when the output of a command is unimportant.

As it turns out, if we're doing multiple commands in a sequence, we don't even need to write the results out to a file until the very end—we can use PIPES to send the output of one command directly into another command.

4 Pipes

Environment Variables

Some programs need configuration that's too annoying to provide as arguments every time.

Definition 4.1 (environment variable)**.** An ENVIRONMENT VARIABLE is a configuration value that's set globally by a program, which applies to itself and any other programs it runs.

We'll talk about environment variables some more in *Lecture 5: Command Line Environment*. For now, we just need to know that they exist and are widely used.

All the Environment Variables

The ENVIRONMENT VARIABLE command ***`env`*** *prints all the environment variables which are currently set.*

[6] Technically, these are FILE DESCRIPTORS rather than files, but the distinction isn't very important for this class so we're glossing over it. If you're interested in learning more, come talk to us or take CS111.

Example 4.2 (env). To print every environment variable:

```
$ env
MAIL=/var/spool/mail/akshay
PWD=/home/akshay
XDG_SESSION_TYPE=wayland
PATH=/usr/local/bin:/usr/bin:/usr/local/sbin
HOME=/home/akshay
USERNAME=akshay
[...]
```

This is quite long on my computer…

Connecting Programs

- Let's see how many environment variables we have!
- *The* WORD COUNT *command is called* `wc`.
- It has a flag `--lines` (or `-l` on Macs) which counts lines in its input instead of words.

Now, we want to get the output of `env` into `wc`. There's two ways we could do this:

Example 4.3 (count environment variables with a temporary file). We can write the output into a temporary file, and give it as input to `wc`:

```
$ env > /tmp/env.txt
$ wc -l < /tmp/env.txt
78
```

This is kind of annoying though…we need to do two separate commands, and we need to create a temporary file somewhere.

As an aside, it's conventional to put temporary files in `/tmp`. Most systems will keep `/tmp` entirely in RAM, so any files there will automatically get deleted when you reboot.

However, we can actually do this in a one-liner, without needing any temporary files at all.

Pipes

We want to send the output of `env` into `wc -l`:

Example 4.4 (count environment variables with a pipe). We can connect the output of `env` and the input of `wc` with a PIPE:

```
$ env | wc -l
78
```

Benefits of Pipes

Definition 4.5 (pipe)**.** A PIPE is a direct connection between the output of one program and the input of another. It can be set up using the `|` (pipe) operator, which connects `stdout` of whatever is on the left with `stdin` of whatever is on the right.

Pipes are superior to temporary files for several reasons:

- They are **parallel**: the programs on the left and right can run at the same time. This is especially useful when the programs need to do lots of computation or external I/O; the right-side program can process the data it has already read, while the left-side program is waiting for or computing data to write.
- They are **lazy**: the program on the right can read exactly as much data as it needs from the program on the left. In many cases, the right-side program may only need part of its input, in which case the left-side program doesn't need to compute the rest. This is especially useful when the left-side program is infinite or long-lived.

More Piping

Example 4.6. <only@1>[the first n environment variables] With the `head` command, we can extract only the first few lines from a file:

```
$ env | head --lines=3
MAIL=/var/spool/mail/akshay
PWD=/home/akshay
XDG_SESSION_TYPE=wayland
$
```

Example 4.7. <only@2>[random numbers] We can lazily evaluate part of an infinitely long "file" such as `/dev/random`:

```
$ cat /dev/random | hexdump | head --lines 1
0000000 4730 003c 6c22 1d16 49ef 6eff 91b2 a9f0
```

5 Conclusion

Getting Help

- The shell is far more complicated than we can possibly cover in an 80-minute lecture (or even a quarter-long class, honestly).
- Today's lecture was just the starting point—try things out and explore! Just like anything other skill, it's super important to practice using the shell on your own. It'll feel slow and clunky at first, but you'll get the hang of it soon!
- Most commands have lots of flags and options.
- We already talked about the `--help` flag, which usually gives you a brief summary of how to use a command.

The System Manual

- One super-useful resource is the UNIX system manual, which is pre-installed on most UNIX-like systems.
- *The* MANUAL *command is* ***man***; it takes as an argument the name of a command, and it displays the manual page ("man page").
- If you don't know the name of a command, you can search the manual using the command `apropos` (or, equivalently, `man --apropos`).

Example 5.1. <only@4>[man wc] To open the UNIX manual page for the `wc` word-count tool:

```
$ man wc
```

Example 5.2. <only@5>[man man] To open the UNIX manual page for the manual itself:

```
$ man man
```

Man pages open in what's called a PAGER; this is a program that makes a long file readable on a short terminal. You start at the beginning of the file, and you can scroll up or down using the arrow keys. You can quit by pressing `q`.

Other Useful Resources

- Practice is vital: try doing file management from the terminal. We didn't cover every command you'll need, so if you don't know how to do something, try searching the manual using `apropos` or searching the web.
- Use `man` pages, `http://cheat.sh/`, or `https://devhints.io/bash` to find out more about shell commands.

Be Careful!

The shell often doesn't warn you when you're doing dangerous things! Be sure to read the `man` page before running commands you find on the internet. Be especially careful with the REMOVE FILE command, `rm`, or when using the `>` (overwrite) operator.

Normally, most OS-critical files on your computer are protected using FILE PERMISSIONS, which we'll learn about in a week or two. However, there's a special command (which we saw briefly on Monday) which can override file permissions and do potentially catastrophic things.

Sudo

Using `sudo` responsibly

- Some commands require the use of `sudo`, the SUPERUSER DO command.
- This gives that command full access to do anything on your computer!
- Sometimes `sudo` won't even ask for your password!
- If you're using `sudo`, make sure you know what the command after it will do.
- `sudo` is necessary for certain tasks (we'll see some in the next few lectures), but it's always good to be careful around it.

Definition 5.3 (superuser)**.** The SUPERUSER is a special account, often called the ROOT ACCOUNT, with full power over a UNIX system.[8] They are the equivalent of the Windows Administrator account. Some regular accounts, called "sudoers", have the ability to act as the superuser using the `sudo` command.

And it's not just me who thinks that `sudo` can be dangerous, the first time you run `sudo` it'll give you a warning about how dangerous it is:

[8] Recent versions of macOS have an extra layer of protection, called "System Integrity Protection", that stops even the superuser from doing *really* dangerous things. However, the superuser can still do a lot of things an ordinary user cannot.

Sudo Warning

```
We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.
```

All that said, there are legitimate reasons to use `sudo` (for example, installing software). Just be careful around it, and don't run commands as `sudo` just because you got a "permission denied" error; make sure there *is* a reason you have to use `sudo`.

Here are a few interesting commands to look up/try out before next class; try looking up their `man` pages and how to use them:

Interesting Commands

`head`: Get the beginning of a file (or pipe).

`tail`: Get the end of a file (or pipe).

`grep`: Search within a file.

`sed`: Find-and-replace.

`cut`: Get a specific "column" of a file (e.g., a CSV file).

`ping`: Test your internet connection.

`sort`: Sort lines in a file.

`uniq`: Remove duplicate lines in a file.

`exit`: Exit the terminal.

Note that `uniq` requires its input to be sorted; it's not super clear why the authors made this decision, it's generally good practice to pipe through `sort` before piping into `uniq`.

Questions?


It's a UNIX system. I know this.

References

[1] M. D. McIlroy, E. N. Pinson, and B. A. Tague. UNIX time-sharing system: Foreword. *The Bell System Technical Journal*, 57(6):1899–1904, July 1978.

[2] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, jul 1974.

CS45, Lecture 3: Data Wrangling

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Winter 2023

Contents

1 Lecture Overview

In Lecture 2, we learned about the shell and how to run basic commands in the shell such as `ls`, `cd`, `cat`, `man`, and `wc`. We also learned about how to use the `|` operator to chain commands together. Finally we learned how to redirect output using `<` and `>`, and to append output to the end of a file using `>>`. In today's lecture we will learn how to combine these commands in powerful ways in order to automate tasks effectively, specifically in the context of data manipulation and analysis.

2 What is Data Wrangling?

Have you ever had large amounts of data and needed to look for a specific set of information? Of course you have. Have you ever done so in a tedious, inefficient way, even though you knew there is a better way of doing so? Bets are you have. In this lecture, we learn about how to manipulate data to suit our needs in the most efficient way possible.

The basic idea of data wrangling is that you take some data and convert or transform it into another form that is more useful. Often times, this "other" form is a condensed or sorted subset of the original data.

With the increased reliance on large amounts of data (i.e. "big data"), data wrangling has become increasingly important for effectively analysis.

3 Data Formats

Before we discuss data wrangling itself, it's worth discussing different data formats. The data wrangling technique you choose to use will heavily depend on the file format you are using to store your data. Here are some common file formats that are used to store data and that we will discuss in this class:

- CSV
- XML
- HTML
- JSON
- TXT

A **CSV** file is a comma-separated values file where information is separated by commas. CSV files are plain text files which make them easy to generate. They allow data to be saved in a tabular format (meaning in a table, with rows and columns). CSV files are useful as they can be opened with different applications and are not application specific. CSV files are most often used to analyze data with spreadsheets.

A **XML** file is an Extensible Markup Language (XML) file that is used to store data in hierarchical format. XML files were created for storing documents in a way that both humans and machines could read. A XML file consists of tags that define a hierarchy within the document.

A **HTML** file is a Hypertext Markup Language file that is used to store data in hierarchical format, specifically for a webpage. HTML files are very similar to XML files in that they are both read by both humans and machines and they both contain tags to define a hierarchy within the document. The key difference between XML and HTML files is that HTML files use a predefined set of tags while XML files do not have any constraints on what tags can be used.

A **JSON** file is a JavaScript Object Notation file that stores structured data in the form of JavaScript objects. JSON files are often used for transmitting data in web applications.

A **TXT** file is a plaintext file that contains data in the form of lines. TXT files have no special formatting (bold, italic, etc.).

4 `grep` and `sed`

Now onto data wrangling! We've already seen a basic example of data wrangling with the `|` operator. Recall that the pipe operator is used to transfer the output of one command to become the input of another command. Consider the command `ls /Documents | grep -i transcript`. This command lists all of the files in my `Documents` folder and searches for files that mention transcript (case insensitive). In this case, we start with a command to produce the data (`ls /Documents`) and then we use a second command to wrangle the data (`grep -i transcript`) by searching for files that contain the word transcript. You will find that the `|` command is often used for data wrangling in this way.

One common use case for data wrangling with the `|` operator is when looking at logs. System logs keep a record of operating system events on a machine. As you can imagine, system logs record a *lot* of information. It is infeasible to read through an entire system log, which makes it a perfect candidate for data wrangling.

In order to display a system's log, you can use the `log show` command for macOS, or the `journalctl` command for Linux. Once we have our log, we can use the `grep` command in order to search for log entries of interest. You might consider running `log show | grep -i Chrome` in order to show all log entries related to Chrome. Logs produce a lot of data so we may want to limit the amount of data we are interested in. Let's limit the amount of data to log entries just from the past 24 hours using `log show --last 1d`. Now we can search for all entries mentioning Chrome in the past 24 hours.

We can even use this model to search for entries on a remote server. We'll talk more about `ssh` in a later lecture, but for now, we just need to know that we can use `ssh` to log into a remote machine. I will use `ssh` to log into a "honeypot" server that the CS45 staff set up. A "honeypot" is a server or machine that attempts to lure potential attackers by inviting those attackers to log on or access said machine or server. We can search for everything related to ssh on our honeypot server: `ssh adrazen@192.9.152.85 journalctl | grep sshd`. Note that we are using a pipe to stream a remote file (i.e. our remote file is the system log on the honeypot server) through `grep` on our local computer.

This is still a lot of content. Let's imagine we are interested in just looking at when a user was disconnected from our honeypot server. In that case we might want to search for the phrase "Disconnected from". In this case, we could use

something like this:
`ssh adrazen@192.9.152.85 journalctl | grep sshd | grep "Disconnected from"`

This will work but we can do better. Given we are using a remote machine, we will be sending the data from the log back to our local machine in order to run `grep sshd` and `grep "Disconnected from"`.

Instead, we should try to run the entire data wrangling pipeline on our remote machine in order to avoid having to send data across. We can do this by adding quotes:
`ssh adrazen@192.9.152.85 'journalctl | grep sshd | grep "Disconnected from"'`

This is still fairly noisy. Perhaps we are are just interested in knowing *who* was disconnected. We can use another tool here called `sed` to parse out the usernames of users who were disconnected from the honeypot server.

`sed` is a stream editor built into Unix. `sed` can be used for searching a file (similar to `grep`), deleting lines from a file, adding lines to a file, and substituting text in a file. We will be using `sed` for substitution, known as the `s` command.

The `s` command in `sed` uses the following pattern `s/REGEX/SUBSTITUTION` where `REGEX` represents a regular expression for the phrase we are looking to matched, and `SUBSTITUTION` represents what we want to change the matched phrase to. In our case, we are looking to take out some noise from our log entries so we can use the following `sed` command to remove redundant information about which machine the error came from:
`sed 's/.*Disconnected from // '`

We can now add this to our data wrangling pipeline as follows:
`ssh adrazen@myth.stanford.edu journalctl | grep sshd | grep "Disconnected from" | sed 's/.*Disconnected from //'`

If we are interested in removing noise from only the first few log entries, we can specify a range of lines in our `sed` command. Perhaps we are only interested in removing noise from the first 10 log entries: `sed '1,10 s/.*Disconnected from //'`

This is good, but it isn't good enough. We are interested in only the usernames of users who signed in to our machine. We will need to write a regular expression to match everything in the line except the username.

5 Regular Expressions

A **regular expression** is a series of characters that specifies a search pattern. Most ASCII characters carry their normal meaning but some characters have special matching behavior. There is some variation between different implementations of regular expressions, but here are some general patterns. First, let's look at groups of characters, which specify *which* characters we are interested in:

- `.` means any one single character (except the newline character)
- `[abc]` means any of the characters included inside the square brackets, which in this case would be `a`, `b`, or `c`
- `[a-z]` means any characters in the range `a` through `z`
- `(a|b)` means either `a` or `b`

Next, let's look at quantifiers, which specify *how many* characters we are interested in:

- `*` specifies that 0 or more of the proceeding characters match
- `+` specifies that 1 or more of the proceeding characters match
- `{X}` specifies that exactly `X` characters should match

Finally, we can use *anchors* to specify what we expect at the beginning or end of a multi-component pattern:

- `^` specifies the start of the line
- `$` specifies the end of the line

Getting used to regular expressions can be very tricky so we recommend using a RegEx cheat sheet as well as a RegEx tester.

5.1 RegEx Example: Email Matching

Let's take a look at an example of using a regular expression to match email addresses. (The regex I will be presenting here does not actually match *all* email addresses but it will 99% of them.)

Let's take my email (`adrazen@stanford.edu)` and try to write a regular expression to match it. The easiest approach to writing a regular expression is to find delimiters to help divide the text you are looking to match into manageable chunks. In the case of my email address, we can choose `@` and `.` as our delimiters.

Let's first focus on writing a regular expression for everything that comes before the `@` sign. In other words, let's write a regular expression to match the text `adrazen`. We can begin by thinking about *which* characters are allowed. We know that this first part of an email address can consist of characters `A` through `Z` (both upper and lower case), any digit `0` through `9` as well as `.`, `_`, `%`, `+`, and `-`. Thus, the allowed character set is `[A-Za-z0-9._%+-]`. Next, we will want to think about *how many* of these characters are allowed.

Given that the username portion of the email address can be as long as you want (more or less), we will say that we can use as many of these characters as you want as long as you have at least one. (Technically, the username portion

of the email address should be 64 characters or less.) Altogether, this gives us the following for the first portion of the email address: `[A-Za-z0-9._%+-]+`.

Next we will consider the portion of the email address that is between the `@` symbol and the `.`. This refers to the email domain name and is allowed to consist of any of the following characters: characters `A` through `Z` (both upper and lower case), any digit `0` through `9` as well as `.`, and `-`. The allowed character set is therefore `[A-Za-z0-9.-]`.

Again, the domain name is allowed to be more or less as long as we want, but at least 1 character long. (Technically, the username portion of the email address should be 255 characters or less.) The regex to match this portion of the email address is: `[A-Za-z0-9.-]+`.

Finally, we have the component after the `.`. This is known as the top-level domain (TLD). According to rules for the TLD, it must be at least 2 characters long and is allowed to consist of any alphabetic characters. Thus, the regex to match this portion of the email address is `[A-Za-z]`.

Finally, we can combine these individual regexes together and we get the following: `[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+[A-Za-z]{2,}`.

5.2 Extracting usernames

Now that we have a basic understanding of how regexes work, let's return to our initial example of trying to extract the usernames from our log file on the honeypot server. The lines in our log file look something like this:

```
Jan 13 20:58:45 honeypot sshd[70942]: Disconnected from authenticating user root 52.142.11.171 port 1024 [preauth]
Jan 13 20:59:25 honeypot sshd[70946]: Disconnected from invalid user dachuang 158.101.97.210 port 48426 [preauth]
Jan 13 20:59:44 honeypot sshd[70949]: Disconnected from authenticating user root 143.110.153.150 port 42314 [preauth]
Jan 13 21:00:24 honeypot sshd[70951]: Disconnected from authenticating user root 5.78.40.253 port 56548 [preauth]
Jan 13 21:00:46 honeypot sshd[70953]: Disconnected from invalid user weiwei 170.64.134.218 port 38632 [preauth]
Jan 13 21:00:53 honeypot sshd[70955]: Disconnected from invalid user web_proj 143.110.153.150 port 36600 [preauth]
Jan 13 21:01:04 honeypot sshd[70958]: Disconnected from authenticating user root 92.27.157.252 port 33673 [preauth]
Jan 13 21:01:29 honeypot sshd[70963]: Disconnected from invalid user klaus 52.142.11.171 port 1024 [preauth]
Jan 13 21:01:39 honeypot sshd[70965]: Disconnected from authenticating user games 5.78.40.253 port 38640 [preauth]
Jan 13 21:01:59 honeypot sshd[70967]: Disconnected from authenticating user root 143.110.153.150 port 59124 [preauth]
Jan 13 21:02:22 honeypot sshd[70969]: Disconnected from authenticating user root 92.27.157.252 port 45223 [preauth]
Jan 13 21:02:45 honeypot sshd[70971]: Disconnected from invalid user es 5.78.40.253 port 54572 [preauth]
Jan 13 21:02:46 honeypot sshd[70973]: Disconnected from authenticating user root 170.64.134.218 port 38602 [preauth]
```

Our goal is to extract the usernames from this data. To do this, we will first write a regular expression to match the entire line. We will then work on extracting the username from that regular expression.

Let's divide each line into three components. First, we have the component before the username (e.g. `Jan 13 21:02:46 honeypot sshd[70973]: Disconnected from authenticating user`). Then, we have the username itself (e.g. `root`). Finally, we have everything after the username (e.g. `5.78.40.253 port 54572 [preauth]`). For the first component, we can write the regular expression as follows: `.* Disconnected from (authenticating |invalid )?user`. For the username, we will assume that the username can consist of any characters. Thus, we will use the following pattern to match the username `.*`. Finally, for the component after the username, we will use the following expression:
`[0-9.]+ port [0-9]+ (preauth)?`

Now that we have matched the entire line, we want to extract out the username. In order to extract the username, we first need to identify where in our RegEx our username is located. If we look at our RegEx, we will find that portion in the middle with `.*` is used to match our username. We can now use a capture group to, well, capture the username and then use it later. Generally speaking, in a regular expression, a **capture group** is any grouping inside of parentheses which allows us to remember a value in order to reuse it later. In fact, our regular expression already consists of two capture groups: `(authenticating |invalid )` as well as `( preauth)`. We will now create another capture group by adding parentheses around the matching for a username: `(.*)`. Now, we can refer back to this part of the text. Given this is the second capture group, we will use `\2` to refer to this capture group.

We can no use this in order to build up our `sed` command. Given we want to only extract the username, we will match the entire line and then in the substitution portion, we will simply replace with just the username.

```
ssh adrazen@192.9.152.85 journalctl
    | grep sshd
    | grep "Disconnected from"
    | sed -E 's/.*Disconnected from (invalid |authenticating )?user
     (.*) \[0-9.\]+ port [0-9]+( \[preauth\])?$/\2/'
```

5.3 RegEx Applications

Regular expressions are especially useful given they can be embedded inside of other tools. We've already seen how a regex expression can be used for substitution in `sed`. Regular expressions can also be used inside of applications such as Excel and Google Sheets that support data processing.

6 Useful Commands

6.1 `sort` and `uniq`

Now that we have extracted just the usernames, we can run some analysis on our data. There are a number of useful commands that will help us run this analysis. We can use the `sort` command to sort the usernames alphabetically. (More generally, `sort` is a command that can be used for sorting alphabetically or numerically).

We can also use `uniq` to find unique usernames within our data. If we call `uniq` with the `-c` flag, then we can get the number of occurrences for each username. `uniq -c` will collapse consecutive lines that are the same into a single line, prefixed with a count of the number of occurrences.

If we want to find which usernames appear most often, we can call `sort` again after we have collapsed usernames with their counts. This time, we will call `sort` with the `-n` flag in order to get a numeric sorting. Our final pipeline will look as follows:

```
ssh adrazen@192.9.152.85 journalctl
    | grep sshd
    | grep "Disconnected from"
    | sed -E 's/.*Disconnected from (invalid |authenticating )?user
     (.*) \[0-9.\]+ port [0-9]+( \[preauth\])?$/\2/'
    | sort
    | uniq -c
    | sort -n
```

6.2 `head` and `tail`

If we want to get the usernames of the ten users that were disconnected most often from our honeypot, then we can use the `tail` command with the `-n10` flag. The `tail` command prints the last `X` lines of a file. Given that the file is sorted with ascending counts, we will use the `tail` command.

If we want to get the usernames of the ten users that were disconnected *least* often from our honeypot, then we can use the `head` command with the `-n10` flag. The `head` command prints the first `X` lines of a file.

6.3 `xargs`

The final useful command that is worth knowing about is `xargs`. `xargs` is a command that allows you to use the output of one command as the arguments to another command. Note that this is different from using the output of one command as the input to another command.

To understand how to use `xargs`, we can look at an example. Imagine we have a file named `filenames.txt` which contains a bunch of file names. If we run `cat` on `filenames.txt`, we can see the contents of `filenames.txt` in the Terminal.

```
adrazen@ayelet-computer ~ % cat file_names.txt
homework.txt
program.py
todo-list.txt
random.txt
```

Now let's imagine that we want to create a new file for each filename in `filenames.txt`. One way to do this would be to run `cat filenames.txt` and to manually create new files using the `touch` command. This would work, but would be horribly inefficient.

Let's see how we might use the `xargs` command to streamline this process. Remember that `xargs` takes the *output* from a first command and passes them as the *arguments* to the second command. In our case, the output of `cat filenames.txt` is the names of the files we are interested in creating using `touch`. Using `xargs`, we can pass these file names from the output of `cat` to be the arguments to `touch`:
`cat filenames.txt | xargs touch`

7 Other Tools

This lecture covers just a subset of the tools that you may want to use for data wrangling and analysis. Here are a few other tools that you may be interested in using:

`awk` is a scripting language for manipulating data and generating reports.

`R` is another programming language that is great at data analysis and plotting.

`perl` is a programming language for text manipulation

CS45, Lecture 3: Shell Scripting

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Winter 2023

Contents

1 Lecture Overview

In Lecture 3, we learned how to use shell commands and pipelines to manipulate and analyze data. We also learned how to write regular expressions and how to incorporate these into tools such as `sed`. Finally, we learned how to run complex shell commands such as `grep`, `sort`, `uniq`, and `xargs`. In today's lecture we will learn how to write shell scripts and the syntax of shell scripts.

2 What is Shell Scripting?

We've already seen how to execute simple commands in the shell and pipe multiple commands together. Sometimes, we want to run many commands together and make use of control flow expressions such as conditionals and loops. This is where shell scripting comes in.

A **shell script** is a text file that contains a sequence of commands for a Unix-based operating system. It is called a script because it combines a sequence of commands-that would otherwise have to be typed into a keyboard one at a time-into a single script.

Most shells have their own scripting language, each with variables, control flow, and its own syntax. (You may have heard of bash scripting as a popular scripting language. Bash scripting is a type of shell scripting.) What makes shell scripting different from other scripting languages is that it is optimized for performing shell-related tasks. Creating command pipelines, saving results into files, and reading from standard input are primitives in shell scripting, making it easier to use compared to other scripting languages. (For example, if you want to run the `cd` command in Python, you would need to import the `os` library and then call `chdir` from that library.)

3 Bash Scripting: Basic Mechanics

Bash scripting refers to writing a script for a bash shell (Bourne Again SHell). You can check what shell you are using by running `ps -p $$`. If you are on Linux, your default shell should be a bash shell. If you are on macOS or Windows, you may need to switch to a bash shell. On macOS run `exec bash` to launch a bash shell. On Windows, run `bash` to launch a bash shell.

3.1 Your Very First Script

Now that we have a bash shell, we can write our very first bash script, called `hello.sh`. Shell scripts normally end with the `.sh` ending to indicate that they are scripts. To run a script, you will type the following at your command line: `sh hello.sh`

This script will make use of the `echo` command that we learned about in Lecture 2.

```
#!/usr/bin/env bash
echo "Hello world!"
```

3.2 Shebangs

The first line in our script (`#!/usr/env/ bash`) is called a shebang, or sharp exclamation. It the combination of the pound symbol (#) and an exclamation mark (!). The shebang is used to specify the interpreter that the given script will be run with. In our case, we indicate that we want our script to be run with a bash interpreter (i.e. a bash shell). If you want to run your script with a zsh shell, you would change your shebang to reflect as such.

There are a number of different ways to write your shebang such as `#!/usr/bin/env bash` and `#!/bin/bash`. We recommend that you always use the former as it increases the portability of your script. The `env` command tells the system to resolve the bash command wherever it lives in the system, as opposed to just looking inside of `/bin`.

Another example of a shebang that may be useful to many of you is using a shebang to write a python script. You may be familiar with writing a Python script and then running the script by calling `python3`. Yous can also create a Python script and specify that it should be run with Python using the shebang.

3.3 Running a Script

You can always run a shell script by simply prepending it with a shell interpreter program such as `sh hello.sh`, `bash hello.sh`, or `zsh hello.sh`.

You can also run a script by turning it into an executable program and then running it. First, you need to turn the program into an executable using the `chmod` (change mode) command. This command is used for changing the file permissions on a file, such as making the file executable. In our case, we will `chmod` with the `+x` argument to turn the script into an executable. You can do so as follows:
`chmod +x hello.sh`

To run the script, you would then simply run the program with `./hello.sh` (which is likely similar to how you have run other programs in the past.

4 Bash Scripting: Variables and Strings

4.1 Variables

Now that we have a basic script, let's talk about the mechanics of bash scripting. When writing a bash script, you can assign variables using the syntax `x=foo`. You can then access this variable using the syntax `$x`. One thing to be careful of is that when you assign a variable in a bash script, you should not add extra spaces. If you write `x = foo` then the line will be interpreted as running a program called `x` with the arguments `=` and `foo`. In general, shell scripts interpret the space character as an argument splitter.

4.2 Strings

We can also define strings in a bash script. If we want to define a string literal, we will use single quotation marks: `'$x'`. If we want to define a string that allows substitution, we will use double quotes: `"$x"`. The double quotes will allow for substitution of variables:

```
x=foo
echo '$x'
# prints $x
echo "$x"
# prints foo
```

5 Bash Scripting: Control Flow Directives

Like other programming languages, bash scripts also have control flow directives such as `if`, `for`, `while`, and `case`.

5.1 If Statements

The syntax for writing an if statement in bash is as follows:

```
#!/usr/bin/env bash

if [ CONDITION ]
then
  # do something
fi
```

The condition we are interested in is denoted by `CONDITION`.

Let's take a look at an example of a bash script with an if statement:

```
#!/usr/bin/env bash

num=101
if [ $num -gt 100 ]
then
  echo "That's a big number!"
fi
```

In this script, we begin by assigning a variable `num` to be equal to 101 on line 3. We then check whether `num` is greater than 100. Notice that we use the syntax `$num` to access the value of `num`. We use the comparison operator `-gt` to compare `num` to `100`. In bash, comparisons for integers and strings are done differently. `-gt` is an integer comparison operator while `>` is a string comparison operator (that compares ASCII values). The use of the square brackets is actually a synonym for the `test` command, which tests the validity of a command or statement.

We can also write an if-statement with multiple conditions. In this example, we will check if `num` is both greater than `100` and less than `1000`.

```
1  #!/usr/bin/env bash
2
3  num=101
4  if [ $num -gt 100 ] && [ $num -lt 1000 ]
5  then
6    echo "That's a big (but not a too big) number!"
7  fi
```

Again, in this script we assign `num` to be 101. We first check if `num` is greater than `100`. We than add a second condition using the `&&` syntax. Our second condition checks if `num` is less than 1000.

We can also use `elif` (for else if) and `else` if we have multiple different blocks of code. Here is the syntax for using `if`, `elif`, and `else`:

```
1  #!/usr/bin/env bash
2
3  if [ CONDITION ]
4  then
5    # do something
6  elif [ CONDITION ]
7  then
8  # do something else
9  else
10 # do something totally different
11 fi
```

The above syntax should look fairly similar to the traditional `if` statement. Here is an example of code that uses `if`, `elif`, and `else`:

```
1  #!/usr/bin/env bash
2
3  num=101
4  if [ $num -gt 1000 ]
5  then
6    echo "That's a huge number!"
7  elif [ $num -gt 100 ]
8  then
9    echo "That's a big number!"
10 else
11   echo "That's a small number."
12 fi
```

5.2 While Loops

We can also add while loops to our bash scripts. The syntax for adding a while loop to a bash script is the following:

```
1  #!/usr/bin/env bash
2
3  while [ CONDITION ]
4  do
5    # do something
6  done
```

Again, the condition of interest is denoted as `CONDITION`.

Here is an example of a script that initializes `num` to `0` and continues looping until `num` reaches a value of `99`.

```
#!/usr/bin/env bash

num=0
while [ $num -lt 100 ]
do
  echo $num
  num=$((num+1))
done
```

Notice how we access the variable `num` with the `$` syntax: `$num`. We use the `-lt` flag to compare `num` to `100`. When then print the value of `num` using the `echo` command. To increment the value of `num`, we use the double parentheses `(( .. ))` for arithmetic evaluation. Inside of the double parentheses, we can increment the value of `num` by 1.

5.3 For Loops

To declare a for loop in bash, we can declare either an index-based for loop or a range-based for-loop. To declare an index based for-loop, we will use the following syntax:

```
#!/usr/bin/env bash

for VARIABLE in 1 2 3 .. N
do
  # do something
done
```

Perhaps, we are interested in implementing the above `while` loop as `for` loop:

```
#!/usr/bin/env bash

num=0
for i in {0..99}
do
  echo $num
  num=$((num+1))
done
```

Notice that we define our iterator `i` and we set the bounds of the for-loop to be 0 and 99.

5.4 Exercise 1

Let's put our scripting expertise to use and write a bash script. You should write a script called `num_loop.sh` that loops through every number `1` through `20` and prints each number to standard output. The script should also conditionally print `I'm big!` for every number larger than `10`.

Solution: Here is one possible solution : `num=$1`:

```
#!/usr/bin/env bash

for i in {1..20}
do
  echo $i
  if [ $i -gt 10 ]
  then
    echo "I'm big!"
  fi
done
```

6 Bash Scripting: Arguments and Functions

6.1 Arguments

Our `big_num.sh` script isn't very interesting as it will always print the same thing: "That's a big number!" Let's take a look at how we might use command line arguments to make this script a little more interesting.

In bash, the variables `$1` - `$9` refers to the arguments to a script. The variable `$0` refers to the name of the script. For example, consider the following:
`adrazen@thinking-computer CS45 % sh my_script.sh ayelet`
In this case, the name of the script (`my_script.sh`) is defined by the variable `$0`. The first argument to the script (`ayelet`) is defined by the variable `$1`. In the case of our `big_number.sh` script, we can change the value of `num` to be dependent on the first argument that is passed in when the script is invoked. In other words, `num` will simply be the value of `$1`.

Let's assume that we invoke the script as follows:
`adrazen@ayelet-computer CS45 % sh big_num.sh 102`

In our script, we can replace the line `num=101` to be `num=$1`:

```
1 #!/usr/bin/env bash
2
3 num=$1
4 if [ $num -gt 100 ]
5 then
6   echo "That's a big number!"
7 fi
```

By changing line 3, we now set the value of `num` to be the first argument from the command line when invoking the script. This means the use could pass a different value every time they invoke the script.

6.2 Functions

We can also define functions in bash. Let's define a function for making a new directory and then entering that directory. (This is a pretty commaon thing that users want to do and there is actually a command called `mcd` in UNIX to allow users to do that.) We can use the commands `mkdir` and `cd` to achieve this. When running `mkdir`, we want to make sure that we also create any necessary intermediate directories. Therefore, we will want to make sure we pass the `-p` flag when calling `mkdir`, which creates all the intermediate directories on the path to the final directory that do not already exist.

When defining a function in bash, you may wonder how to pass arguments to the function. Let's take a look at our function so far and how you might think of passing arguments:

```
1 #!/usr/bin/env bash
2
3 make_and_enter(directory_name) {
4   mkdir -p directory_name
5   cd directory_name
6 }
```

Unfortunately, this won't work in bash. In bash, we will refer to arguments that are passed into a function based on their position. For instance, we might use the value of the very first argument and refer to it using `$1`.

```
1 #!/usr/bin/env bash
2
3 make_and_enter() {
4   mkdir -p "$1"
5   cd "$1"
6 }
```

When we want to invoke our function, we will use the following line:
`make_and_enter new_folder`

In this case, we are calling our function (i.e. `make_and_enter`) with the argument `new_folder`. Our script will look as follows:

```
#!/usr/bin/env bash

make_and_enter() {
  mkdir -p "$1"
  cd "$1"
}

make_and_enter new_folder
```

Now consider that we want the name of the new folder to be passed from the command line whenever the script is invoked. In that case, we would invoke the script as follows: `adrazen@ayelet-computer CS45 % sh mcd.sh my_folder` In order for the argument `new_folder` to be used inside of our function, we need to make sure that we pass it into the function. In this case, we need to pass the argument from the command line invocation to the function call, and then from the function call to the function body. Our final script would look as follows:

```
#!/usr/bin/env bash

make_and_enter() {
  mkdir -p "$1"
  cd "$1"
}

make_and_enter "$1"
```

6.3 Exercise 2

Let's try another exercise to solidify our function-writing and argument-passing skills. In this exercise, you should write a shell script called `my_folder.sh` that takes in two arguments: your name (e.g. `ayelet`) and your name with the `.txt` ending (e.g. `ayelet.txt`). The script should call a function that creates a folder by the name of the first argument (e.g. `ayelet`) and then create a file inside by the name of the second argument (e.g. `ayelet.txt`). For my name, my function would create a folder named `ayelet` and a file named `ayelet.txt` inside of `ayelet`.

Solution:

Here is possible solution to this problem:

```
#!/usr/bin/env bash

make_my_folder() {
  mkdir "$1"
  cd "$1"
  touch "$2"
}

make_my_folder $1 $2
```

7 Exit Codes and Command Substitution

7.1 Exit Codes

The notion of exit codes allows for verifying the success or failure of a previous command. An **exit code** or **return value** is the way scripts or commands can communicate with each other about how execution went. A return value of 0 means that everything went OK. A return value other than 0 means that an error

occurred. `$?` provides the return value from the previous command. (For students who are familiar with C/C++, you may notice that the `main()` function always returns an `int` and that this `int` is often 0. The `int` returned by `main()` is the exit code for the program.)

If you ever need a placeholder for a command that succeeds or fails, you can use the true and false commands. `true` is a command that does nothing except return an exit status of 0. `false` is a command that does nothing except return an exit status of 1.

Below is an example of a script that randomly generates either 0 or 1 and runs a either the `true` or the `false` command based on this random value. The script then checks the return value of the previous command. As you can tell, this script is a bit contrived but it demonstrates how you might use the return value of the previous command as an input to the next command.

```
#!/usr/bin/env bash

result=$(($RANDOM % 2))
if [ $result -eq 0 ]
then
  true
  echo "$?"
else
  false
  echo "$?"
fi
```

Return values are useful if you want to conditionally execute commands based on the execution of the previous command. In addition to using if-statements, we can also conditionally execute commands using `&&` and `||`.

7.2 Exercise 3

Exercise 3: Write a shell script called `file_checker.sh` that checks if a file exists or not. The script take in a file name as an argument and try to run `cat` on that file. The script should then check the exit code of the `cat` command to determine if the file exists or not. If the file exists, the script should print `File exists!`. If the file does not exist, the script should print `File does not exist!`.

Bonus: change the script to suppress the actual output of `cat` and only include your script's output (e.g. `File exists!` or `File does not exist!`).

Solution:

Below is one possible solution:

```
#!/usr/bin/env bash

cat $1
if [ $? -eq 0 ]
then
  echo "File exists!"
else
  echo "File does not exist!"
fi
```

To suppress the output of `cat`, you should modify the script as follows:

```
#!/usr/bin/env bash

cat $1 &> /dev/null
if [ $? -eq 0 ]
then
  echo "File exists!"
else
  echo "File does not exist!"
fi
```

7.3 Command Substitution

Command substitution is another useful feature of bash scripting. You might want to run a command and then use its output as a variable to some other piece of code.

Here is an example of a script that uses command substitution:

```
#!/usr/bin/env bash

for element in $(ls ~/Desktop)
  do
    echo "Desktop contains file named $element"
  done
```

8 Bash Scripting: Other Syntax

There is plenty of other syntax to keep in mind when it comes to bash scripting. Here are a few other syntax details for bash.

8.1 Comparisons

Bash draws a distinction between comparisons for numbers and comparisons for strings. In order to compare numbers in a bash script, use the following:

- `a -eq b` for checking if a is equal to b
- `a -ne b` for checking if a is not equal to b
- `a -gt b` for checking if a is greater than b
- `a -ge b` for checking if a is greater than or equal to b
- `a -lt b` for checking if a is less than b
- `a -le b` or checking if a is less than or equal to b

In order to compare strings in a bash script, use the following:

- `s1 = s2` for checking if s1 is equal to s2
- `s1 != s2` for checking if s1 is not equal to s2
- `s1 < s2` for checking if s1 is less than s2 by lexicographical order
- `s1 > s1` for checking if s1 is greater than s2 by lexicographical order
- `-n s1` for checking if s1 has a length greater than 0
- `-z s1` for checking if s1 has a length of 0

8.2 Exercise 4

Exercise 4: Write a shell script called `timely_greeting.sh` that greets you based on the current time. The script should call the date command, extract the current hour (look into using %H) and then print the following greeting based on the time.

- If it is between 5AM (05:00) and 12PM (12:00): Good morning!
- If it is between 12PM (12:00) and 6PM (18:00): Good afternoon!
- If it is between 6PM (18:00) and 5AM (5:00): Good night!

Solution: Here is one possible solution:

```
#!/usr/bin/env bash

time=$(date +%H)
if [ $time -gt 5 ] && [ $time -lt 12 ]
then
  echo "Good morning!"
elif [ $time -gt 12 ] && [ $time -lt 18 ]
then
  echo "Good evening!"
elif [ $time -gt 18 ] && [ $time -lt 5 ]
then
  echo "Good night!"
fi
```

CS 45, Lecture 5

Text Editors

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Spring 2023

Outline

Contents

1 Text Editing: An Overview

1.1 Rich Text

When we think about editing a document, we usually think of doing that in a *rich text editor*, something like Word or Google Docs.

Rich text is for *humans communicating with humans*– its elements are structured around elements of prose, such as words, paragraphs, headings, etc., and its features are centered around making consuming written text easier for humans– things like varying fonts, emphasizing text with bold, italic, or underline, the ability to insert pictures or other multimedia, and so on and so forth.

However, while this information is helpful and sometimes really necessary in human-to-human communication, it's unnecessary and gets in the way when we're desiring to communicate with a computer (or give it instructions). This is why we use *plain text* for computers!

1.2 Plain Text Editors

Lots of different kinds of programs have been developed to edit plain text– in fact, it's really one of the core affordances a computer offers. Some plain text editors, like Windows' Notepad or macOS' TextEdit are extremely basic, and fulfill the mantle of a plain text editor with no frills. However, knowing how we frequently use plain text to communicate, configure, and program computers, many more plain text editors integrate additional tools to make these tasks easier.

Computer programs in general tend to fall into one of three categories:

1. *GUI (Graphical User Interface) programs.* These are programs that display a graphical interface in some way, and are the kinds of applications you're almost certainly most familiar with.
2. *TUI (Text User Interface) programs.* These are programs that display a sort of imitation of a graphical interface using text within a terminal. These are distinct from CLI programs in that the application stays open and allows for continued operation, taking over the terminal and designed for interaction directly with the user via the keyboard and sometimes mouse.
3. *CLI (Command Line Interface) programs.* These are programs that are usable only from the command line, by invoking the program with a set of flags and arguments, and potentially information from standard input.

As a caveat, TUI applications especially are often fairly inaccessible to screen readers; since they just display information– including UI– as a bunch of text that isn't distinguished or hierarchical in any way.

1.3 Learning a new editor

No matter what kind of editor you choose to jump into learning, there's going to be a learning curve. Our recommendation is to *choose one GUI editor and one TUI editor*, then stick with them for a while. We'd estimate that you'll reach the same speed as you'd use any other editor after about 10-20 hours of use, and often will be much faster than others after about 20 hours of use.

Don't be afraid to look some stuff up, too– often, there's a faster way to go about doing things that's just a google search away!

Since we expect that you all are pretty familiar with GUI editors (such as PyCharm), we'll jump into doing a short demo with `vim`.

1.4 Why `vim`? Or a TUI editor at all?

The biggest reason is for *remote editing.* Computers whose purpose is to serve content or provide resources– i.e. servers– often do not have GUIs installed at all, as they are a significant resource use overhead that has no good purpose when CLI and TUI applications exist that don't require all that overhead. Thus, it's a good idea to get used to some editor you can use via only a terminal.

2 Vim

2.1 A Quick History

`vim` was inspired by and spun off of an editor called `vi`, and stands for VI iMitation (or VI iMproved, depending on who you ask). `vi` was one of the first TUI editors, based on the line-editor `ed` (and also the `vi`sual mode of a CLI tool called `ex`), which required you to edit line by line using certain commands.

`vi`, and `vim`, continue to use that idea of *commands and modes*!

2.2 A Modal Editor

`vim` *uses different "modes" to control editing.*

1. You always start in *normal mode*, used for navigating around the file
2. You press `i` to enter *insert mode*, to write new text
3. You press `R` to enter *replace mode*, to overwrite text
4. You press `v` to enter *visual mode*, for copying or deleting lines or blocks of text at a time
5. You press `:` to enter *command mode*, which allows you to do all sorts of things (like save, quit, find-replace, etc.).

2.3 Learning to Navigate `vim`

Example 2.1 (codeblock). We did a demo in class. You can access the demo using the commands below:

(Curious what these commands do? We introduced a handy-dandy website called explainshell.com that can break down commands for you using text from the `man` pages. In the case of the commands above, the first one (`curl`) downloads a file, and the second instructs vim to open the file that was downloaded.)

You should open the file in `vim` and try navigating through it; we did this exercise for 15 or so minutes live in class.

2.4 Windows & Buffers

We're all used to the idea that each window is responsible for one file. If you want to open a new file in another window, you can do that.

`vim` plays with this idea a little bit. *Buffers* refer to open files, and are an abstract concept. A single buffer may be open in one *or more* windows!

Meanwhile, *windows* are "views" into a buffer. This means you could have *multiple windows open to the SAME buffer!*– if you do this, that means your changes to the buffer in one window would instantly reflect in the other. This is useful if e.g. you want to scroll to multiple different parts of a file at once.

`:q` always closes the current window. You can also "split" a window using the `:sp` ("split") command, or vertically with the `:vsp` command.

2.5 Configuring `vim`

You can customize your installation of `vim` by writing a `.vimrc` file in your home directory (i.e. `/.vimrc`).

`.vimrc` files contain a list of commands that run when you start `vim`. For example, mine makes the mouse work, adds line numbers, and makes backspace and the arrow keys work in a manner I'd expect from an editor.

You can also add 3rd-party plugins to `vim`, either manually or using a plugin manager like `vundle`.

2.6 Demoing `.vimrc`

Example 2.2 (codeblock). We did a demo in class. You can access the demo using the commands below:

(The above command will open a temporary copy of the file at that URL, without saving it permanently.)

The above `.vimrc` is a slightly simpler version of the one I use, commented so you can see what each part does. It is based on a `.vimrc` file that was distributed as part of CS107 back when they used vim.

3 Visual Studio Code

Visual Studio Code is our IDE of choice, as it stands right now!

3.1 What's an IDE?

An IDE, or *Integrated Developer Environment*, is an application for software development and software code editing that bundles together *lots of functionality for developer productivity into one place.*

In particular, this usually means bundling code editing tools together with syntax highlighting and smart autocomplete, as well as error checking, build tools, testing tools, the ability to run code, and some other tools that scan and index your code automatically to help you understand and navigate it quickly, all in one tool.

3.2 Why VSCode?

We like VSCode for a couple reasons; besides the fact that it is totally free, it also has *plug-and-play language support* (you can make it richly support new languages by just installing a plugin to it)– and there are a *lot* of available language plugins that are very good for VSCode.

Another key reason is that VSCode offers strong support for *remote editing*, which means that you can access and edit resources *on a server*, without needing a GUI shell to be installed on the server at all.

3.3 VSCode Demo

We did a demo in class, demonstrating some of the features of an IDE, the ability to install plugins for language support, and showing off remote editing.

CS 45, Lecture 6

Command Line Environment

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Spring 2023

Outline

Contents

Announcements

- Assignment 1 is due today. Reach out if you don't think you will be able to get it done in time.
- Assignment 2 is out! It's due a week from today on Wednesday, April 26th at 11:59 PM.

1 Review

Recap

In the previous lecture, we saw:

- How to edit files in the terminal
- How to enter/exit a full screen program (`vim`)

In this lecture, we will see:

- How to configure and customize your shell
- How to multitask in the terminal
- How to run multiple programs side-by-side

Before we get into that, there are a few similar terms that are often confused; let's get into what they mean and how they're different.

Terminal vs. Shell vs. Command Line

Definition 1.1 (terminal)**.** The TERMINAL is the window you open. Think of it like a web browser.

Definition 1.2 (shell)**.** The SHELL is the program you use to launch other programs. Think of it like Google.

Definition 1.3 (cli)**.** A COMMAND LINE INTERFACE (CLI) is a generic term for a text-based program which runs within a terminal. Think of this like "the web". A CLI PROGRAM or a TUI PROGRAM is like a website.

You may often see a few acronyms in the context of user interfaces:

CLI: Command-Line Interface (like the shell)

TUI: Text User Interface (like `vim`)

GUI: Graphical User Interface (like Microsoft Word)

2 The Environment

We've already seen how you can control the behavior of CLI programs using flags, but there's another way to configure them. Each program runs in what's called an "environment".

Contents of the Environment

The "environment" a program runs in includes several things:

- The user who's running it
- The files on the filesystem
- Environment variables (configuration variables)
- `stdin` and `stdout` (and `stderr`)

Each of these affects running programs in different ways. Some of them control what a program can or cannot do, others control the default behavior of programs.

2.1 Configuration

Input/Output

We already saw this in Lecture 2, but we can control the default input and output files of a program using REDIRECTION, i.e., the `<`, `>`, `>>`, and `|` operators.

By default, input comes from the terminal (`/dev/tty*` or `/dev/pts/*`); you can see the name of the "controlling terminal" of a program by running `tty`.

A CONTROLLING TERMINAL is essentially which terminal window a program is running in.

Input and output can be redirected, but a program is bound to a specific window. When that window is closed, the program will exit.

The command `tty` stands for "teletypewriter". A teletypewriter was, essentially, a typewriter than was plugged into a computer and used for input/output. As we talked about in Lecture 2, these got replaced later with "video terminals" (which had a basic screen instead of a piece of paper), which were themselves later replaced with terminal emulators (the program called "Terminal" on your computer). Nowadays, terminal emulators tell your OS to create what's called a "pseudo-terminal" (or "pty"), which lets them pretend to be a terminal even though they're just normal programs.

Environment Variables

ENVIRONMENT VARIABLES are a way to configure a program's default behavior.

We've already seen shell scripting variables, environment variables are basically the same thing except they're "exported" so other programs can use them.

For example, the `$PATH` variable determines where programs can be located. If a program isn't found "on your `$PATH`", you'll get a "command not found" error.

Other common variables:

`$TERM`: Which terminal you're using.

`$USER`: Your username

`$EDITOR`: Which editor you prefer

`$PWD`: Your current directory

PATH

My `$PATH` looks like this:

```
/home/akshay/.local/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:
/var/lib/flatpak/exports/bin:/usr/bin/site_perl:
/usr/bin/vendor_perl:/usr/bin/core_perl
```

This is a list of directories, where each directory is separated by colons (`:`).

When you run a program like `grep`, the shell looks in each directory on your `$PATH` from left to right.

Setting Environment Variables

You can "export" a shell variable to turn it into an environment variable as follows:

```
export MYVAR="hi"
python -c 'import os; print(os.getenv("MYVAR"))'
```

Note that even though I ran a Python program, not a shell program, it was able to read the value of `$MYVAR`.

You can temporarily set an environment variable as follows:

```
MYVAR=hi python -c 'import os; print(os.getenv("MYVAR"))'
```

In this case, only that one line gets run with `$MYVAR` set to "hi"; after that, `$MYVAR` goes back to its old value (or becomes undefined).

Environment variables are "inherited"—child programs (and their descendants) will be able to see their value, but **not** any other programs.

Environment variables are ubiquitous in UNIX, and pretty much every programming language has a way to read and write them. The shell makes it incredibly easy by treating them as normal variables; in other languages, you'd have to call a function to get or set them.

By default, environment variables only persist until you exit the shell (or close the terminal window). We'll see later how to make them persist.

2.2 Permissions

Users and Groups

We also talked about this a bit in Lecture 2, but every command you run runs as a specific user.

The variable `$USER` conventionally holds your username (although this isn't guaranteed); you can also run `whoami` to see who is logged in.

Every user may belong to one or more "groups", which you can see by running `groups`.

For example, I'm in the groups:

```
 % groups
docker uucp audio wheel akshay
```

Each of these groups gives me special permissions and abilities. For example:

- The group "docker" means I can run the program "Docker" (which usually only root can run) without using `sudo`.
- The group "uucp" means I can read and write to serial ports (the name is a holdover from the old "unix-to-unix copy" program).
- The group "audio" means I can use my computer's speakers (you wouldn't have these permissions on a remote server like `myth`, for example).
- The group "wheel" gives me permission to use the program `sudo`. Trying to use `sudo` without this permission will cause an error.
- The group "akshay" gives me permission to… access my own files? This is kind of useless nowadays (I can *always* access my own files, since I own them), but it used to be used as a way to share files among different users on the same computer.

Historically, groups were used to share files among different users on a multi-user system (like *myth*). For example, Professor Engler's research group might have had a corresponding UNIX group named `engler_group`, so we could all collaborate on files and source code. Nowadays this has been replaced with other means of collaboration like `git`, and everyone has their own laptop, so this use has become more rare.

Permissions

On UNIX, you must have the appropriate "permissions" to do certain actions.

Source: xkcd 838

One of the most important types of permissions (and probably the only one you'll see frequently) is permissions on specific files.

File Permissions

Every file has an "owner" and a "group".

Every file has three sets of permissions: owner permissions, group permissions, and everyone else permissions.

`r` means "permission to read", `w` means "permission to write", and `x` means "permission to execute (i.e., run)".

You can see file permissions by running `ls -l`.

File Permissions Example

Output of `ls`

```
-rwxr-xr-x 1 root root      153736 Sep  4 07:33 grep
```

These are the permissions on my `/usr/bin/grep` binary, as given by `ls -l`.

File Permissions Example

Owner

```
-rwxr-xr-x 1 root root      153736 Sep  4 07:33 grep
```

The owner (root) can read, write, and execute `/usr/bin/grep`.

File Permissions Example

Group

```
-rwxr-xr-x 1 root root      153736 Sep  4 07:33 grep
```

The members of the group "root" can read and execute `/usr/bin/grep`, but **not** write to it.

File Permissions Example

Everyone

```
-rwxr-xr-x 1 root root      153736 Sep  4 07:33 grep
```

Everyone else can read and execute `/usr/bin/grep`, but **not** write to it.

Changing Permissions

Owner

We can change the owner or group of a file using the `chown` and `chgrp` commands.

Example 2.1 (chown). Changing the owner of a file `hello.txt` to the user `akshay`:

```
chown akshay hello.txt
```

Changing Permissions

Group

We can change the owner or group of a file using the `chown` and `chgrp` commands.

Example 2.2 (chgrp). Changing the group of a file `hello.txt` to the group `staff`:

```
chgrp staff hello.txt
```

Changing Permissions

We can change the permissions on a file using the `chmod` command (CHANGE FILE MODE).

A file's MODE is the combination of its read, write, and execute permissions, for example `rw` for readable/writable or `rx` for readable/executable. See the `chmod` man page for more info!

We've already seen this!

Example 2.3. <only@2>[chmod +x] Make a shell script executable:

```
chmod +x my_script.sh
```

Example 2.4. <only@3>[chmod -w] Make a file read-only.

```
chmod -w my_safe_file.txt
```

Example 2.5. <only@4>[chmod -r] Make a file non-readable:

```
chmod -r my_secret.txt
```

By default, `chmod` changes the permissions for everyone at once. You can also specifically change one of the three sets of permissions:

```
chmod u+x my_script.sh
chmod g+rw group_plan.txt
chmod o-r my_secret.txt
chmod 777 open_permissions.txt
```

In the last example, I set the permission bits of the file directly. They are represented as an octal number; each of the digits represents three bits (read, write, and execute respectively). There are three digits for the three kinds of permissions: owner, group, and everyone. For example, a file that's 644 is readable and writable by the owner, but only readable by everyone else.

Note that, if you are the owner of a file, you can easily override the permissions. These permissions are mostly to protect your files from *others*, not from yourself.

Types of File

There are a few types of files, with different properties. You can tell them apart by the first character in the output of `ls -l`.

```
lrwxrwxrwx 1 root root       21 Oct  8 16:05 os-release -> ../usr/lib/os-release
drwxr-xr-x 1 root root       18 Oct  8 16:15 ostree
-rw-r--r-- 1 root root       79 Nov 29 02:14 ostree-mkinitcpio.conf
```

This is from my `/etc` directory, which is where programs store their configuration files.

Types of File

- A regular file.

b A block device (like a hard disk).

c A character device (like a serial port).

d A directory.

l A symbolic link.

n A network file.

p A "named pipe".

s A "named socket".

Of these, the ones you'll see most are regular files, directories, and symbolic links. We haven't seen symbolic links before, so let's look at them in a little more detail.

2.3 Shortcuts

Symbolic Links

Definition 2.6 (symlink)**.** A SYMBOLIC LINK (or "symlink") is a shortcut to a file or directory.

You can create one with the `ln -s` command, as follows:

```
ln -s $target $link_name
```

When you try to read from a symlink, you actually read from the file it's pointing to. The `readlink` command tells you where a symlink points.

They're useful in cases where you want the same file to exist in multiple places, but you don't want to have a bunch of copies (which can get out of sync if someone edits one but not the others). For example, I use them to have copies of files in both my "Dropbox" and "Documents" folders.

Permissions are *shared* between a symlink and the target file. Trying to change the permissions on the link will change the permissions on the file itself.

In addition to shortcuts to files, we can also define shortcuts to commands. These are called aliases.

Aliases

Definition 2.7 (alias)**.** A ALIAS is like a shortcut for a specific command.

You can create one with the `alias` command, as follows:

```
alias hi="echo 'hello'"
```

Running an alias will run the command it points to. You can see what an alias named "hi" does by running `alias hi`.

Just like environment variables, aliases only last until you exit the shell.

Aside: Searching for Files

The FIND tool (which has the unusually logical name `find`) is a powerful way to search for files. We're not going to go into too much detail, because it's ultimately not all that important, but here are a few examples:

Example 2.8. <only@2>[find -name] Find all files named "hello":

```
find . -name "hello"
```

Example 2.9. <only@3>[find -executable] Find all files marked "executable":

```
find . -executable
```

Example 2.10. <only@4>[find -type] Find all regular files, directories, and links:

```
find . -type f,d,l
```

Example 2.11. <only@5>[find] Find all regular files (but not links) which are marked executable and named "hello".

```
find . -type f -name "hello" -executable
```

Example 2.12. <only@3>[find -type] Find all regular files, directories, and links:

```
find . -type f,d,l
```

If you want to learn more about `find`, look at the man page! There are a lot of examples in there.

3 Shell Configuration

Just like you can configure `vim` using `.vimrc`, you can configure your shell using a special "run commands" file.

Configuring your Shell

If you're using `bash`, your shell configuration file is called `~/.bashrc`. If you're using `zsh`, it's called `~/.zshrc`.

This file is a shell script that's run every time your shell starts. You can use it to define aliases and environment variables.

For example, my `.bashrc` includes the lines:

```
alias ls='ls --color=auto'
PS1='[\u@\h \W]\$ '
export EDITOR=vim
export PATH=$PATH:~/bin
```

The first line defines an alias for the command `ls`, which automatically runs it in "color" mode.

The second line configures my prompt, using the prompt variable `$PS1`.[1]

The third line sets my default editor to be `vim`. A lot of programs use this, so set it to your favorite terminal editor! You can also set it to VSCode using `export EDITOR='code --wait'`.

The fourth line adds the directory `~/bin` to my `$PATH`. This means any executable files within `~/bin` (for example, a shell script) can be run as ordinary programs from anywhere.

Note that `$PS1` is a shell variable (not exported), while `$EDITOR` and `$PATH` are environment variables (exported).

The shell (and most CLI programs) are incredibly customizable, far beyond what even a whole quarter could cover. If you're running a program and you wish it did something slightly different by default, there's probably a way to make that happen.

4 Multitasking

We know how to run one command at a time right now, but if we want to run two commands at once we have to open a new terminal. Honestly, nowadays opening a new terminal window isn't too bad, but in the olden days when a "terminal" was a piece of hardware, it wasn't super feasible.

4.1 Job Control

Jobs

Definition 4.1 (job)**.** A JOB is a task you're doing in the terminal, usually corresponding to a program that you're running. You can have one FOREGROUND JOB and many BACKGROUND JOBS running at the same time. You can also have many SUSPENDED JOBS which are frozen (i.e., not running).

Whenever we run a program from the shell, we're starting a new foreground job. Jobs are tied to their "controlling terminal", and will exit when the terminal window is closed.

Suspending Jobs

You can "suspend" a job (put it to sleep) by pressing CONTROL-Z on your keyboard. Try it from `vim`!

You can see all the jobs in your current terminal and their statuses by running `jobs`.

[1] If you want to learn how to customize your own prompt, try `man bash` or `man zshmisc`.

Background Jobs

You can "background" a suspended job (wake it up, but hide it) by running `bg`.

If you try to background a program like `vim`, it'll immediately suspend itself again because it needs to be connected to a terminal. However, if you have a long-running command like a download, you can background it without any issues.

If you have multiple jobs suspended, `bg` will run the most recent one. You can specify a different one using the job number from `jobs`:

```
bg %1
```

Background Jobs

You can also run a new job in the background by adding an ampersand (`&`) to the end of the command:

```
sleep 5 &
```

You can also do this to a set of commands:

```
(sleep 5 && printf "\a") &
```

Foregrounding Jobs

You can "foreground" a suspended or background job (wake it up and let it take over the terminal) by running `fg`.

If you have multiple suspended or background jobs, `fg` will run the most recent one. You can specify a different one using the job number from `jobs`:

```
fg %1
```

Quitting Jobs

Usually, you can "kill" a foreground job (quit it) by pressing CONTROL-C on your keyboard.

You can "kill" a suspended or background job (wake it up and let it take over the terminal) by running `kill`. You must specify a job number from `jobs`:

```
kill %1
```

Note that it may take some time for the program to exit, and this may not work on certain programs like `vim`.

Force-quitting Jobs

The `kill` command works by sending the process (program) the `SIGTERM` signal (which politely asks it to exit).

`SIGTERM` is a bit like pressing the "close" button on a window. Most of the time it'll work, but sometimes the program will just not respond.

Some processes (programs) may ignore `SIGTERM`. In this case, you can use `SIGKILL` to force-quit it.

```
kill -s KILL %1
```

Or, equivalently:

```
kill -9 %1
```

Sending `SIGKILL` is the equivalent of ending a program from Task Manager (on Windows) or force-quitting a program (on macOS). Any unsaved work will be lost, and the program may exit in an invalid state (which might cause error messages the next time you open it). Don't do this unless you absolutely have to.

You might also want to know about the `killall` and `pkill` commands, which are similar to `kill` but take the *name* of a process rather than the process's ID.

4.2 Multiplexing

Splitting the Terminal

Sometimes we want to have multiple terminal programs open *at the same time*. In other words, we want to "split" our terminal window.

You might be wondering why you wouldn't just open a second terminal window. If you're working locally, that's actually a great option! It'll integrate with the rest of your non-terminal programs far better than anything else. However, a lot of the time you'll be working on a remote shell over `ssh`; in this case, opening a new terminal window is a hassle.

Job control will only let us open one program in the foreground at a time.

Unfortunately, there is *no built-in way* to have multiple programs open at the same time.

Fortunately, the shell is almost 60 years old, and other people have solved this problem for us.

Terminal Multiplexers

A TERMINAL MULTIPLEXER is a program which splits one "real" terminal (i.e., one window) into many "virtual" terminals.

There are a few terminal multiplexers around:

- `screen` is old but installed on most computers
- `tmux` is new but needs to be installed manually

For this class, we'll be talking about `tmux`!

Prefix Keys

We need some way to "talk" to `tmux` to give it commands.

But we also want to talk to the program running inside `tmux` so we can use it!

`tmux` solves this problem using a PREFIX KEY; any time you want to talk to `tmux`, you start by pressing CONTROL-B.

If you want to send a CONTROL-B to a program *inside* tmux, press CONTROL-B twice in a row.

Using `tmux`

If you run `tmux`, you're given a shell prompt with a status bar at the bottom.

There's a bunch of keyboard shortcuts to do various things in `tmux`. Remember to press CONTROL-B before using any of them!

Splitting the screen (vertically): `%`

Splitting the screen (horizontally): `"`

Going to the next "pane": `o`

Going to a specific pane: `q <number>`

Close the current pane `x`

Check out https://tmuxcheatsheet.com/ or https://quickref.me/tmux for more!

Advanced `tmux`

`tmux` has another use; you can "detach" from your virtual terminal and reattach to it from another terminal window.

To detach: CONTROL-B `d`

To attach: `tmux attach`

Why `tmux`?

Where `tmux` really shines is when used with `ssh`.

- You only need to enter your `ssh` password once.
- If your Wi-Fi drops and you lose your `ssh` connection, your programs keep running.
- You can detach a `tmux` session containing a long-running job and come back to check on it later.

CS 45, Lecture 7

Compilers

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Spring 2023

Learning Goals

In this lecture, we will see:

- How UNIX and C were designed together
- How C compilers work
- How package managers work
- How Python has taken over UNIX scripting

Outline

Contents

1 The C Language

C

- C is a general-purpose programming language from the 1970s.
- C is used everywhere, and has inspired:
  - Every major operating system
  - Every mainstream programming language
- Even if you never write C, you indirectly use it every day.

Before C

- Nowadays, C is considered a weird, old (and sometimes scary!) language.
- The languages before C were *even worse*.

This is the "hello, world!" program in B (the language before C):

```
main( ) {
    extern a, b, c;
    putchar(a); putchar(b); putchar(c); putchar('!*n');
}

a 'hell';
b 'o, w';
c 'orld';
```

In B, strings were limited to four characters. If you wanted to say something longer than four letters, you had to declare multiple variables. How anyone ever programmed anything useful in this language is a mystery to me. Amazingly, the first versions of UNIX were written in B; however, they quickly realized they needed a better language.

Early C

- C was designed in the 1970s by Dennis Ritchie, one of the authors of UNIX.
- It was officially published in the 1978 book *The C Programming Language* by Brian Kernighan and Dennis Ritchie, commonly called "K&R C".

Here's the "hello, world" program in K&R C:

```
main( ) {
        printf("hello, world");
}
```

Much better, right?

The C programming language was used to implement most of UNIX's kernel and userspace.

Problems with K&R C

Early C had some issues:

- The compiler just translated C into assembly language. This assembly then needed to be "assembled" into machine language.
- If you didn't tell the compiler what data type a variable was, it just assumed it was an integer.
- It had no memory protection or error detection, so even minor bugs would cause your program to crash.
- Some OS-specific functions, like `printf`, had to come from *somewhere*. This meant the machine code had to be "linked" to a C "standard library" which came with the OS.

A "library" is just a collection of helpful functions. If you've ever used an `import` statement in Python, you've used a library.

Modern C

- Now we have *modern* versions of C, like C99, C11, and C23!
- These versions fix… none of the problems I mentioned.

- The C way of doing things is now just considered the "correct" way of doing things, so we're stuck with it.

Here's the "hello, world" program, rewritten in modern C:

```c
#include <stdio.h>
int main(int argc, char **argv) {
  printf("hello, world\n");
}
```

On the upside, modern C compilers do at least usually warn you when you try to do something dangerous. Programming in C still takes a lot of focus and awareness of what you're doing though, so it's really best left for the cases where it's actually necessary.

2 Compilation

How source code becomes machine code

The modern C-style process of compilation can be broken into three steps:

1. A COMPILER turns C code into assembly code (`cc`).
2. An ASSEMBLER turns assembly code into machine code (`as`).
3. A LINKER takes many different pieces of machine code (often from different source code files) and weaves them into a single program (`ld`).

Modern C compilers let you do all of these steps with a single command, but they still do each step separately behind the scenes.

I'm talking specifically about C compilers here, but other compiled languages do more or less the same thing. They generally do a better job of hiding the steps though, so you may never notice; C compilers do a terrible job of hiding the steps, so you'll get mysterious "linker errors" that don't make any sense if you don't know what's going on under the hood.

Compiler Input

A COMPILER reads source code, like this:

```c
#include <stdio.h>
int main(int argc, char **argv) {
  printf("hello, world\n");
}
```

This is meant to be human-readable and portable. The same code would work on Linux on an Intel CPU or macOS on an ARM CPU.

We can compile this by running `cc -S hello.c -o hello.s` to get an assembly file called `hello.s`.

On Macs, `cc` (short for C Compiler) is a symbolic link to the LLVM C Compiler, `clang`; on Linux, it points at the GNU C Compiler, `gcc`. These are the default C compilers on these respective OSes, but you can use `clang` on Linux if you want to (using `gcc` on macOS is difficult; by default `gcc` on macOS is a symbolic link to `clang`).

Compiler Output

A COMPILER writes assembly code, like this:

```
        .file   "hello.c"
        .text
        .section        .rodata
.LC0:
        .string "hello, world"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        call    puts@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 12.2.1 20230111"
        .section        .note.GNU-stack,"",@progbits
```

This is technically still considered human-readable!

Interestingly, the compiler emitted a call to a function called `puts` instead of one called `printf` like in the C source code. This is one of many optimizations that a compiler can do to a program to make it run faster.

Assembler Input

An ASSEMBLER reads assembly code, like this:


```
        .file   "hello.c"
        .text
        .section        .rodata
.LC0:
        .string "hello, world"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        call    puts@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 12.2.1 20230111"
        .section        .note.GNU-stack,"",@progbits
```


We can assemble this by running `as -c hello.s -o hello.o` to get an object file called "hello.o".

Assembler Output

An ASSEMBLER writes machine code, which is not human-readable, but the disassembly looks like this:

```
Disassembly of section .text:

0000000000000000 <main>:
 0:     55                      push   %rbp
 1:     48 89 e5                mov    %rsp,%rbp
 4:     48 83 ec 10             sub    $0x10,%rsp
 8:     89 7d fc                mov    %edi,-0x4(%rbp)
 b:     48 89 75 f0             mov    %rsi,-0x10(%rbp)
 f:     48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # 16 <main+0x16>
16:     48 89 c7                mov    %rax,%rdi
19:     e8 00 00 00 00          call   1e <main+0x1e>
1e:     b8 00 00 00 00          mov    $0x0,%eax
23:     c9                      leave
24:     c3                      ret
```

The call to `printf` is missing because the assembler doesn't know where it is! This incomplete machine code is called an OBJECT FILE.

The "disassembly" of machine code is when a program (called a "disassembler") attempts to turn the machine code back into human-readable assembly code. This process is pretty unreliable and hard to use since machine code gets rid of all information that's unnecessary for running a program; for example, disassembly will almost never have variable names. Still, it can be a useful tool when you're trying to see what exactly your program compiled to.

Linker Input

An LINKER reads incomplete machine code in object files:

```
Disassembly of section .text:

0000000000000000 <main>:
 0:     55                      push   %rbp
 1:     48 89 e5                mov    %rsp,%rbp
 4:     48 83 ec 10             sub    $0x10,%rsp
 8:     89 7d fc                mov    %edi,-0x4(%rbp)
 b:     48 89 75 f0             mov    %rsi,-0x10(%rbp)
 f:     48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # 16 <main+0x16>
16:     48 89 c7                mov    %rax,%rdi
19:     e8 00 00 00 00          call   1e <main+0x1e>
1e:     b8 00 00 00 00          mov    $0x0,%eax
23:     c9                      leave
24:     c3                      ret
```

We could theoretically link this using `ld` to get an executable, but the exact command is complicated and system-dependent. Instead, we'll ask `cc` to do it:

```
cc hello.o -o hello
```

Linker Output

A LINKER writes complete machine code as an executable binary; once again, let's look at the disassembly:

```
0000000000001139 <main>:
    1139:       55                      push   %rbp
    113a:       48 89 e5                mov    %rsp,%rbp
    113d:       48 83 ec 10             sub    $0x10,%rsp
    1141:       89 7d fc                mov    %edi,-0x4(%rbp)
    1144:       48 89 75 f0             mov    %rsi,-0x10(%rbp)
    1148:       48 8d 05 b5 0e 00 00    lea    0xeb5(%rip),%rax        # 2004 <_IO_stdin_used+0x4>
    114f:       48 89 c7                mov    %rax,%rdi
    1152:       e8 d9 fe ff ff          call   1030 <puts@plt>
    1157:       b8 00 00 00 00          mov    $0x0,%eax
    115c:       c9                      leave
    115d:       c3                      ret
```

Now the call to `printf` is fixed (actually called `puts` because of a compiler optimization). This is because the program is now "linked" to the system C library, *libc*, which has a definition of `printf`/`puts`.

Shared Object Files

- We can compile C into a SHARED OBJECT, which is a library any program can use.
- On Linux, these files end with `.so`; on macOS, they sometimes end with `.dylib` instead. The Windows equivalent ends with `.dll`.
- `cc -shared test.c -o test.so` creates a shared object file containing all the functions from `test.c`.
- These "shared objects" are linked at runtime by a DYNAMIC LINKER, which is part of the OS.
  - These `so` files are usually in `/usr/lib/`
  - The environment variable `$LD_LIBRARY_PATH` can add more locations.
- The `ldd` command tells you which shared libraries a program requires to run; if they're not installed, the program will give an error and exit.

So why does anyone use C?

C code is buggy, hacky, and hard to understand. So why is it so popular?

- It's fast: C translates really well into assembly, so C programs are orders of magnitude faster than programs written in some other languages.
- It's omnipresent: everyone else uses C for everything, so the only way to interact with their code is using C.

- It's required: UNIX is *defined* in terms of C. A UNIX-based OS **must** have a C compiler. All the interfaces to ask the OS for anything are designed for C programs.

As a sidenote, many languages actually have a "foreign function interface", which lets their code call C functions. Depending on the language, this could be easy or hard; in C++ it's almost the same as calling a native (C++) function; in Rust it involves jumping through a bunch of hoops to make the Rust and C code compatible. Even when different languages want to interact with each other, they often use C as a *lingua franca*, simply due to how prevalent it is.

3 Package Management

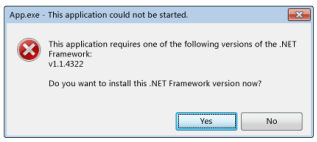
Package Managers

- We get basic functions like `printf` and file I/O from *libc*, which comes with the OS.
- How do we get more complicated functions? (besides writing them ourselves)
- C is so tightly integrated into the OS that we install C libraries the same way we install programs: a package manager.
- On Linux, this comes with the OS: `apt`, `dnf`, `rpm`, `pacman`, etc.
  - If you've ever heard of a Linux "distribution", like Ubuntu, Debian, or Fedora, that basically means what package manager it uses.
- On macOS, this is something we have to install, like `brew`.

Dependency Management

- When you install a program, it may depend on other programs or libraries.
  - For example, `firefox` depends on `ffmpeg`.
- Those programs/libraries may themselves depend on other programs/libraries.
- Trying to figure out the entire dependency tree is somewhere between annoying and impossible, depending on the piece of software.

Without a dependency manager of some sort, you get annoying errors like this:



Solutions

One solution is STATIC LINKING.

- This means every dependency of a program must be *compiled into* that program's binary file.
- This is common nowadays; the iOS and Android app stores (more or less) use this.

Another solution is PACKAGING programs.

- This means every program comes with a *list* of dependencies which are automatically downloaded/installed alongside it.
- This is more widespread; UNIX uses it by default, as do programming language-specific tools like `pip`.

Package Managers

- A PACKAGE MANAGER is a tool which installs programs or libraries, and automatically takes care of resolving dependencies.
- Package managers may be *global*, like `apt` or `brew`, where any program on the same computer can later use a dependency.
- Global package managers have the downside that they may have version conflicts; different programs may need different versions of shared libraries.
- Package managers can also be *local* like `npm`, where they install dependencies into a specific "project".
- Local package managers have the downside that they may do redundant work; different programs may get duplicate copies of the same library.

When you install a program or library with `apt` or `brew`, any other program on your computer can use it. When you install a library with `npm`, only *that specific project* will be able to use it.

4 Python

Python

- Python is a "new"[1] programming language which has taken over a lot of the traditional roles of C within UNIX.
- Python is a SCRIPTING LANGUAGE like `bash`; typically a programmer writes short scripts which combine tools written by someone else.
- Python is INTERPRETED; there's no compilation step needed, but you can't run a `.py` file on a computer that doesn't have `python` installed.
- Python is **compatible with C**.

In fact, almost all scripting languages (including `bash`) are interpreted. The shebang line at the top of a script is actually telling the OS which interpreter to use. There's no equivalent for a C program, since you have to explicitly compile a C program into a standalone binary.

Python Hello World

Python looks a lot more friendly than C:

```
#!/usr/bin/env python3
print("Hello, World!")
```

- Note the shebang line, like a shell script, but calling `python3` instead of `bash`.
- There are two incompatible versions of Python; `python` is usually Python 2, but `python3` is Python 3. **Use `python3` for anything new you write.**

Creating a new Python Project

- If you're writing a little standalone script, you can just create a Python file like you'd create a shell script. Remember the shebang line!
- If you're writing a more complicated program, you probably want to create a new project.

[1] The 90s is "new" as far as UNIX goes.

- As far as Python knows, a project is just a directory, so we can create a new Python project with `mkdir`.

Dependencies

- Much like C programs, we sometimes want our Python program to depend on code someone else wrote.
- We can install programs with the Python package manager, `pip` (or more accurately, `pip3`).
- However, we generally don't want to copy the C approach of installing libraries globally, since then we could only have one version of a library installed for all our Python scripts.
- Instead, we create a VIRTUAL ENVIRONMENT and install our required dependencies in there.

Virtual Environments

- A VIRTUAL ENVIRONMENT is like a little bubble isolated from other Python programs on your computer.
- Anything you install within a virtual environment will stay inside it.
- We can create a virtual environment named "myenv" in the current directory by running the command `python -m venv myenv`.
- We can ACTIVATE the virtual environment by running the command `source ./myenv/bin/activate`. You should see your prompt change to indicate that you're within the environment.
- We could deactivate the environment by running `deactivate`.

Let's practice this! First, create a directory named `python-test`; then create and activate an environment named `myenv` inside it!

Requirements

Let's install some packages inside your new virtual environment.

- First, make sure the environment is activated.
- Now, let's install the `numpy` package, which lets us do vector math: `pip3 install numpy`.
- We can now launch a Python Read-Eval-Print-Loop (REPL) by running `python3`. This lets us type Python commands and have them immediately executed, just like the shell!
- Inside the REPL, run `import numpy` to import the `numpy` library we just installed.
- Try running `numpy.array([1, 2, 3]) + numpy.array([4, 5, 6])`.

Python Scripting

Let's write a Python script.

In fact, let's translate a shell script we wrote in Lecture 4—`my_folder.sh`—into Python!

Create a new file called `my_folder.py` and open it in your favorite editor.

Python Scripting: Example

```
akshay@akshays-thinkpad ~ % python my_folder.py akshay akshay.txt
```

```python
#!/usr/bin/env python3
import os
import sys

def make_my_folder(folder_name, file_name):
    os.mkdir(folder_name)
    os.chdir(folder_name)
    with open(file_name, 'a'):
        pass

make_my_folder(sys.argv[1], sys.argv[2])
```

Be careful about whitespace (spaces and tabs)! Python, unlike C, is extremely sensitive about whitespace. Try using the `:retab` command in `vim` to fix the whitespace in the file.

The `sys` library here lets us access system details, like the command-line arguments to our function. The `os` library lets us do UNIX-related things, like creating and changing directories.

Python Scripting: Example 2

Let's write a Python script that uses a dependency, `vector_norm.py`.

```
akshay@akshays-thinkpad ~ % python vector_norm.py 1 2 3
```

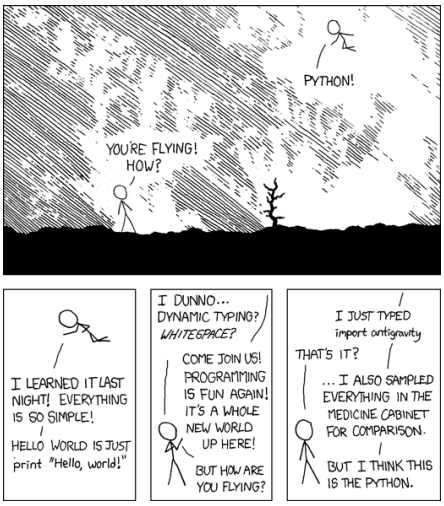
```python
#!/usr/bin/env python3
import numpy as np
import sys

vector = np.array(sys.argv[1:])
print(np.linalg.norm(vector))
```

Packaging a Virtual Environment

The virtual environment you created is customized to your computer, so what do we do if we want to send a Python project to someone else?

1. Create a `requirements.txt` file by running `pip3 freeze > requirements.txt`.
2. Send your Python files (`*.py`) and `requirements.txt` to someone else.
3. Have the other person create a virtual environment and run `pip3 install -r requirements.txt` inside it to install all the requirements.

What's so great about Python?



Source: xkcd 353

What's so great about Python?

Python is fundamentally based on C, but hides it under really good abstractions.

If you really need to use C from inside Python, you can! But most of the time you never need to.

Python scripts can *link against C shared object files*. The entire C software ecosystem is usable from inside Python.

At the end of the day, Python doesn't have to *replace* C, it just has to hide its problems well enough that no one minds them anymore. And, lucky for us, it does exactly that!

Python-C Foreign Function Interface

Say we have a C file `add.c`:

```c
int add(int a, int b) { return a + b; }
```

We can compile it into `add.so`:

```
cc -shared add.c -o add.so
```

We can use it from Python:

```python
import ctypes
lib = ctypes.CDLL("./add.so")
print(lib.add(1, 2))
```

Compared to how hard it is to do this in other languages, this is amazing! The only other mainstream language which makes importing C libraries so easy is C++, and it has to inherit all the problems of C to do so. The fact that turning a C library into a Python library is so easy means Python programmers can use any C libraries they want, without worrying about whether someone has translated it into Python yet.

CS 45, Lecture 9

Version Control I

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Spring 2023

Outline

Contents

1 Review

Learning Goals

In this (and next) lecture, we will see:

- How to safely store your files (code or text)
- How to collaborate on files with others over the internet
- *How to avoid losing your data!*

File Versions

- Many of the files you work with will be text:
  - Source Code
  - Documentation
  - Markup Files
- As you change these files over time, you'll eventually want some way to keep track of different "versions" of the file.
- What we need is a "version control system".

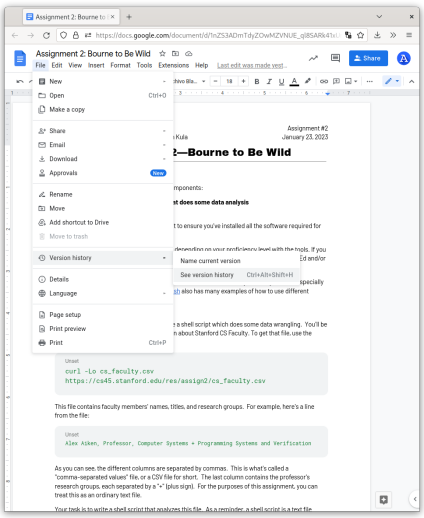
2 Version Control

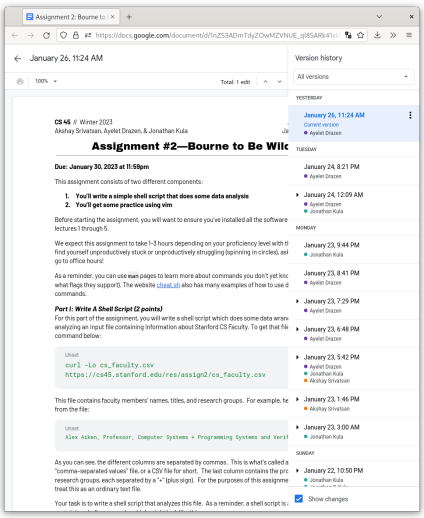
2.1 Version Control Systems

Version Control Systems

- A VERSION CONTROL SYSTEM (VCS) is a piece of software which manages different versions of your files and folders for you.
- A good VCS will let you look at old versions of files and restore files (or information) which you might have accidentally deleted.
- You've seen these before!

Version Control Systems





Version Control Systems

A good version control system:

- Will store many versions of your files
- Will let you "revert" a file (or a part of a file) to an older version
- Will track the order of different versions
- Will ensure each "version" is neither too big nor too small

A great version control system:

- Will let you collaborate on files with other people
- Will help you combine "branched" versions of the files produced by different people working independently

2.2 Comparison of VCSs

There are many different ways to set up a version control system. Let's see the pros and cons of some of the more common ones.

Google Docs

Google Docs automatically keeps track of file history in a basic VCS.

Pros:

- Great for rich text
- Allows real-time collaboration
- Saved on the cloud automatically[1]

Cons:

- Bad for plain text (especially code)
- Requires an internet connection
- Only supports a single "current" version of a single file

You might wonder why you'd ever want multiple "current" versions of a file. One example is when we are planning out this class—we have multiple ideas for each part of an assignment, and try to flesh them out enough to see what works and what doesn't. It would be nice if we could have multiple, equally-valid, versions of the assignment at the same time, but right now the only way to do that is to copy the file.

Copying Files

You can make a bunch of copies of files or folders with `cp` as a simple form of version control. You can compare versions with `diff`.

Pros:

- Works on either rich or plain text (or anything else)
- It's simple and makes it easy to move data between versions

Cons:

- It's messy and a lot of manual work
- It's hard to tell what the relationship between different versions is
- It takes a lot of hard drive space

Zip Files

Instead of just `cp`ing folders, we could bundle them up into a Zip file (a single file which can be "unzipped" into a folder).

Pros:

- Tracks versions for an entire folder at once
- Easy to share a version with someone else (email)

Cons:

- It's still a lot of manual work
- It's hard to tell what the relationship between different versions is
- It's hard to extract a single file from an old version

[1] Stay tuned for our lecture on the cloud!

Zip Files++

- What if we had a tool which did all this zip file stuff automatically?
- We could tell it to take a "snapshot" of a directory, and it would save all the changes in it.[2]
- We could ask it to recover an old version of a specific file, or to reset everything to an old version to "undo" our work.
- The tool could track the relationships between different versions, so we can have multiple "current" versions at the same time.
- If we want to combine different versions, the tool can automatically do it for us (instead of us copying and pasting the parts together).

Git

`git` is a version control system which tracks "commits" (snapshots) of files in a REPOSITORY.

- Git stores old versions of files in a hidden folder (`.git`), and automatically manages them.
- We can tell Git to keep track of certain files, and tell it when to take a snapshot.
- We can ask Git to go back to an old snapshot (even for a single file).
- We can ask Git to keep track of who's working on what, so multiple people can work on different things without conflicting.
- If we want to combine multiple people's work, we can ask Git to automatically merge them together. If it can't for some reason, it'll ask us to manually merge them.

There are several other version control systems in use, but Git is by far the most popular. Different companies sometimes decide to use different VCSes for various reasons; for example, Facebook/Meta uses a version of Mercurial (another VCS) they call "Sapling". However, these different VCSes are conceptually very similar, so moving from one to another is pretty easy.

3 Git

3.1 Linear History

Basic Workflow

The simplest way to use git is the "linear" workflow, which is the same way you'd use Google Docs:

1. `git init` to enable Git in a certain directory
2. `git add` any files you want Git to "track"
3. `git commit` the currently "staged" changes to save a snapshot
4. make changes to your files
5. `git add` the changed files to "stage" them again
6. Repeat from 3

You can use `git log` to see your commit history, and use `git status` to see the current state of staged/unstaged/untracked changes.

Before you do anything in Git, you should tell Git who you are:

[2] Or, even better, we could tell it *which* files to save in the snapshot. Everything else stays as it was in the previous snapshot.

```
git config --global user.name "<YOUR FULL NAME>"
git config --global user.email "<YOUR PUBLIC EMAIL ADDRESS>"
```

Note that these configuration values will be permanently baked into any commits you make, so other people may be able to see them. Use a pseudonym or a pseudonymous email address if you don't want to publish your information; see the GitHub Docs for more info.

Basic Workflow

Demo

Let's practice how to:

- Create a new Git repository
- Commit a new file
- Commit changes to files
- Revert commits
- Look at an old version of a file
- Compare two versions of files
- See your commit history

3.2 Branching Workflow

Branching Workflow

We can also split our "repo" into multiple BRANCHES, which are like alternate versions of a folder. This means different people can work on different things without interfering with one another.

1. Make sure your repository is "clean" (i.e., you have no uncommitted changes).
2. `git checkout -b <branch>` to create a new branch and move to it; at this point, the new branch will be identical to the old one.
3. Make changes, `git add`, `git commit` as usual
4. `git checkout` to switch between branches

3.3 Combining Branches

Branching Workflow

Combining Branches

Now that we have multiple branches, we probably want to join them back together at some point.

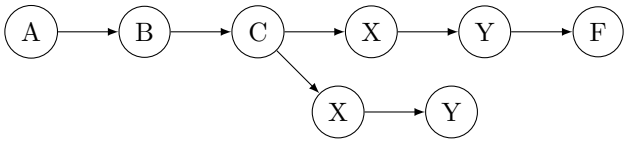
There are several ways to do this:

- `git merge` two branches into one
- `git merge --fast-forward` a long branch onto a shorter version of itself
- `git rebase` one branch onto another branch
- `git cherry-pick` a specific commit from one branch to another

Branching Workflow

Fast Forwarding

The simplest case of MERGING is called FAST-FORWARDING.

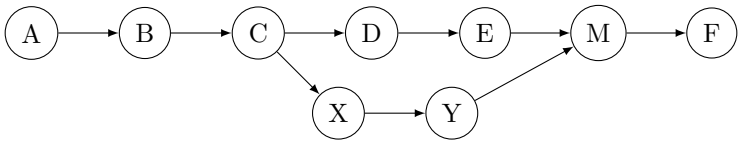


Assume there was a main branch containing A and B. We then created a new branch based on B, containing X and Y. Merging the main branch and our new branch is easy—our new branch is just a longer version of the main branch, so we can "fast-forward" the main branch to catch up. We can then commit F on the main branch, based on Y.

Branching Workflow

Merging

MERGING (in general) creates a MERGE COMMIT to join the two branches.

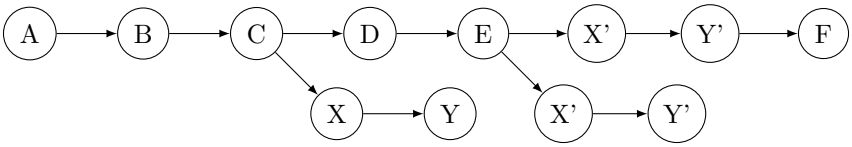


In this case, assume as before that we had a main branch containing A and B. We created a new branch and added X and Y. However, while we were doing that, someone else added D and E to the main branch. Our branch has now DIVERGED from the main branch; we both have different, possibly contradictory ideas of what the "real" state is. We fix this by creating a MERGE COMMIT M, which contains a combination of E and Y; if E and Y were contradictory, Git will ask the user to resolve the MERGE CONFLICT by picking which version to keep. Now we're back to a single "real" version M, so we can add F on the end.

Branching Workflow

Rebasing

REBASING moves the "base" of a branch to be a different commit. *REBASING edits Git's history to make FAST-FORWARDING possible.*

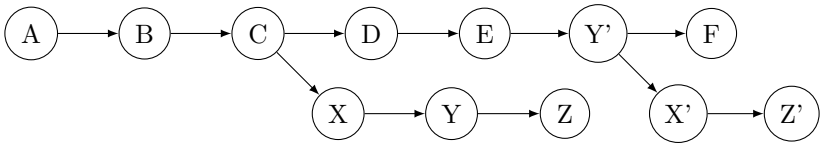


Again assume that we had a main branch containing A and B, from which we created a new branch and added X and Y. Once again, someone else added D and E to the main branch while we weren't looking. However, this time, we want to maintain the illusion of "linear history"; the idea that every version has exactly one predecessor and one successor. This is obviously not true in this case—the commit C has two successors—so we "rebase" our branch. This creates a new commit X' which combines X and E, and a new commit Y' which combines Y and X'. However, if you look at the Git history, you never see a reference to the original X or Y; for all practical purposes, X and Y never existed. Now we're back to the fast-forward case; our branch is a longer version of `main`, so we can fast-forward `main` to include X' and Y'. Now we can commit a new F based on Y'; if anyone in the future ever looks at the repository, it'll look like everything up to F was a linear, non-branching, sequence of changes.

Branching Workflow

Cherry-Picking

CHERRY-PICKING copies a *single commit* from one branch to another branch. *CHERRY-PICKING and rebasing is a good way to move a single commit from one branch to another.*



Sometimes we want to grab a specific change from another branch, without merging or rebasing the entire branch. This is a use case for "cherry-picking", selecting a specific commit to copy into another branch. In this case, assume a similar setup to the previous cases—a main branch containing A, B, C, D, and E, and a new branch containing A, B, X, and Y, and Z. We're on the main branch at E, and we realize we need something that was added in commit Y, but we **don't** want to include commit X or commit Z. We can't literally just copy Y onto our main branch, since it depends on X and doesn't include anything from D or E, so we create a new commit Y'. Y' is like a mashup of Y and E, but without any of the stuff from X. Now we can commit F based on Y', and F will contain no references to X or Z.

Note that this can get weird if you later do a merge—Y and Y' will both be in your history! If you cherry-pick commits like this, you probably want to do a rebase to make your history less confusing; in this case we could rebase the new branch onto Y', creating a new X' and Z', which would effectively erase the original X, Y, and Z from the new branch.

Branching Workflow

When to merge/rebase/cherry-pick?

- **fast-forward** when possible (`git merge --ff-only`).
- **rebase and then fast-forward** if possible, i.e., if you're the only one working on the branch; **never** rebase a branch other people are using (`git rebase` and `git merge --ff-only`).
- **merge** if neither of the above are possible (`git merge`).
- **cherry-pick** if you want to copy a specific commit to another branch (`git cherry-pick`)[3].

Some projects insist upon having linear history in `main`, which means any merge commits will be rejected. In this case, you should first `rebase` your changes onto the most recent `main`, then `merge --ff-only` to fast-forward `main` to include your changes. Most projects are okay with merge commits, so you can just use `merge` and forget that `rebase` ever existed.

Branching Workflow

Branching Demo

Let's practice how to:

- Split our repository into two branches
- Switch between branches
- Make commits on either branch
- Merge two branches together

[3]This is pretty rare, I've only used it a handful of times.

To Be Continued...

We'll pick back up with merge conflict resolution and collaboration in Lecture 10.

Some commands which (probably) came up during class:

`git checkout`: essentially means "move to a different commit"; doesn't change your git history

`git reset`: "resets" the entire repository to the way it was in an old commit (and changes git history to match)

`git revert`: "undoes" a specific old commit by creating a new commit that does the opposite

Note that, even though Git commits are technically versions, Git's commands often operate on the *changes* between versions.

You can view all your active branches with `git branch`, and delete branches with `git branch -d`. Remember that a "branch" is basically just a named version of a file; "main" is typically the official version, while other branches are so-called "feature branches" where different people can experiment with adding different things. Generally a feature branch should be "owned" by a single person to avoid conflicts (only that person should ever change that branch).

One last thing: git has separate man pages for each subcommand, since they're each so complicated they need their own docs. For example, the man page for `git commit` can be viewed using `man git-commit`.

CS 45, Lecture 10

Version Control II

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Spring 2023

Outline

Contents

1 Review

Overview

Last lecture, we saw:

- How to keep track of linear (ordered) histories of files
- How to turn non-linear version history into pseudo-linear history

In this lecture, we will see:

- How to resolve merge (or rebase) conflicts
- How to collaborate on files with others over the internet
- How to back up your files and their history on the internet

Git is Confusing

- Git is confusing!
- Ask as many questions as you need, and don't let me move on if you don't yet understand something.

```
commit c8f3dcfb80e87d4aa334f6bcdd541ddb78135881 (origin/master, origin/HEAD)
Merge: 480cbe2 2a7f97b
Author: engler <ddd.rrr.eee@gmail.com>
Date:   Thu Feb 13 14:27:50 2020 -0800

    git.  sucks.

    Merge branch 'master' of github.com:dddrrreee/cs140e-20win
```

Terminology

HEAD is the branch you're currently looking at

a branch is a named version of a repository

fast-forwarding means moving an old branch "forward" to add new commits from a more recent branch

merging is a way of combining branches by creating a single "merge commit"

cherry-picking is a way of moving commits from one branch to another

rebasing is a way of moving an *entire branch* to have a different "base"

2 Merge Conflicts

Merge Conflicts

Definition 2.1 (merge conflict)**.** A merge conflict is what happens when you try to combine two contradictory branches. Git can't always figure out how to resolve the contradiction, so it'll ask the user (you).

- Git normally resolves merge conflicts automatically.
- Some conflicts have multiple valid resolutions (e.g., what if one person edited a file that another person deleted?).
- If Git doesn't know what to do, it'll ask you to resolve the conflict.

Merge Conflicts

Git will tell you which files conflicted, and tell you to resolve the commits and commit the results:

```
Auto-merging hello.txt
CONFLICT (content): Merge conflict in hello.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Merge Conflicts

Conflict Markers

Git will also add conflict markers to the files:

```
Hello, my name is Akshay Srivatsan.
<<<<<<< HEAD
I'm doing my PhD in the Stanford CS department.
=======
```

```
I am a PhD student studying CS at Stanford.
>>>>>>> add-major
I'm currently co-teaching CS45 and doing research.
```

This might look scary, but it's not that bad!

Merge Conflicts

Conflict Markers: The Base Branch

The top part (labeled `HEAD`) are the changes in the base branch (the branch you're currently on):

```
Hello, my name is Akshay Srivatsan.
<<<<<<< HEAD
I'm doing my PhD in the Stanford CS department.
=======
I am a PhD student studying CS at Stanford.
>>>>>>> add-major
I'm currently co-teaching CS45 and doing research.
```

Merge Conflicts

Conflict Markers: The Incoming Branch

The top part (labeled with a branch name or commit message) are the changes in the incoming branch (the one you're merging):

```
Hello, my name is Akshay Srivatsan.
<<<<<<< HEAD
I'm doing my PhD in the Stanford CS department.
=======
I am a PhD student studying CS at Stanford.
>>>>>>> add-major
I'm currently co-teaching CS45 and doing research.
```

When you see these conflict markers, all you have to do is make the files look the way you want them to look at the end. In this case, I added the text "I'm doing my PhD in the Stanford CS department." on `main`, but I added the text "I am a PhD student studying CS at Stanford." on the branch `add-major`. When I tried to merge `add-major` into `main`, Git didn't know what to do, so it's asking me. Now I can choose either of the two sentences to keep, and delete the other (or I could keep both of them, if I wanted).

As an aside, you might see the name `HEAD` pop up in Git. This basically just means "what commit you're currently looking at".

Merge Conflicts

Resolving a Conflict

Pick how you want to resolve the conflict (i.e., decide what the "correct" result of the merge is), and make the file look that way!

```
Hello, my name is Akshay Srivatsan.
I'm a PhD student in the Stanford CS department.
I'm currently co-teaching CS45 and doing research.
```

In this case, I mixed together both versions. The "correct" answer often depends on what exactly you're doing, which is why Git can't figure it out for you.

Merge Conflicts

Commiting the Merge

Resolve all the conflicts in all the files however you want, then:

1. `git add` your changes to track them
2. `git commit` the changes (with no message)

Git will auto-generate a message, and open your `$EDITOR` to have you confirm it:

```
Merge branch 'add-major'
```

Save the file in your editor and close it (`:wq` in Vim), and Git will save the merge commit. That's it—the merge conflict is gone!

That's all you have to do—make the files look "correct", then commit! A merge conflict really isn't as bad as people sometimes make it sound; all it means is that there are multiple ways to merge the two branches, and Git wants you to pick one.

If you decide to use `rebase` instead, the process is pretty much the same—just run `git rebase --continue` instead of `git commit` at the end.

Merge Conflicts

Rebase Conflicts

Resolve all the conflicts in all the files however you want, then:

1. `git add` your changes to tell Git you fixed them
2. `git rebase --continue`

Since rebasing doesn't create a merge commit, you don't run `git commit`; use `git rebase --continue` instead!

Remember, rebasing happens *backwards*; the base branch (the one onto which you're rebasing) becomes `HEAD`, and the "feature" branch becomes the incoming branch.

Resolving Merge Conflicts

To resolve a merge conflict:

1. Don't panic!
2. Look at the files in conflict (run `git status` to see what's going on).
3. Fix each conflict, one-by-one.
4. When you're done, `git add` all the fixed files and `git commit`.

Let's practice!

Merge conflicts usually happen in shared repos, so let's CLONE one of my repos onto your computer:

```
git clone https://github.com/Akshay-Srivatsan/cs45-23win-demo-repo.git
```

We'll go into more detail about how shared repositories work in the last section of this lecture, but for now:

- You can "clone" a shared repository using `git clone`, which makes a local copy.
- You can "fetch" commits from the shared repository into yours using `git fetch`. The commits will go into a separate branch so they don't conflict with yours; by convention, the branch names have the prefix "origin/" prepended to them, so `main` goes into `origin/main`.

Pulling Changes

You might have seen references to the `git pull` command before. This is a combination of two commands, but the exact two depends on your Git version and configuration:

`git pull --ff-only`: `git fetch` and `git merge --ff-only` (Default)

`git pull --no-rebase`: `git fetch` and `git merge` (Old Default)

`git pull --rebase` `git fetch` and `git rebase`

Depending on your preferences, you can configure `git pull` to do any of these.

I personally use `git pull --rebase` the most often, since I don't like having merge commits in my repo history.

3 Commit Etiquette

Commit Messages

Git only saves work that we've committed, so we want to commit as often as possible, but…

- Other people will also look at your commit history to see what you did.
- Your commit messages in the history should be short and specific, but descriptive enough that someone new can understand what they do.
- Similarly, each of your commits should do a single thing, so a single message can describe it easily.
- Good commits are BISECTABLE; you should be able to checkout any commit in `main` and get a valid (e.g., compilable) state of your repo.

Writing good commit messages is part of being a good programmer!

These goals might seem contradictory; how do we commit as-often-as-possible, but still make sure each of our commits are meaningful and discrete? The answer is: we don't! While we're developing, we commit as often as we want. Then, when we're ready to share our work with others, we *edit our commit history* to make it look like we made nice, easy-to-understand, discrete commits.

Squashing Commits

We can commit often locally but still have meaningful commits in the end by SQUASHING commits together with INTERACTIVE REBASE.

Editing History

Interactive rebasing edits history! Don't do this on a branch you share with other people (like `main`). In general, only do this on commits you **have not** pushed. Otherwise, you'll have to FORCE-PUSH (`git push --force`) your changes, which will **destroy** everyone else's changes.

You can start an interactive rebase using the command `git rebase --interactive <base>`; for example, `git rebase --interactive main` will let you edit every commit that's in your branch but not in `main`.

Interactive Rebasing

Git will open `$EDITOR` with a list of actions (which you can edit!).

```
pick 0cd3296 start working on new file
pick 594a80c continue working
pick 162392b almost done
pick bf45520 done
pick c545ae9 oops, had a bug
pick 9b3d056 fix the bug for real this time
```

Each line represents one commit. The first word is a "command"; `pick` cherry-picks (i.e., includes) the commit in the new history, `reword` lets you edit the commit message, `edit` lets you change the commit contents, `squash` and `fixup` both squash the commit into the previous one, and `drop` removes the commit.

You might notice that the default behavior here is to cherry-pick every commit. This is the exact same as a normal (non-interactive) rebase! What we called "rebasing" earlier is actually a special case of editing history, but you can go far beyond that with interactive rebasing.

Squash and Fixup Commits

Squash commits let you specify that two commits are closely related, so they should be combined into a single commit with both messages.

Fixup commits let you specify that a particular commit just "fixes" a previous one, and therefore should be absorbed into the previous commit.

```
reword 0cd3296 start working on new file
squash 594a80c continue working
squash 162392b almost done
squash bf45520 done
squash c545ae9 oops, had a bug
squash 9b3d056 fix the bug for real this time
```

Note that we could also use `fixup` here, the only difference is whether the original message gets saved or thrown away. In this case, we're using `reword` on the first commit anyway, so it's a moot point.

Rewording Commits

When you want to `reword` a commit, Git will open `$EDITOR` and ask you for a new commit message. Enter the message you want, save, and quit.

```
Add a file providing more information about the project

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Fri Feb 3 22:34:02 2023 -0800
```

Amending Commits

If you want to edit the most recent commit you made (i.e., `HEAD`), you can skip the rebase and just AMEND it, using `git commit --amend`.

This will also let you edit the commit message of the last commit.

If you want to add to an earlier commit but don't want to do the full interactive rebase yet, you can use `git commit --fixup <hash>` to mark a commit as being a fixup commit of an earlier commit.

You can then use `git rebase --interactive --autosquash <base>` to automatically absorb your fixup commits into the original commits.

Again, only do this if **no one else** is using your branch.

Good Commit Messages

From Stanford Stagecast: Proleptic:

```
b68f706 Add training trick for handling missing notes
7c0afae Fix audio issue in metronome
fa18a3d Add a metronome for beat tracking
739aa0a Add test for per-millisecond prediction on MIDI files
dc4693a Use last eight piano roll columns to predict next
eb327b4 Fix timing bug in MIDI file parser
b46c6f6 Add MIDI training demo program
a624b32 Switch to cross-entropy loss for MIDI classifier
db3c6f3 Fix bug in MIDI parser
5b0f299 Return effective learning rate from training wrapper
726886d Print more relevant training info from full-piano predictor
```

These commit messages aren't perfect, but they're short, descriptive, and make it clear what one thing each commit does. They generally follow the format VERB-OBJECT (although the implied subject isn't always consistent), and they describe which part of the codebase they're touching.

These commits are also bisectable; that means, if I notice a bug, I can *binary search* to figure out which commit introduced the bug. Git actually has a tool for this built-in, called `git bisect`—you give it a start and end commit (the start commit definitely doesn't have the bug, and the end commit definitely does), and it'll checkout commits in between to help you figure out where the bug was introduced.

Bad Commit Messages

From CS 140E, Winter 2023:

```
a527839 minor
adaf72e minor
c9c6193 minor
d64a6ef minor
ff2636e minor
4a988f2 minor
cb901d5 minor
8d4e80a minor
53b5e84 minor
0321f79 minor
4126899 minor
f1d7231 minor
cefba82 minor
```

```
44e7773 minor
571c20b minor
059cb3f minor
eaa75ae minor
ebbe9db minor
13570e0 minor
3e51470 minor
95a0fad minor
5d2c780 minor
d5caf55 minor
c26b868 minor
080ddf2 minor
f492a3f minor
```

```
2e67fd8 minor
e530f6e minor
70387f2 minor
e3d971e minor
91b236e minor
de176a8 minor
461e76a minor
48cd0ff minor
0543316 minor
40b48f6 minor
fb0ec84 minor
3a124af added basic files.
```

These commit messages are useless; if you try to look back at your commit history, you're going to have no idea what's going on. If one of these commits had a bug, it's hopeless to try and figure out which one introduced it. Additionally, messages like this are often a symptom of poorly-separated commits; it's

impossible to describe what a commit does because each commit does way too many things (or, alternatively, a single discrete change is scattered across many commits, so there's no meaningful description of what a single commit did).

For the record, these are both real sequences of commit messages from projects I've worked on. You'll probably run into both ends of this as you work with different groups of people, but whenever you can, try to make your commit messages more like the first example.

4 GitHub

GitHub

- One of the most common reasons to use Git is to be able to collaborate.
- Git has built-in support for REMOTE repos, which exist on the internet somewhere.
- You CLONE a remote repo to get a local copy. You can then make commits on the local repo. The remote repo is conventionally named `origin`.
- You FETCH while inside a clone, which copies the remote `main` branch into a branch called `origin/main`.
- You MERGE or REBASE your local `main` into/onto `origin/main`.
- You PUSH your new `main` back to the remote, which updates its `main` and your `origin/main`.

Remember, you can combine `fetch` and `merge` (or `rebase`) using `pull`, if you want to.

You can actually have multiple remote repos for a single local repo. For example, you might have `origin` as your copy of the repo on GitHub, and `upstream` as someone else's copy of the same repo from which you want to cherry-pick changes.

GitHub Demo

Let's create a new repository on GitHub!

You'll need the `git` command and the GitHub CLI (`gh`).

1. Go to `https://github.com/new` and pick a name.
2. Click "Create repository" to continue.
3. Run `git clone` with the URL of your new repo.
4. Run `gh auth login` from inside your new clone. Tell `gh` that you want to use it to authenticate with `git`.
5. Make some changes (add a file), and run `git push` to upload them!

GitHub Demo

Let's start collaborating!

1. On the GitHub website for your repo, go to "Settings" and click on "Collaborators".
2. Add the person sitting next to you as a collaborator!
3. Make a clone of their repo, make some changes, then commit and push them. Use `git fetch` or `git pull` to download their changes to your repo.
4. What happens if you both try to edit the same file at the same time?

5. Can you push a new branch to your partner's repo?[1]

Pull Requests

- It's dangerous to give access to the `main` branch on your repo to everyone; someone might start messing with it!
- In "Settings/Branches", you can enable BRANCH PROTECTION for `main`. Specifically, you can enable "Require a pull request before merging".
- A PULL REQUEST[2] is a way to review a change before merging it. You (the repo owner/maintainer) can choose whether to approve or reject the request.
- To create a pull request: create a new branch, make your changes, push your new branch, then run `gh pr create`.

When to use Git

- When you want to look at past versions of a folder.
- When you want to be safe from accidentally overwriting your work.
- When you want to collaborate with other people asynchronously (use GitHub!).
- When you want to keep a backup copy of a folder with full history (use GitHub!).
- You want to "fork" a project already using Git/GitHub and contribute back to it.

I personally use Git pretty much whenever I write code, and even sometimes when I'm writing prose. Even my lecture slides for this class are tracked in Git… and one of the lectures in Winter Quarter only happened because Git saved my slides from accidental deletion.

I actually use Git so often that I have a bunch of aliases, both in my shell and in Git itself, to make using it faster. From my `.zshrc`:

```
alias ga="git add"
alias gc="git commit"
alias gc!="git commit --amend"
alias gcmsg="git commit --message"
alias gca="git commit --all"
alias gcam="git commit --all --message"
alias gca!="git commit --all --amend"
alias gp="git push"
alias gf="git fetch"
alias gfm="git pull --no-rebase"
alias gfr="git pull --rebase"
alias gff="git merge --ff-only"
alias gst="git status"
alias gr="git rebase"
alias gri="git rebase --interactive"
alias glog="git log"
alias gco="git checkout"
alias gb="git branch"
```

And from my `.gitconfig`:

[1] Hint: you might have to use the `--set-upstream` flag; Git will tell you exactly what to do.
[2] This is misleadingly named, it's really a "merge request"

```
[alias]
  co = checkout
  c = commit
  st = status
  b = branch
  hist = log --pretty=format:"%Cred%h%x09%Cgreen%cs%x09%Creset%s%x20%Cblue[%an]%Creset"
  uncommit = reset --soft HEAD^
  amend = commit --amend
  histedit = rebase -i origin/main
  unstash = stash pop
  unadd = restore --staged
  skip = update-index --skip-worktree
  unskip = update-index --no-skip-worktree
  skipped = ! git ls-files -v | grep '^S' | cut -d' ' -f2
  list = ls-files -v
  ff = merge --ff-only
  delete-remote-branch = push origin --delete
```

I wouldn't recommend copying all of these, since some of them are particular to the way I use git, but they might give you ideas of ways you can make your own Git usage more convenient. A git alias can be run as a git subcommand, so I can run `git unadd hello.txt` instead of `git restore --staged hello.txt`.

CS 45, Lecture 17

Encoding

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Spring 2023

Outline

Contents

1 Overview

What is Encoding?

- Computers are great at numbers.
- Computers are terrible at everything else.
- We need to turn everything else into numbers for computers to deal with them.

Actually, that's not strictly true. Computers are terrible at numbers as well; the only thing they can *really* handle is truth values (true or false).

2 Encoding Numbers

Encoding Numbers

Numbers themselves can be encoded many ways:

roman: XLV $= 50 - 10 + 5$

binary $101101_2 = 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^2$

octal: $55_8 = 5 * 8^1 + 5$

decimal: $45 = 4 * 10^1 + 5 * 10^0$

hexadecimal: $\texttt{2d}_{16} = 2 * 16^1 + 13$

For computers, we want a way that works well with digital logic circuits (i.e., using only 0 or 1).

Human Representation

Instead of specifying the base every time we write a number, computer scientists often use different notations as shorthand:

binary `0b101101`

octal: `0o55` or `055`

decimal: `45`

hex: `0x2d`

Binary

Binary looks great for computers (it's just zero or one), but there's a few problems:

- It's unwieldy for humans: `0b1010000001110` vs 5134.
- It's hard to convert to decimal (but easy to convert to hexadecimal or octal).
- There's no "maximum" number, so there's no maximum number of bits to store "one number".
- There's no easy way to represent negative numbers (like -01100001_2).
- It can't represent non-integer numbers (e.g., what's $\frac{01_2}{10_2}$?)

In practice, computer scientists often represent numbers as hexadecimal for concision or decimal for convenience, even though the underlying representation is binary.

Sized Integers

The size problem is solved by defining different "sizes" of numbers, with different numbers of bits:

nibble 4 bits

byte 8 bits

short 16 bits

int 32 bits

long 64 bits

Different CPUs have different "default" sizes; this size is often called "word". For example, 64-bit machines (most computers today) have 64-bit words and 32-bit "halfwords".

These particular names aren't set in stone; sometimes they have different meanings based on context. On some old computers, an `int` would be 16 bits and a `long` would be 32 bits. However, most of the time nowadays, they're used in the way described here.

Processing numbers bigger than a "word" on a computer is usually slow, while processing numbers smaller than a word is fast.

Signed Integers

The negative numbers problem is solved by using Two's Complement encoding, wherein a negative number is produced by:

1. Starting with the positive version of that number.
2. Inverting every bit (0 to 1, 1 to 0).
3. Add 1 to the number.

This means 31 (as a byte) becomes `0b00011111`, but -31 becomes `0b11100001`. The uppermost bit can be thought of as a "sign bit".

To interpret a number, you need to know if it is SIGNED or UNSIGNED; `0b11100001` could be -31 or 225.

Integer Ranges

The range of numbers we can represent depends on the encoding we use:

Width	Unsigned	Signed
8	$[0, 255]$	$[-128, 127]$
16	$[0, 65535]$	$[-32768, 32767]$
32	$[0, 4294967295]$	$[-2147483648, 2147483647]$
64	$[0, 2^{64} - 1]$	$[-2^{63}, 2^{63} - 1]$

Overflow and Underflow

Overflow/Underflow

A huge number of real-world bugs come from integer OVERFLOW and UNDERFLOW.

```
// unsigned byte overflow:
(uint8_t)(255 + 1 == 0);
// unsigned byte underflow:
(uint8_t)(0 - 1 == 255);
// signed byte overflow):
(int8_t)(127 + 1 == -128);
// signed byte underflow:
(int8_t)(-128 - 1 == 127);
```

Endianness

When a single number is represented by multiple bytes, there are two valid ways to order those bytes: with the "big end" first or the "little end" first.

`0xAABBCCDD =`

Offset	Big Endian	Little Endian
0	0xAA	0xDD
1	0xBB	0xCC
2	0xCC	0xBB
3	0xDD	0xAA

Most CPUs are little-endian, but most network protocols are big-endian. The way we write numbers (e.g., `0xAABBCCDD`) is big-endian.

Floating Point

Real numbers (i.e., non-integers) are represented using "floating-point" arithmetic.

A number like 1701.47 could be written as 170147×10^{-2}.

The IEEE 754 Standard specifies how computers would store 170147 and -2 so they can do math easily.

Floating point numbers are inherently approximations, and prone to inaccuracies:

```
0.1 + 0.2 == 0.3 // false
0.1 + 0.2 // 0.30000000000000004
```

Sometimes 32-bit floating point numbers are called `float` and 64-bit floating point numbers are called `double`.

3 Text Encoding

Now that we know how to represent numbers, we can use those numbers to represent more complicated things.

3.1 English Text

Text

Let's say we want to save this file:

```
Hi!
```

One idea might be to do a simple substitution: A=1, B=2, C=3, etc. Problems:

- We need to handle both upper- and lower-case letters.
- We need to handle punctuation.
- We need to handle numbers.
- We need to handle special things like "space" and "enter"/"return".

ASCII

In the 1960s, Bell Labs created the American Standard Code for Information Interchange, which deals with all this:

```
Hi!
```

becomes

```
48 69 21
```

This is a 7-bit encoding (each letter/symbol takes 7 bits of data), but is often treated as an 8-bit encoding for convenience.

This encoding became super popular and is used everywhere.

ASCII Table

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Image Credit: https://simple.wikipedia.org/wiki/ASCII

3.2 Text Around the World

ASCII outside America

- ASCII (also called ISO 646-US) is unabashedly America-centric.
- It only supports the "basic latin alphabet" (the 26 letters in English).
- Some other countries created slight modifications of ASCII for their use:
  - ISO 646-GB adds £ (UK English).
  - ISO 646-FR adds à, ç, é, ù, and è (French).
  - ISO 646-ES adds ñ, Ñ, ¡, ¿, and ç (Spanish).
- In some cases, people used the extra top bit to encode up to 256 extra characters in an 8-bit encoding, e.g., Code Page 1252.

Code Page 1252

- Code Page 1252 was historically the default text encoding on Windows.
- The subset ISO 8859-1 is also considered equivalent in most cases.
- These are the most widespread single-byte (8-bit) text encodings in the world today, but still only feature on 1.7% of websites.

Code Page 1252 Table

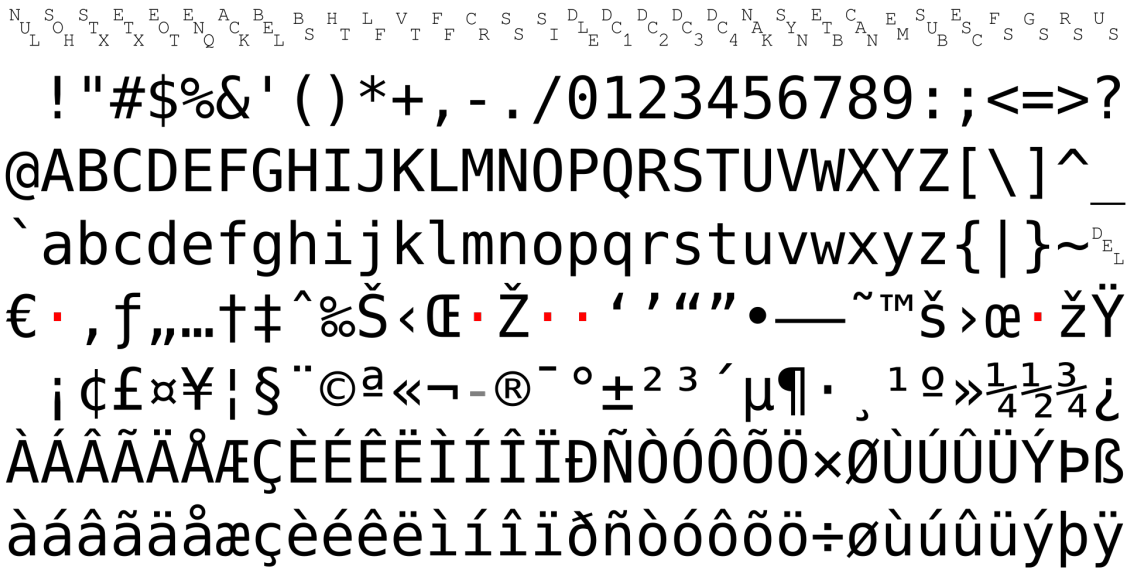


Image Credit: https://en.wikipedia.org/wiki/Windows-1252

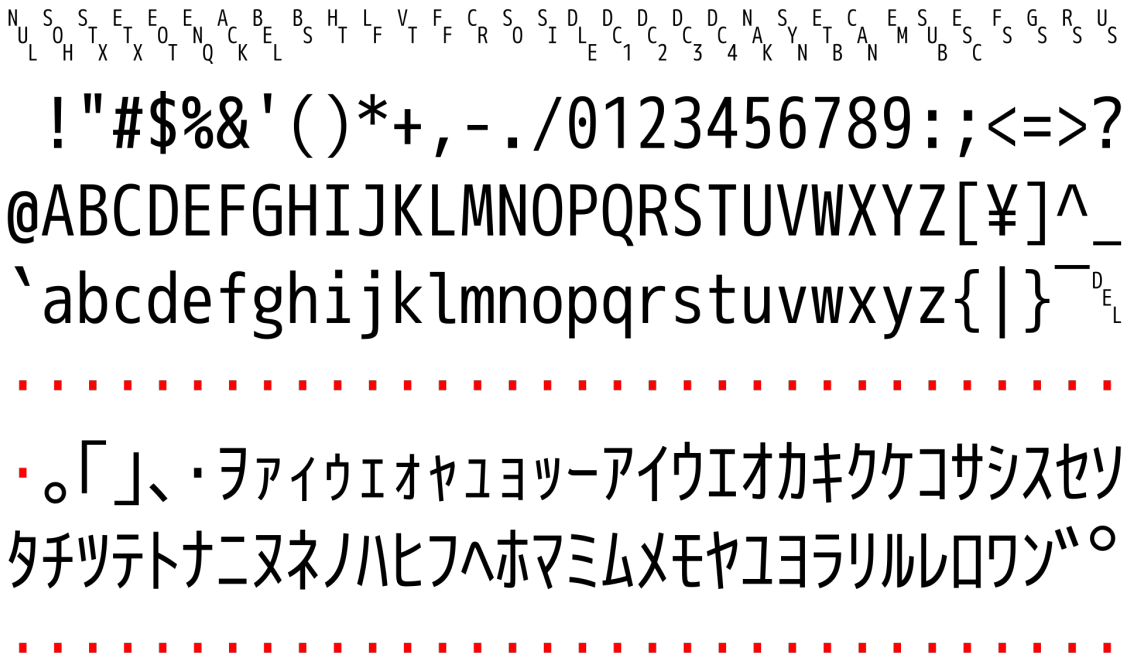
By now you've probably noticed an issue—even though we can encode a bunch of letters from different Latin-based alphabets, we still can't represent other alphabets at all (e.g., Greek, Cyrillic, Chinese, Korean).

Alternatives to ASCII

Countries which use other alphabets came up with their own ASCII/ISO 646 variants.

These generally encode all the ASCII characters for compatibility, but take advantage of the 8th bit to encode their own characters (or, often, a subset thereof).

JIS X 0201 (ISO 646-JP)



This was superseded by Shift JIS, which uses 2 bytes (16 bits). Image Credit: https://en.wikipedia.org/wiki/JIS_X_0201

Custom Encodings

Some companies also came up with custom encodings for their products.

Generally these would only be usable on devices made by that company, so they tried to remain backwards-compatible with ASCII.

Over time, there grew to be thousands of incompatible character sets.

Custom Characters

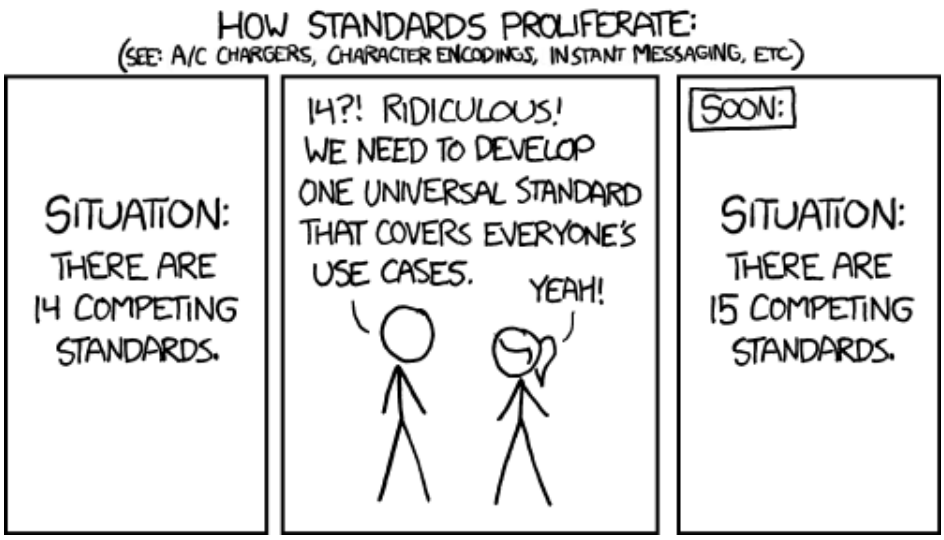
In 1997, J-Phone (a Japanese phone company) released this character set:

Image Credit: https://emojitimeline.com/the-real-original-emojis/ This was the first appearance of EMOJI in a character set.

3.3 Unifying all the Codes

This whole situation is pretty well summed up by this XKCD:

Standards



Source: xkcd 927

Encoding Chaos

Mixing up encodings is problematic:

à®...à®•à¯à®·à®¯à¯
à®¶à¯à®°à¯€à®µà®¤à¯à®¸à®©à¯

/ÊŒkÊ‚ÊŒj É›É¾iËÊ‹ÊŒdsÊŒn/

à®µà¯‡à®±à¯ à®Žà®´à¯à®¤à¯à¯ à®®à¯à®±à¯ˆà®•à®³à®²à¯: Akshay Srivatsan
à®®à®±à¯à®±à¯à®®à¯ à¤...à¤•à¥à¤·à¤¯à¥ à¤¶à¥à¤°à¥€à¤µà¤¤à¥à¤¸à¤¨ à¥
à®®à®±à¯à®±à¯à®®à¯ ð'€...ð'€"ð'†ð'€±ð'€¬ð'† ð'€°ð'†ð'€ð'€»ð'€¯ð'€cð'†ð'€²ð'€¦ð'†
à®®à®±à¯à®±à¯à®®à¯ aká¹£ay srÄ«vatsan à®®à®±à¯à®±à¯à®®à¯ ð'Œ...ð'Œ•ð'ð'Œ·ð'Œ¯ð'
ð'Œ¶ð'ð'Œ°ð'€ð'Œµð'Œ¤ð'ð'Œ¸ð'Œ¨ð'

à®¤à®®à®¿à®´à¯ à®®à¯Šà®´à®¿à®¯à¯ à®¤à®µà®¿à®°, à®‡à®¨à¯à¯ à®¤
à®‡à®£à¯ˆà®¯à¯à®¤à®³à®®à¯ à®†à®™à¯à®•à®¿à®²
à®®à¯Šà®´à®¿à®¯à¯à®¿à®²à¯à®®à¯, à®²à®¤à¯à®¤à¯€à®©à¯
à®®à¯Šà®´à®¿à®¯à¯à®¿à®²à¯à®®à¯, à®¹à®¿à®¨à¯à¯à®¤à¯à¿
à®®à¯Šà®´à®¿à®¯à¯à®¿à®²à¯à®®à¯ à®•à®¿à®Ÿà¯ˆà®•à¯à®®à¯®à¯.

à®Žà®©à¯à®©à¯ˆà®ªà¯ à®ªà®±à¯à®±à®¿

This is what happens when a text file is saved with one encoding but opened with another. Back when the internet was young, this was a huge problem—people would use whatever encoding made sense locally to save their webpages, but others on the internet (possibly on the other side of the planet) might use a different encoding to open it. The result was garbled nonsense, and for a long time browsers included tools to try different encodings to see what actually worked.

Unification

- In 1988, three engineers—Joe Becker, Lee Collins, and Mark Davis—proposed a unifying encoding that would supersede all the existing character sets.
- They wanted an encoding that was:
  - universal (to every language)
  - uniform (fixed-width), and

  - unique (no ambiguity)
- They called this encoding UNICODE.
- In 1991, the Unicode Consortium was founded to develop this encoding further.
- In 1992, the Unicode 1.0 was released.

Info from: https://unicode.org/

Representing every Character

- Unicode defines hundreds of thousands of characters.
- Eventually, Unicode wants to encode *every* character used for human writing.
- Each character is assigned a CODE POINT, a number uniquely identifying it.
- Related characters (e.g., letters in the same alphabet) are grouped together into "code planes".
- To uniquely represent all of these characters using a fixed-width encoding, each character would need a lot of bits… but this would be inefficient.

UTF-16 and UTF-32

- Unicode used to have fewer than $2^{16} = 65356$ characters in it (i.e., it had a single "plane").
- At that point, it made sense to use a fixed 16-bit representation for each code point.
- This representation is called the 16-bit Unicode Transformation Format, or UTF-16.
- After Unicode 3.0, there were far too many code points to fit in 16 bits.
- The logical next step, UTF-32, was considered wasteful (every code point would then take 4 bytes).

UTF-16 and UTF-32 may be big-endian or little-endian, adding even more chaos into this.

Ken Thompson Saves the Day: UTF-8

- Ken Thompson (the UNIX guy) and Rob Pike developed a new variable-length encoding for Unicode called UTF-8.
- Under this encoding, more frequently used symbols (like the Latin alphabet) get represented by shorter sequences, while less frequently used ones (like hieroglyphics) get longer sequences.
- Each character is now represented by both a CODE POINT and a UTF-8 BYTE SEQUENCE.
- This encoding is the most popular encoding today, and used on all major operating systems.

There's no concept of endianness for UTF-8 by definition; there is only one correct order for any sequence of bytes.

Line Endings

- Since UTF-8 is backwards-compatible with ASCII, it inherits an issue from ASCII: line endings.
- ASCII was originally designed to be compatible with typewriters.
- On typewriters "carriage return" (`0x0D`) moves the paper to the right, while "line feed" (`0x0A`) moves the paper up.
- MS-DOS (and later Windows) preserved this exactly: a newline was `0x0D 0x0A`.
- Unix decided to save a byte: a newline was just `0x0A`.

- This distinction exists to this day; some people standardized on "DOS line endings" (the web, email, etc.) and some on "UNIX line endings" (git, compilers, etc.).

Generally, internet protocols tend to prefer DOS line endings while command-line tools prefer UNIX line endings. Internet *documents* themselves might use either one, but the protocols (e.g., the SMTP email headers you set in assignment 6) mandate DOS endings.

4 Audio Encoding

Audio

- Just like text, there are many ways to encode audio.
- At the most basic level, audio is an array of "loudness" over time.
- Each array element corresponds to some amount of time, determined by the SAMPLE RATE. A common sample rate for audio is 44100 samples per second (44.1 kHz).
- Each array element has a certain size, corresponding to its accuracy. Eight-bit audio sounds worse than 16-bit audio.
- As most humans have two ears, we tend to prefer STEREO audio, where there are left and right CHANNELS; each channel is a separate array.
- When these arrays are stored raw, they're called PULSE-CODE MODULATION files (or PCM) files. It's common to store these in files with a `.wav` file extension.

Audio Codecs

- PCM data takes up a lot of space, and audio is usually predictable, so there are a lot of encoders/decoders (codecs) to store audio more efficiently:
  - MPEG-1 AUDIO LAYER III (MP3)
  - FREE LOSSLESS AUDIO CODEC (FLAC)
  - ADVANCED AUDIO CODING (AAC)
  - OPUS
- Some of these codecs are LOSSY, meaning they discard some data they consider unnecessary from the original audio.
- Some of these codecs are LOSSLESS, meaning they can be reversed 100% accurately to the original audio.

Lossy protocols provide smaller files, at the cost of some audio quality. Lossless protocols provide the best audio quality, at the cost of larger files. Which one is best depends on the particular use case. Note that most lossy protocols do let you configure the *amount* of loss, so you can use them to find a middle ground if you so desire. However, if you reencode an audio file through lossy encodings enough times, you will eventually end up with an inaudible file.

Audio Containers

- While a codec determines how raw audio data is represented as numbers, a CONTAINER format determines how that encoded audio is saved on disk.
- While some containers and codecs are often found together, it's generally possible to mix-and-match a codec and a container format.
- The container stores information like song titles and artist names.

- The container may include multiple channels of audio.

5 Image Encoding

Image Formats

- Images are slightly simpler than audio, since each container format generally contains a specific codec.
- There are two kinds of images: raster and vector.
- Raster images are essentially a grid of pixels, each of which has a specific color.
  - Photographs
- Vector images are a set of instructions to draw an image.
  - Drawings, Logos, Diagrams
- Vector images can be rendered at any size on-demand, while raster images have a fixed size.

Vector images are also more efficiently stored than raster images. In general, it's a good idea to use vector images whenever possible. The main case where raster images are useful is for photographs, since cameras can only produce raster images. Raster images may also be useful for speed, since vector images need to be converted to raster before they can be displayed on a computer monitor.

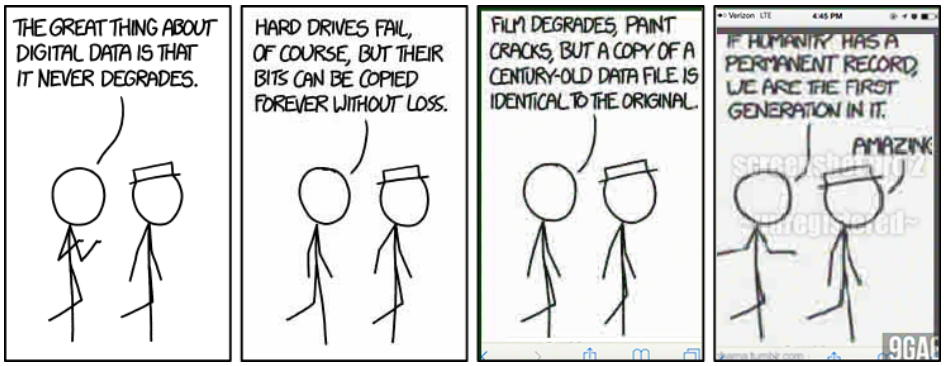
Raster Formats

Common raster image formats include:

- Joint Photographic Experts Group (JPEG)
- Graphics Interchange Format (GIF)
- Portable Network Graphics (PNG)
- High Efficiency Image File Format (HEIF/AVIF)
  - High Efficiency Video Coding (HEVC)
  - AOMedia Video 1 (AV1)
- Tagged Image File Format (TIFF)
- Windows Bitmap (BMP)

Once again, there is a lossy vs. lossless distinction.

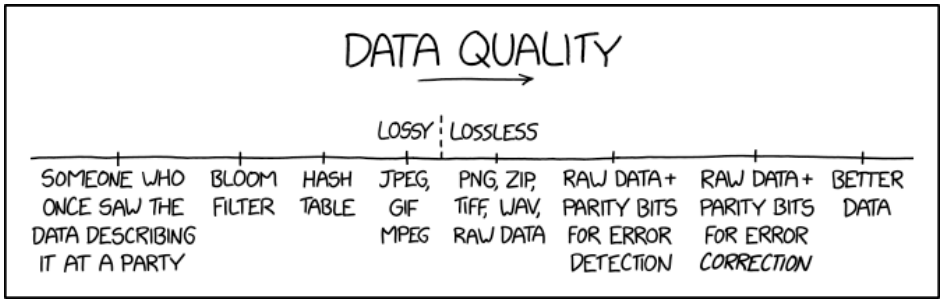
Lossy Compression



Source: xkcd 1683

And, while we're on the topic:

Data Quality



Source: xkcd 2739

Vector Formats

Common vector image formats include:

- Scalable Vector Graphics (SVG)
- Gerber (for printed circuit board designs)

Vector formats are also common for 3D objects due to their relatively small size:

- Wavefront OBJ
- Blender
- various Autodesk formats

Raw Images

- High-end cameras can "shoot in raw", which captures the raw pixel data coming from the image sensor.
- These images contain ridiculous amounts of data, most of which isn't useful.
- Photographers need to "develop" RAW images into a final image.

Color Spaces

- While light forms a continuous spectrum of colors, humans can usually only see the intensities of three wavelengths.
- These wavelengths roughly correspond to red, green, and blue, so we can try to parametrize a color by how much red, green, and blue (RGB) need to be mixed to make it.
- The exact way human brains reconstruct colors from these three wavelengths is complicated, so there are various COLOR SPACES which attempt to approximate it.
- There are other ways to parametrize colors; for example, hue, saturation, and value/brightness (HSV).

A particular image format may require a specific color representation, or it may support multiple color spaces and require the software reading it to translate as appropriate.

Even More Options

- As if different color spaces weren't bad enough, there are different representations of each color space.
- RGB (red, green, blue) is pretty common, but sometimes BGR (blue, green, red) or other permutations are also used.
- Sometimes you'll see RGBA or ARGB ("A" stands for "alpha", which is transparency).
- RGB888 uses 8 bits per color (24 bits total), RGB565 uses 5 or 6 per color (16 bits total), RGB332 uses 2 or 3 (8 total).
- 24-bit color is called "true color", 16-bit color is called "high color", and anything higher than 24 is called "deep color".

High Dynamic Range

- Human eyes have a very high "dynamic range"; they can resolve detail even when some parts of an image are very bright and some are very dark.
- Cameras don't have this, so they fake it by taking photos with different exposure settings and merging them together.
- Only some codecs support HDR, including HEIC and AVIF.

Resolution

The "size" of an image in pixels is called its RESOLUTION.

Common resolutions include:

HD 1280×780

Full HD 1280×1080

Full HD 1920×1080

4K Ultra HD 3840×2160

8K Ultra HD 7680×4320

Square power-of-two resolutions are also common (e.g., 16×16, 32×32, 2048×2048), especially for logos.

6 Video Encoding

Video Codecs

Like audio, there are several different video codecs in use:

- High Efficiency Video Coding (H.265)
- Advanced Video Coding (H.264)
- VP8 and VP9
- Theora

Uncompressed video is almost entirely useless outside of professional use, since the files are ridiculously large. Video is compressed using lossy algorithms.

Video Containers

There are also several container formats:

- Matroska (MKV)
- MPEG-4 (MP4)
- QuickTime (MOV)
- Audio Video Interleave (AVI)
- WebM
- Ogg

Video containers also double as audio containers, since most videos have associated audio.

Frame Rate

In addition to all the parameters from both images and audio, video also has an additional parameter: the frame rate.

The frame rate of a video controls how many images are shown per second.

High frame rates look smoother but require more data.

Typical frame rates are 24, 30, and 60 frames per second.

Reencoding Video

- In order to do almost any operation to a video, it needs to be reencoded.
- Reencoding a video is slow; usually it's about 1:1 with the length of the video.
- Reencoding a 90-minute lecture video takes about 90 minutes on my laptop.
- Reencoding also effectively recompresses a video, losing information every time if the encoding is lossy.

7 Conclusion

Useful media tools

`file`**:** identifies the format of many files

`iconv`**:** converts between text encodings

FFmpeg: converting (or identifying) audio and video

ImageMagick: editing images

Pandoc: converts documents between formats

Review

- There are lots of ways to encode any given piece of media.
- The appropriate encoding for a specific purpose depends on a lot of factors.
- Different people/software/systems may expect different encodings, and run into problems when given data in the wrong encoding.

- It's important to keep encoding in mind when working with media, and generally not mix together things with different encodings.

Assignment 0: Getting Set Up!

Due April 12th, 2023 at 11:59pm // [Gradescope]

Table of Contents

Overview

This assignment consists of two different components:

1. **You'll set up a shell environment on your personal computer**
2. **You'll run a shell command to make sure your shell is working**

We expect this assignment to take less than an hour. If you find yourself unproductively stuck, ask on Ed and/or go to office hours!

Software Installation

Follow the instructions labelled "Setting Up Your Shell Environment" (the first section) on our Software Installation page to set up your computer with the software we'll be using this quarter. The instructions vary by OS; make sure you're following the instructions for Windows, macOS, or Linux, depending on what you have installed.

If you're unable to follow the instructions (e.g., you have a Chromebook or a very old computer, or you don't have access to a personal computer), let us know—we may be able to help.

Testing Your Setup

Open the terminal program on your computer, and run the command `uname -a` at the prompt. Copy-and-paste the output into Gradescope.

Note that you may not be able to copy from a terminal with `control-c` on Windows or Linux. Try using `control-shift-c` instead, or right-clicking with the mouse and selecting "copy" from the menu.

If you're curious, the `uname` command prints information about your system, like what OS you're running and what CPU your computer has. The `-a` flag (short for `--all`) tells `uname` to print all available information.

Software for Lectures

On the same page as above, we also provide instructions to install the specific software we'll be using during lectures. If you want to follow along with us (which we highly recommend!), you'll want to install those pieces of software before the respective lecture.

Submitting Your Assignment

Submit your answer for "Testing Your Setup" on Gradescope.

Software & Computer Setup

> *We've detected that you're using a Linux computer. Click here if you're using Mac or Windows. The instructions on this page will be tailored to the detected system.*

Setting Up Your Shell Environment

If you're running linux, you're already all good to go! Note that the install instructions below assume you're running some flavor of Ubuntu or Debian; if you use a different distribution, replace `apt-get install` with the appropriate installation command for the package manager your distribution uses (e.g. `pacman` or `yum`)

Software for the Class

Lecture 2: The Shell and Shell Tools

Follow the shell setup instructions and you'll be good to go.

I demoed a non-standard command called `bat` during the lecture. This can optionally be installed using:

```
apt install bat
```

We'll add more here as we go through the quarter; be sure to check back before each lecture!

Lecture 5: Text Editors

You'll want to install the `vim` program to fully follow along:

```
sudo apt-get install vim
```

Lecture 6: Command-Line Environment

You'll want to install the `tmux` program to fully follow along:

```
sudo apt-get install tmux
```

Lecture 8: Computer Networking

You'll want to install `python3`, `node`, `ngrok`, `dig` and optionally Wireshark to fully follow along. You'll also need to sign up for an `ngrok` account.

On Linux, you may also need to install `traceroute` and `dig`:

```
sudo apt-get install inetutils-traceroute dnsutils python3 wireshark curl dnsutils
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash
```

...then restart your terminal, and run...

```
nvm install node

curl -s https://ngrok-agent.s3.amazonaws.com/ngrok.asc | sudo tee /etc/apt/trusted.gpg.d/ngrok.asc >/dev/null
```

Lectures 9 and 10: Version Control

You'll want to install the `git` and `gh` programs to fully follow along:

Many distributions ship with `git` already, but if it's not installed you can install it via a package manager:

```
sudo apt-get install git
```

To install `gh`, follow the steps from the official installation instructions

You'll also want to sign up for a GitHub account at github.com/signup

Lecture 15: Virtual Machines & Containers

You'll want to install a virtual machine hypervisor for your platform, and Docker Desktop. You'll also need to grab a copy of an Ubuntu Server disk image, which you can download from here. **If you're on an M1 Mac, please make sure to download the "Ubuntu Server for ARM" verison from here**.

Download Virtualbox from here.

And finally, download and install Docker Desktop from here.

Lecture 16: Cloud & Serverless Computing

You'll need to sign up for an account with Vercel and AWS.

Sign up or log in to your Vercel account, and sign up or log in to your AWS account.

(Note that you will need to provide a payment method to AWS in order to complete sign-up. You will not be charged. Let us know if this presents you with any issue!)

Lecture 17: Media Encoding

It might be useful to install `ffmpeg` (which also comes with `ffprobe`) for Assignment 8.

```
sudo apt install ffmpeg
```

Software & Computer Setup

We've detected that you're using a Windows computer. Click here if you're using Mac or Linux. The instructions on this page will be tailored to the detected system.

Setting Up Your Shell Environment

Installing WSL

Windows Subsystem for Linux, or WSL, allows you to run a traditional Unix shell on your Windows machine. We'll do our best to show off Windows-compatible and Windows-native software as we go through, but many useful and/or necessary tools used when doing software developemnt are only available via a Unix shell. (This is a problem Microsoft realized, and WSL is their answer to it!)

To install WSL, ensure you're running a recent version of Windows 10 or Windows 11. Then, right-click the start menu, and click **Command Prompt: Administrator** or **Windows Powershell: Administrator**. Press "Yes" when prompted. Finally, type in the following command:

```
wsl --install -d Ubuntu
```

Wait for it to finish. Once you're done, you should find a new application on your computer entitled **Ubuntu** – launch this to get a Unix shell. All the install instructions below will assume you're working within a Unix shell.

Software for the Class

Lecture 2: The Shell and Shell Tools

Follow the shell setup instructions and you'll be good to go.

I demoed a non-standard command called `bat` during the lecture. This can optionally be installed using:

```
apt install bat
```

We'll add more here as we go through the quarter; be sure to check back before each lecture!

Lecture 5: Text Editors

You'll want to install the `vim` program to fully follow along:

```
sudo apt-get install vim
```

Lecture 6: Command-Line Environment

You'll want to install the `tmux` program to fully follow along:

```
sudo apt-get install tmux
```

Lecture 8: Computer Networking

You'll want to install `python3`, `node`, `ngrok`, `dig` and optionally Wireshark to fully follow along. You'll also need to sign up for an `ngrok` account.

```
sudo apt-get install curl dnsutils
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash
```

...then restart your terminal, and run...

```
nvm install node
```

```
curl -s https://ngrok-agent.s3.amazonaws.com/ngrok.asc | sudo tee /etc/apt/trusted.gpg.d/ngrok.asc >/dev/null
```

You can download Wireshark here.

Lectures 9 and 10: Version Control

You'll want to install the `git` and `gh` programs to fully follow along:

Many distributions ship with `git` already, but if it's not installed you can install it via a package manager:

```
sudo apt-get install git
```

To install `gh`, follow the steps from the official installation instructions

You'll also want to sign up for a GitHub account at github.com/signup

Lecture 15: Virtual Machines & Containers

You'll want to install a virtual machine hypervisor for your platform, and Docker Desktop. You'll also need to grab a copy of an Ubuntu Server disk image, which you can download from here. **If you're on an M1 Mac, please make sure to download the "Ubuntu Server for ARM" verison from here**.

Download Virtualbox from here.

And finally, download and install Docker Desktop from here.

Lecture 16: Cloud & Serverless Computing

You'll need to sign up for an account with Vercel and AWS.

Sign up or log in to your Vercel account, and sign up or log in to your AWS account.

(Note that you will need to provide a payment method to AWS in order to complete sign-up. You will not be charged. Let us know if this presents you with any issue!)

Lecture 17: Media Encoding

It might be useful to install `ffmpeg` (which also comes with `ffprobe`) for Assignment 8.

```
sudo apt install ffmpeg
```

Assignment 1: *Shell* We?

Due April 19th, 2023 at 11:59pm // [Gradescope]

Table of Contents

Overview

This assignment consists of two different components:

1. **You'll familiarize yourself with RegEx by completing a few exercises**
2. **You'll combine your RegEx and shell expertise to wrangle some words**

We expect this assignment to take 1-3 hours depending on your proficiency level with the tools. If you find yourself unproductively stuck or unproductively struggling (spinning in circles), ask on Ed and/or go to office hours!

As a reminder, you can use `man` pages to learn more about commands you don't yet know (especially what flags they support). The website cheat.sh also has many examples of how to use different commands.

Learning Goals

We think it's important that our assignments exercise the tools and concepts we're learning in class. In general, lectures will provide a conceptual framework for understanding these systems, while assignments will give examples of the tools used in practice. Here's what you can expect to take away from this assignment:

- You'll get practice with designing regular expressions to match interesting patterns in text.
- You'll get practice using shell tools and controlling their behaviors with flags and arguments.
- You'll gain a significant amount of experience in using the shell, including:
    - calling different programs to perform specific tasks
    - chaining programs together with pipelines
- You'll grow your ability to discover information about tools and utilities. This is really important, because in the "real world" (a job or another class), this will be one of the major ways you self-teach yourself whatever tools are relevant to the task at hand.

Part I: Write Simple Regular Expressions (1.5 points)

For this part of the assignment, you will get some practice writing simple regular expressions. You should complete up through Lesson 10 in the regular expressions tutorial found here.

Place all of your answers in a simple text file called `regex.txt`. Remember that you can make a new text file using the `touch` command. You can open a file by using the `open` command (on macOS), the `explorer.exe` command (on Windows) or the `xdg-open` command (on Linux). The answer for each lesson (including lesson 1½) should go on a new line inside of `regex.txt`. *Note that there are many possible correct answers to each exercise!*

Important: Make sure you name your file `regex.txt` and not something else. Make sure that your file is exactly 11 lines long, one line per lesson.

Part II: Write Data Wrangling Pipelines (1.5 points)

For this part of the assignment, you will get some practice with `sed` and `grep` by writing commands to parse data inside a dictionary file we provide, which is a newline-delimited list of English words. Each exercise in this part can be solved multiple different ways (as is often the case with data wrangling!). In particular, many can be solved with either `sed` or `grep`, since there's a lot of overlap in what they do; for this assignment, feel free to use either tool.

To begin, use the command below to download a dictionary file. (Different machines have different dictionaries installed, so we want to make sure everyone is using the same one):

```
curl -Lo dictionary.txt https://cs45.stanford.edu/res/assign1/dictionary.txt
```

You can run `cat dictionary.txt` to see what the dictionary of words looks like.

As a reminder, the regex debugger at regex101 is really helpful! We'd recommend making use of it while you develop your regexes, to make sure it's doing what you think it is.

Subpart 1: Three O's

First, find the number of words in `dictionary.txt` that have at least three `o`'s. Take a look at the man page of `grep` or `sed` for possible ways to do this.

Subpart 2: Separated O's

Next, find the number of words in `dictionary.txt` that have **exactly** three `o`'s *and* where all `o`'s are separated by at least one non-`o` character. In other words, you should match a word such as `microbiology` (as all three `o`'s have at least one non-`o` character separating them) but you should not match a word such as `zootrophy` (as there are two adjacent `o`'s that are not separated by a non-`o` character). This exercise is a little tricky as you need to consider that some words begin with an uppercase `O` and regexes are case sensitive by default.

Subpart 3: Common Suffixes

Finally, find the three most common last three letters for words in `dictionary.txt` that have two adjacent `o`'s. We recommend doing this in two steps:

1. Find a list of all words in `dictionary.txt` that have two adjacent `o`'s. (The words can have more than two o's as long as at least two of them are adjacent).
2. Then find the three most common final three letters of these words. (Hint: take a look at how you might use `grep` to strip off the final three letters and how you might use `sort` and `uniq` to find the most frequent occurrences for a given set of data).

To submit your answers, create a file named `words_commands.txt` and place the *commands you used to generate your answers* for each subpart on a new line. *This file should be exactly three lines long.* Each line should contain a single command pipeline that, if typed in at the shell, would output the requested data.

Feedback Survey (0.5 points)

Once you have completed the assignment, you can earn an additional 0.5 points by completing our anonymous feedback survey. Given this is the first offering of the course, we want to collect as much feedback as possible to improve the course in the future. You can complete the survey here.

Once you complete the survey, you will receive a completion code which you should place in a text file named `survey.txt`. The survey is anonymous so submitting the completion code is the only way to verify that you completed the survey. *Please do not share this code with anyone, as that would constitute a breach of the honor code.*

Submitting Your Assignment

Once you have finished this assignment, you will need to upload your files to Gradescope. Make sure to upload both files to the Assignment 1 submission page. You should also upload `survey.txt` if you completed the survey.

Here are the files we're expecting:

- `regex.txt`: eleven lines, each of which contains a regular expression
- `words_commands.txt`: three lines, each of which contains a valid command pipeline
- (optional) `survey.txt`: one line, which is the survey code

Gradescope will autograde your submission and let you know if your solutions worked. You may resubmit as many times as you'd like before the deadline.

All files must have the same names as specified above.

Assignment 2: Bourne to Be Wild

Due April 26th, 2023 at 11:59pm // [Gradescope]

Table of Contents

Overview

This assignment consists of two different components:

1. **You'll write a simple shell script that does some data analysis**
2. **You'll get some practice using vim**

Before starting the assignment, you will want to ensure you've installed all the software required for lectures 1 through 5.

We expect this assignment to take 1-3 hours depending on your proficiency level with the tools. If you find yourself unproductively stuck or unproductively struggling (spinning in circles), ask on Ed and/or go to office hours!

As a reminder, you can use `man` pages to learn more about commands you don't yet know (especially what flags they support). The website cheat.sh also has many examples of how to use different commands.

Part I: Write A Shell Script (2 points)

For this part of the assignment, you will write a shell script which does some data wrangling. You'll be analyzing an input file containing information about Stanford CS Faculty. To get that file, use the command below:

```
curl -Lo cs_faculty.csv https://cs45.stanford.edu/res/assign2/cs_faculty.csv
```

This file contains faculty members' names, titles, and research groups. For example, here's a line from the file:

```
Alex Aiken, Professor, Computer Systems + Programming Systems and Verification
```

As you can see, the different columns are separated by commas. This is what's called a "comma-separated values" file, or a CSV file for short. The last column contains the professor's research groups, each separated by a "+" (plus sign). For the purposes of this assignment, you can treat this as an ordinary text file.

Your task is to write a shell script that analyzes this file. As a reminder, a shell script is a text file containing shell commands, which might look like this:

```
#!/usr/bin/env bash
name="world"
echo "hello $name"
echo "you can run other shell commands here"
```

Each line of the shell script is run as if you typed it at the shell, except for comments (lines beginning with "#").

Create a shell script called `analyze.sh`, and download `cs_faculty.csv` into the same directory. You can run your script by running `sh analyze.sh`. In the shell script, add commands which do the following:

1. Make a new subdirectory called `analysis`. You might run the shell script multiple times, which means the directory may already exist; make sure your script works whether or not the directory already exists (hint: you might need to pass a flag to `mkdir`).

2. Delete any files inside `analysis`, if there are any. Once again, make sure your script works whether or not there are files inside the subdirectory.

3. For each of the following research groups, create a file in the `analysis` directory with the information of every professor who researches that topic. Make sure the information is in the same format as the original file. (In other words, if you find a faculty member who researches a topic, you should include the *entire* line with all of that faculty member's information in the research group specific file). Make sure each file has the following specified name, and is within the `analysis` subdirectory.

    1. "Artificial Intelligence" => `ai.csv`
    2. "Computational Biology" => `bio.csv`
    3. "Computer Graphics" => `graphics.csv`
    4. "Computer Security" => `security.csv`
    5. "Computer Systems" => `systems.csv`
    6. "Human-Computer Interaction (HCI)" => `hci.csv`
    7. "Theory" => `theory.csv`

> *One hint here is that it actually does not matter that the data is in CSV format, and we strongly recommend that you do not parse the data into separate columns. Instead, we recommend treating each line of data as a single unit and searching for the research group of interest on that line. (Hint: think about how you might use `grep` to search for a given string of text within each line assuming that you treat the line as a single unit.)*

4. Create a file named `big_groups.txt` in the `analysis` directory with the names of every research group with more than 10 members. Think about how you might use the files you created in Step 3 in order to make this easier. Make sure to output the name of the *group* (e.g., "Artificial Intelligence"), not the name of the *file*.

5. Create a file `analysis/systems-ai-join.csv` with the information of every professor who researches both "Artificial Intelligence" and "Computer Systems". Remember that they may also research other things.

    1. There's a few different ways to do this, but here's a hint for one of the ways: the command `uniq` (which normally removes duplicate lines from a file) can be used to print *only* duplicate lines with a certain flag.
    2. Note that `uniq` requires its input to be sorted.

6. Create a new file `analysis/names.txt`, with the names of every faculty member sorted alphabetically. Make sure this file only contains their names, not their title or research groups. You may want to use the `cut` command, although there's a few others that would work too.

You're free to output whatever you want (with `echo`) to help you debug. You might want to add the command `set -x` at the beginning of the script; this makes the script print out each command as it runs it.

Make sure your shell script has the name specified above (`analyze.sh`), and creates files with the correct names.

Part II: Try vim out! (1 point)

For this part of the assignment, you'll try `vim` out by editing a file on a remote computer (ooo!) – the computer you will be editing on is the same one that we checked out some logs from during the Data Wrangling lecture. You'll be fixing up a Python file with a few errors—but don't worry if you're rusty with Python, the comments will instruct you what to do and what to write.

Configuring vim

Let's make your vim a little easier to use, more helpful, and fun :)

The first step is to create a `.vimrc` file if you don't already have an existing one. To check if you have a `.vimrc` file (you may have configured one in the past!), you should run the following: `cat ~/.vimrc`. If you get an error message that reads `No such file or directory`, then you don't have a `.vimrc` file and will need to create one.

If your `.vimrc` file is blank, you might consider using ours! You can read it right in `vim` with:

```
vim https://web.stanford.edu/class/cs45/res/lec5/.vimrc
```

You can copy it over right to your home directory (note this will overwrite an existing configuration):

```
curl -Lo ~/.vimrc https://web.stanford.edu/class/cs45/res/lec5/.vimrc
```

To create or edit a `.vimrc` file, you should run `vim ~/.vimrc`. This will allow you to open your `.vimrc` file with... `vim`!

We recommend the following customizations (which are included in our `.vimrc` file already):

- Enable line numbering
- Enable syntax highlighting

- Enable mouse usage
- Enable "hidden" buffers (opening multiple files without having them all on-screen at the same time; see :help hidden for more info)
- Enable filetype detection, plugins, and indentation (see :help :filetype-overview for more info)

You are more than welcome to add other customizations to your `.vimrc` file, but we recommend at least using the ones above.

If you submit your `.vimrc` file, our autograder will automatically check it to make sure you enabled the above customizations, but it's not required.

> Editing a Python file

To access the file, copy and paste the lines below into your command line– (that's right, you can copy/paste multiple commands at once!). You should change `<SUNET>` to **your SUNet ID,** (which is the part of your Stanford email before @stanford.edu).

```
export SUNET=<SUNET>
vim sftp://s-cs45-assign2-$SUNET@192.9.152.85/exercise.py
```

For example, if your SUNet ID was `example`, your commands would look like:

```
export SUNET=example
vim sftp://s-cs45-assign2-$SUNET@192.9.152.85/exercise.py
```

You may have to press ENTER once when prompted to see the file. If you have any issues accessing the file, let us know in an Ed private post!

Protip: before saving with `:wq`, go into normal mode and enter the command `:retab` to make sure all your whitespace is consistent– Python requires this.

When you submit your assignment, the autograder will automatically download your file and grade it. You don't need to submit it with the rest of your files.

If you decide to resubmit this portion of the assignment, you will need to re-run the autograder.

Feedback Survey (0.5 points)

Once you have completed the assignment, you can earn an additional 0.5 points by completing our anonymous feedback survey. Given this is the first offering of the course, we want to collect as much feedback as possible to improve the course in the future. You can complete the survey here.

Once you complete the survey, you will receive a completion code which you should place in a text file named `survey.txt`. The survey is anonymous so submitting the completion code is the only way to verify that you completed the survey. *Please do not share this code with anyone, as that would constitute a breach of the honor code.*

Submitting Your Assignment

Once you have finished this assignment, you will need to upload your files to Gradescope. Make sure to upload all files to the Assignment 2 submission page. You should also upload survey.txt if you completed the survey.

The only files you need to submit to Gradescope is `analyze.sh` (and `survey.txt` if you completed the survey). If you'd like, you can also upload your `.vimrc` to verify that it contains the customizations we recommend, though this is not part of your grade and is not required. You don't need to worry about `exercise.py`, as it will be downloaded by the autograder automatically when you submit.

It may be difficult to upload your `.vimrc` file into Gradescope, as the file is hidden—but it's easy to add all these to a zip file!

```
# Run this command in your assignment directory:
zip -jv assign2_submission.zip ~/.vimrc ./analyze.sh ./survey.txt
```

Once you have created a zip file, you can upload it to Gradescope.

All files must have the same names as specified above.

Bonus: More on `grep` & `sed` (optional, but helpful extra practice!)

We got some feedback from the last assignment that it would be helpful to return to `grep` and `sed` for some additional practice. This exercise will emphasize the difference between these two utilities and will walk you through a couple different examples you can try.

Start by grabbing the files we'll use by copy-pasting the line below into your terminal and pressing enter:

```
for i in {1..10}; do curl -Lo emails_$i.txt https://cs45.stanford.edu/res/assign2/emails_$i.txt; done
```

Inside these text files, there are lists of email addresses. There is one email per line, each with a comma afterwards. We want to accomplish two goals: (1) Locate certain email addresses, and (2) replace the commas with semicolons (Outlook only accepts semicolon-delimited email addresses!).

(1) Locating certain email addresses

When your task is locating information, grep is your friend. There's a particular email address to locate: bighearted.piscatorial.americanbulldog@example.com, and it's in one of those ten email files. Now, you could use grep or sed on each one of them, searching for a file where a line matches, or you could...

```
grep -n 'bighearted.piscatorial.americanbulldog@example.com' *.txt
```

The line above uses **globbing** to find all the `.txt` files (`*.txt`), and give them to grep. (Note that while this looks like RegEx, the syntax is simplified– e.g. the above glob knows "any files ending in .txt." If you were using RegEx for this, you would probably write `^.*\.txt$`).

This will search through *all* the text files at once, and show you what file it found that match in, along with the line number (`-n`). Woah! I use this tactic all the time when searching for something in a bunch of files (for example, finding where I used a certain class or variable name in my code...)

(2) Replacing commas with semicolons

However, if your goal is to modify a file or text– not simply search– grep just won't quite cut it. Let's replace all the semicolons in each file:

```
sed -i .backup 's/,/;/' *.txt
```

This will **replace all the commas with semicolons** in all the .txt files it finds in the current directory, and back up the original versions to a `<filename>.backup` file.

Assignment 3: Off The Beaten `$PATH`

Due May 3rd, 2023 at 11:59pm // [Gradescope]

This assignment consists of three different components:

1. **You will modify your `$PATH` variable, shell prompt, and add aliases**
2. **You'll get some practice using the networking tools we learned about in Lecture 8**
3. **You'll run a small server that our grading machine will connect to**

We expect this assignment to take 1-3 hours depending on your proficiency level with the tools. If you find yourself unproductively stuck or unproductively struggling, ask on Ed and/or go to office hours!

Part I: Customizing Your Environment Variables (1 point)

In Lecture 4: Shell Scripting, we saw how to write a shell script and make it executable from anywhere on the computer. Imagine you have a shell script called `hello.sh` that simply prints `Hello`. You create the script inside of a folder called `CS45` on your Desktop. (The folder would thus have a path of `~/Desktop/CS45`.) Given the script is located inside of the `CS45` folder, you can only run it from within that folder. Let's change that!

To make `hello.sh` script executable from anywhere on your machine, you will want to move it somewhere that is recognized by your `$PATH` environment variable. Whenever you type a command in the shell prompt (e.g. the command `grep`), your computer searches every folder inside of your `$PATH` environment variable to see if any of those folders have an executable by the name of the command you typed in (e.g. an executable by the name `grep`). For example, here is a sample `$PATH` environment variable:

```
/Library/Frameworks/Python.framework/Versions/3.11/bin:/opt/local/bin:/opt/local/sbin:/opt/homebrew/bin:/opt/h
```

When you type in `grep`, your computer first searches inside of `/Library/Frameworks/Python.framework/Versions/3.11/bin` to see if it finds an executable by the name of grep. If it doesn't find it there, it would then search inside of `/opt/local/bin`. It would then search inside of `/opt/local/sbin`, and so forth.

To check what paths are currently set on your computer, you can run `echo $PATH` at the command line prompt.

Let's create a new `bin` folder that belongs to the user who is currently signed in (presumably you!). A `bin` folder is short for a binary folder, which, as the name suggests, contains binary files (executable files) for programs that you might want to run. This new `bin` folder will allow us to store any of the local binary files that should be accessible anywhere on the computer for the current user.

Step 1: Creating A Bin Folder

First, you will want to navigate to your home directory using `cd ~`. Once you are in your home directory, you will want to make a new folder called "bin" using `mkdir bin`. Next, you will want to enter this new directory using `cd`. We will need the absolute path of this directory for Step 3. To get the absolute path of your newly-created bin folder, you should run `pwd` inside `~/bin`.

Step 2: Finding Your Shell

Your next task is to find out what shell you are running. You can normally do this by running `ps -p $$`. Your output may look something like the following:

```
karel@karel-computer bin % ps -p $$
PID TTY TIME CMD
25466 ttys001 0:00.32 -zsh
```

Step 3: Adding the Path

Now that you have your local bin folder (`~/bin`) and your shell, you will need to update (or make!) the config file specific to your shell:

- If you are using a zsh shell, your config file will be `.zshrc`
- If you are using a bash shell, your config file will `.bashrc`
- If you are using a tcsh shell, your config file will be `.tcshrc`

To check if you already have an existing config file, you should navigate back to your home directory (`~`) and then run `ls -a` (which will list all hidden files, such as your configuration files). If you don't have the right configuration file for your shell, you can just

create the file using `touch <NAME_OF_FILE>` (i.e. `touch .zshrc` for a zsh shell).

Once you have your configuration file, you will want to open it and the following line, replacing `<NEW_PATH>` with the full path you discovered in Step 1 using `pwd`:

```
export PATH=$PATH:NEW_PATH
```

In other words, if your path from step 1 was `/Users/karel/bin`, then your line inside of your config file would read `export PATH=$PATH:/Users/karel/bin`

Congratulations! You've successfully added a new path to your environment. You should now test out your new powers by creating a script called `hello` (you can drop the `.sh` ending as we will be turning the script into a command). The script should simply `echo Hello` along with your name. Make sure it has a valid shebang line so your computer knows how to run it! You can create the script in any directory you choose. Once you have created your script, turn it into an executable by running `chmod +x hello`

Now you will want to move the script to `~/bin`. You can do so by running mv hello `~/bin/`. You should now be able to say hello to yourself at any time of day, from anywhere on the computer!

In addition to modifying your path variable, there are other useful configurations that we will guide you through. Let's work on implementing the following two configurations:

- Adding colors to your ls command
- Customizing your shell prompt

Adding Colors to ls

To add colors for `ls`, you will want to add an alias for `ls` that changes the standard `ls` command to an `ls` command with colors. The way to do this will depend on which shell you are using.

If you are on macOS, you should add the following two lines to your `.bashrc` or `.zshrc` file:

```
export CLICOLOR=1
alias ls='ls -G'
```

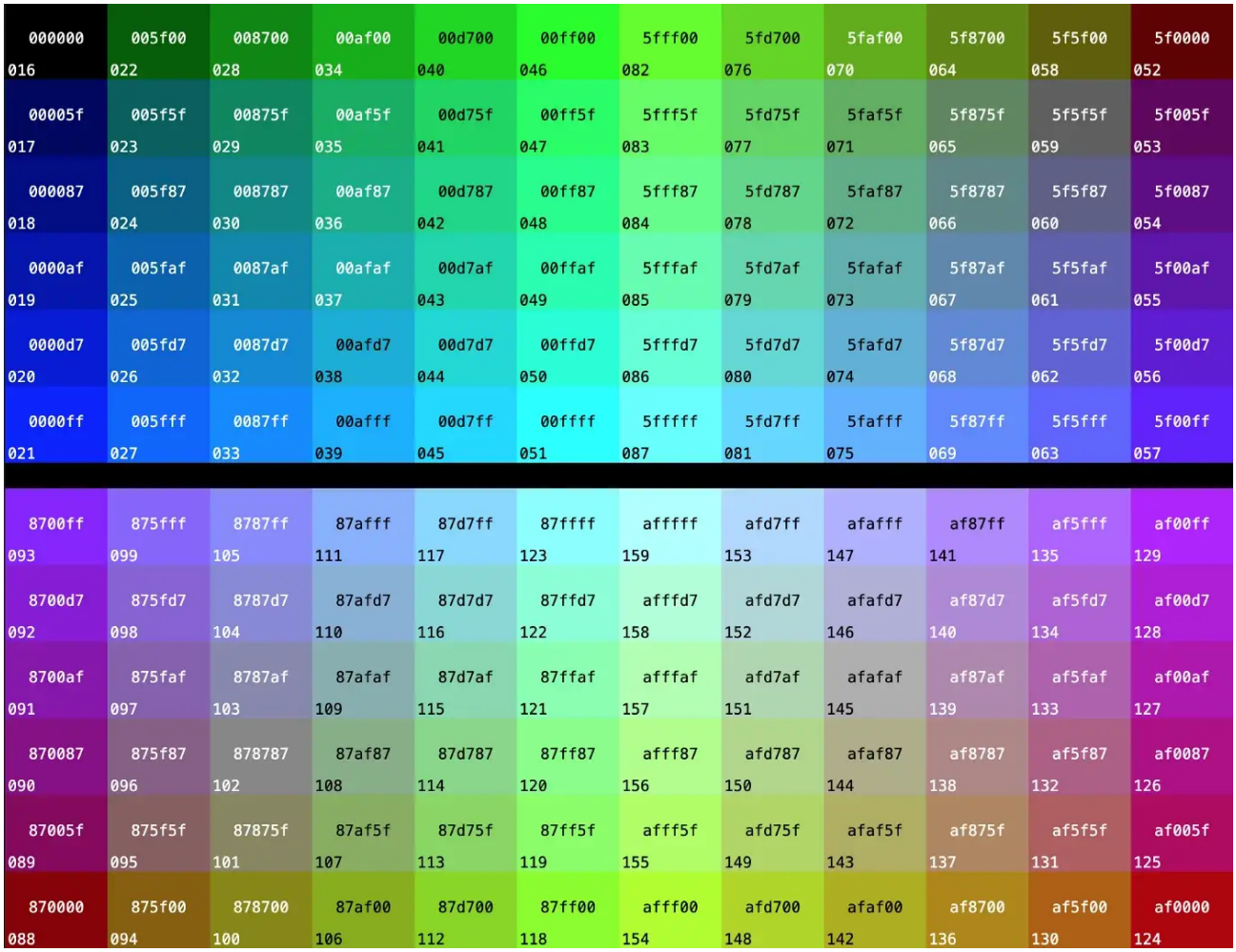
If you are using Linux or Windows (WSL), you should add the following line to your `.bashrc` or `.zshrc` file instead:

```
alias ls="ls --color=auto"
```

Customizing Your Shell Prompt

Now we will customize our shell prompt by adding colors to it and modifying what contents it has. Though we will leave it up to you as to what customization you would like to include, we have also provided a sample customization with CS45 themed-colors.

If you are using a zsh shell, you should use zsh guidelines for customizing your prompt. Here is a tool to build a zsh prompt. You can also use the chart below to choose zsh colors.



If you'd like to use our CS45-themed shell prompt, you should add the following line to your .zshrc file:

```
PROMPT='%B%F{75}%n%f@%b%F{88}%m%f:%~ %# '
```

If you are using a bash shell, you can also customize your shell. You will need to use bash specific syntax. You can easily customize your prompt using this tool. If you'd like to use our CS45-themed shell prompt, you should add the following line to your .bashrc file:

```
PS1="\[$(tput bold)\]\[\033[38;5;75m\]\u\[$(tput sgr0)\]@\[$(tput sgr0)\]\[\033[38;5;88m\]\h\[$(tput sgr0)\]:
```

For students who have access to the myth machines, we also recommend adding an alias for `ssh`-ing into the myth machines. (This part is not graded as not all students have access to the myth machines.) While you won't be able to have it automatically enter your password into `ssh` (for security reasons, `ssh` requires a human to type in the password), it'll at least reduce the tedium of typing `ssh $SUNET@myth.stanford.edu` over and over again.

Note: *If you are using another shell and we didn't include specific instructions here, reach out to us! We are happy to help.*

For this part of the assignment, you should submit your configuration file (e.g. `.bashrc`, `.zshrc`, etc).

Part II: Networking Short Answers (1 point)

In this part of the assignment, you'll be exploring some of the networking tools we learned about in Lecture 8 to get some information about your computer's networking environment. Make sure to install the software for Lecture 8! Note that for the commands below, if your computer uses Windows, you should use the Windows commands (even if you're inside WSL!)

(1) *Network Interfaces & IP Addresses*

To start, let's take a look at what network interfaces your computer has. On Windows, you can run the command `ipconfig.exe`, on macOS you can use `ifconfig`, and on Linux you can use `ip addr`. This will list all the network devices your computer has available!

For example, running this command on a Mac may output a network interface that looks as follows:

```
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    options=400<CHANNEL_IO>
    ether 00:31:4e:01:f5:bc
    inet6 fe80::3d:a0f4:5974:d420%en0 prefixlen 64 secured scopeid 0xc
    inet 10.36.40.71 netmask 0xfffff000 broadcast 10.36.47.255
    nd6 options=201<PERFORMNUD,DAD>
    media: autoselect
    status: active
```

The `en` designation indicates that this is an Ethernet (now Wi-Fi) interface. On Linux, you may see that a Ethernet/Wi-Fi interface is designated using `wlan`. Windows doesn't have a standardized set of short names for Wi-Fi interfaces. Instead, you should look for an interface that has the words "Wi-Fi" in it.

1.1 Using the command above, redirect its output into a file called interfaces.txt.

1.2 Look inside the output in `interfaces.txt`. Write the name of the interface that appears to be your wireless connection into a file called `wifi_interface.txt`

1.3 Look inside the output of `interfaces.txt`. Find and write your local IP address inside a file called `local_ip.txt`.

(2) *Routes*

Let's take a look at the routing table your computer is using. On Windows, you can run the command `route.exe print -4`, on macOS you can use `netstat -nrf inet`, and on Linux you can use `ip -4 route`

2.1 Using the command above, redirect its output into a file called `routes.txt`

2.2 Look inside the output in `routes.txt`. Find the default route (this is the address of the router that's connecting you to the rest of the internet!) and put its IP address in `default_route.txt`.

(3) Traceroute

Let's see what path it takes to get to a server hosted in another country (in this case, we'll be connecting to a website that gives information about a town in Japan)! On Windows, you can use the command `tracert.exe www.town.okutama.tokyo.jp`, on Linux you can use `traceroute -I --resolve-hostnames www.town.okutama.tokyo.jp`, and on macOS you can use `traceroute -I www.town.okutama.tokyo.jp`. Note that these commands may take a while to complete. If you have trouble with these commands, please let us know on Ed as soon as possible!

On some Linux machines, the `--resolve-hostnames` flag won't work and therefore you can't effectively examine the `traceroute` output. In that case, or if you run into other issues that make the traceroute output unusable, you can use our `traceroute` output.

*3.1 Using the commands above, pass them to a special program called `tee` which lets you redirect output to a file **and see it on your terminal at the same time!** The output of your command should go to a file called traceroute.txt.* For example:

```
tracert.exe www.town.okutama.tokyo.jp | tee traceroute.txt  # Windows
traceroute -I --resolve-hostnames www.town.okutama.tokyo.jp | tee traceroute.txt # Linux
traceroute -I www.town.okutama.tokyo.jp | tee traceroute.txt # Mac
```

3.2 How many hops did it take to get to the destination server? Put the number into a file `hops.txt`, or write "the traceroute didn't complete" if it wasn't able to find the destination within 64 hops.

3.3 Which hop number do you think was the last hop inside Stanford's campus? Put the number in `last_stanford_hop.txt`

3.4 Which hop do you think is the first server you see that's located in Japan (if any)? Place your answer and justification in `jump.txt`

Part III: Running a Small Server (1 point)

In Lecture 8: Introduction to Computer Networking, we learned all about how information travels from one computer to another. In this part of the assignment, you'll get some practice running your own development server over the network. Make sure to install the software for Lecture 8!

> *Note that this part of the assignment will expose parts of your computer to the internet. Please ensure that you follow the commands below in a **new, empty directory** to avoid exposing unwanted or private information.*

In a **new directory**, create a new file called sunet.txt that contains your SUNet ID (the part before your email!):

```
mkdir my_server_directory
cd my_server_directory
echo "Your SUNet Here" > sunet.txt
```

Then, you'll want to start your server. First, open a Python server as before:

```
python3 -m http.server --bind localhost 8080 &
```

We want to also publish the server to the internet using `ngrok`; the `&` at the end of the command above instructs the shell to **immediately place your server into the background** without needing to suspend it first. Neat!

Finally, let's publish it on ngrok. Make sure to install it first and follow the instructions to set up your account!

```
# Only do this once to log in:
ngrok config add-authtoken <your authtoken here>

# Start the server
ngrok http 8080
```

Open up a new terminal window to create a file called `server_url.txt` and copy/paste the URL that ngrok gives you into the file – it should end in `.ngrok.io` or `ngrok-free.app`. Then, submit it to Gradescope (keeping your computer open, `ngrok` running, and the Python HTTP server running)–our autograder will connect to your server and verify that your `sunet.txt` file matches your SUNet in Gradescope.

If you need to resubmit your assignment, make sure the server is running and the URL in `server_url.txt` is up-to-date—otherwise our autograder won't be able to connect to your computer.

Feedback Survey (0.5 points)

Once you have completed the assignment, you can earn an additional 0.5 points by completing our anonymous feedback survey. Given this is the first offering of the course, we want to collect as much feedback as possible to improve the course in the future. You can complete the survey here. Once you complete the survey, you will receive a completion code which you should place in a text file named survey.txt. The survey is anonymous so submitting the completion code is the only way to verify that you completed the survey. *Please do not share this code with anyone, as that would constitute a breach of the honor code.*

Submitting Your Assignment

Once you have finished this assignment, you will need to upload your files to Gradescope. Make sure to upload all files to the Assignment 3 submission page. You should also upload survey.txt if you completed the survey. For this assignment, **be sure to use the `zip` command below** to bundle all the files together; picking individual files through the Gradescope UI has been unreliable in the past.

You can submit all necessary files by running the following command, replacing `<CONFIG_FILE>` with the configuration file for your shell (e.g., `.bashrc` or `.zshrc`).

Run this command in your assignment directory:

```
zip -jv assign3_submission.zip ./server_url.txt ./survey.txt ./interfaces.txt ./wifi_interface.txt ./local_ip.
```

Once you have created a zip file, you can upload it to Gradescope. Make sure your server is running and available at the URL specified in the server_url.txt file while the autograder is running.

All files must have the same name as specified above.

Assignment 4: It's `git`-ting HOT in here!

Due May 10th, 2023 at 11:59pm // [Gradescope]

Table of Contents

Overview

This assignment consists of three different components:

1. Do a Git tutorial
2. Practice using Git from the command line
3. Practice using GitHub

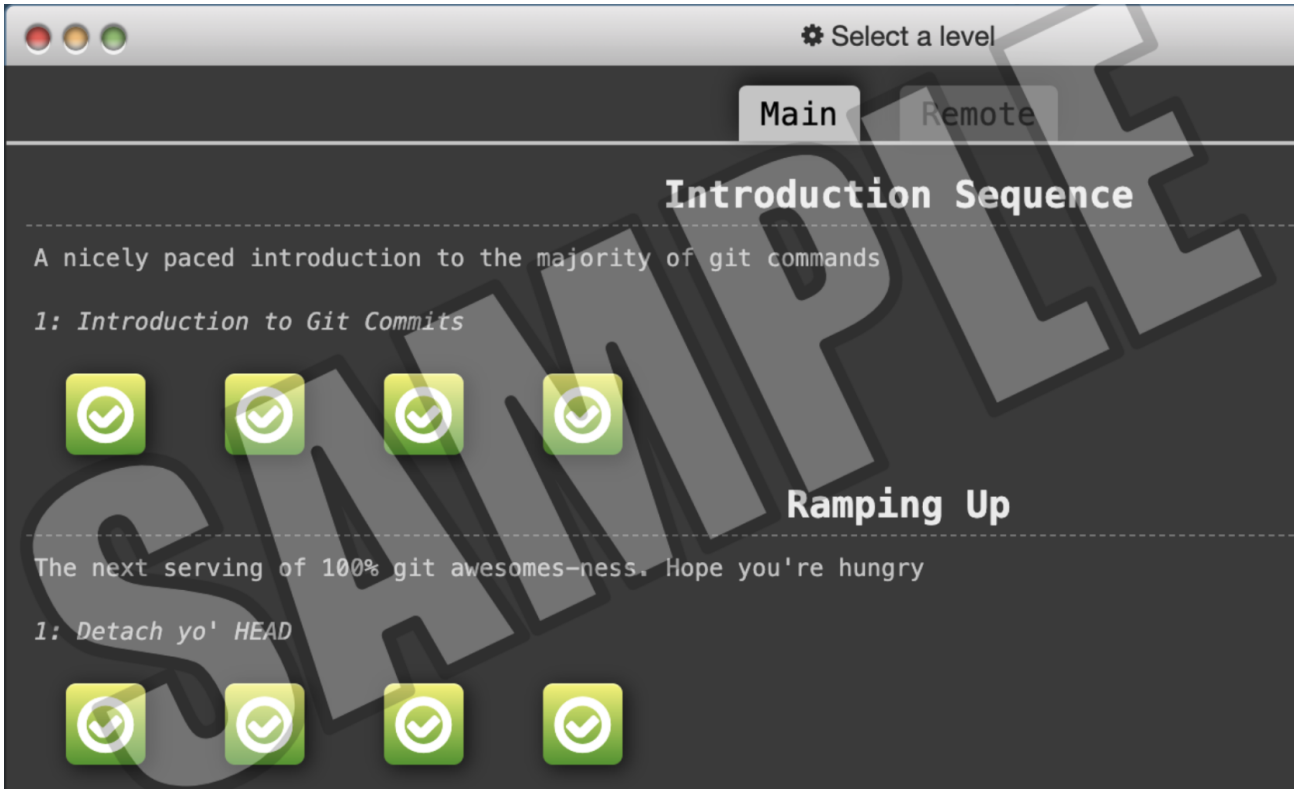
We expect this assignment to take 1-3 hours depending on your proficiency level with the tools. If you find yourself unproductively stuck or unproductively struggling, ask on Ed and/or go to office hours!

Part I: Master `git` (1 point)

In this part of the assignment, you will complete a tutorial to help you master `git`. The tutorial can be found here. You should complete the entire Introduction Sequence and Ramping Up sections*. Once you are done, submit a screenshot of your progress card.

**When submitting your screenshot, don't worry if the Autograder messes up– we'll go through everything manually and double check ;)*

**Note! You can do more if you want to, it's good practice! But we're only requiring the first two sections.*



Part II: Your First `git` Repo

Let's create your first git repository together—we'll be making a simple **rock paper scissors** game in Python, and using a `git` repository to track changes as we add features.

A `git` Repository is Born

The first step is to create a directory where you'll store your files. Go into your Terminal and create a new directory, then **initialize your git repository**.

Then, let's create our file– let's call it `rock_paper_scissors.py`. Here's what you want to do.

1. **While working on the game, commit your changes each time you finish a unit of functionality.** For example, a good first commit might be setting up a main function and greeting the user (your "initial commit.") Another one might be adding an explanation of the rules and how to play; etc. Remember, that your code must compile at each commit.
2. Your application should meet the following requirements:
    1. It should greet the user when it starts
    2. It should explain the rules of Rock Paper Scissors, which (for those of you who don't know) are as follows: two players secretly pick one of "rock," "paper," or "scissors." Both players reveal their selection to the other player at once; the winner is chosen based on what the selections are. Rock beats scissors (by crushing them); scissors beats paper (by cutting it); and paper beats rock (by covering it). If both players select the same one, it is a tie.
    3. It should allow the player to enter in their selection (you can use the `input` method). You can assume the user will type in `rock`, `paper`, or `scissors` (all lower-case).
    4. It should print out the computer's selection, which should be random (hint: you can use `import random` then do `random.choice(list)` to select a random element from a given list)
    5. It should declare a winner, or a tie.

Extending Our Game

Now that you've got a basic game working, let's extend it to make it just a bit fancier– when we tie, the computer should rematch us until one of us wins or loses! **To extend with our experimental new feature, let's create a new branch called `no-ties`!**

After creating this branch, implement your new feature. Feel free to commit as many times as makes sense (e.g. for a first pass implementation, and then again as you fix any bugs). Try to have at least two commits by the end of this.

Once you finish, **let's squash all your new commits into a single one**, then **fast-forward our `main` branch to include the new feature**. This will keep the history clean and linear, while still bringing in all our new work. For this, let's use the **interactive rebase** function on your `no-ties` branch. Keeping your first commit of your no-ties feature, tell git to squash all the ones that came after. Then, exit your `$EDITOR` and watch git do its magic! Once that's done, switch back to `main` and ask git to `merge --ff-only` your `no-ties` branch (which now only has a single commit!) into it.

Part III: Practice with GitHub (1 point)

For this part of the assignment, you will get practice with using GitHub in order to collaborate with other students in the class. We will be using this repository in order to collaborate together.

Student Hobbies

First, we want to compile a list of hobbies for all of the students in the course.

You can do this part from the GitHub online or desktop interface, then by cloning the directory locally.

If you are working locally: You will want to **fork** this repository, then **clone** it on your computer so you can make edits. Before you start making edits, you will want to pull any new changes from the remote repository on Github. Remember to create a GitHub account if you don't already have one, and log in on the shell with the `gh auth login` command.

You can fork the repository and clone it to your computer all in one go from the github CLI tool; simply do `gh repo fork stanford-cs45/spr23-assign4 --clone=true`

Once you have navigated to the repository, you should add a hobby to the list of hobbies found in `student_hobbies.txt` and include your SUNet in parentheses. You should also find a hobby that you share with someone else (i.e. a hobby someone else added that you also enjoy). For the hobby you share, you should add your SUNet in parentheses, separated by a single space.

Based on the number of people who share the hobby, move the line to the appropriate part of the file where the hobbies shared by the most people are at the bottom of the file and the hobbies shared by the fewest number of people are at the top of the file.

Once you are done, push these changes to your fork of the repository.

For example, the `student_hobbies.txt` file may look as follows when you first download it:

```
Surfing (adrazen)
Languages (akshay01)
```

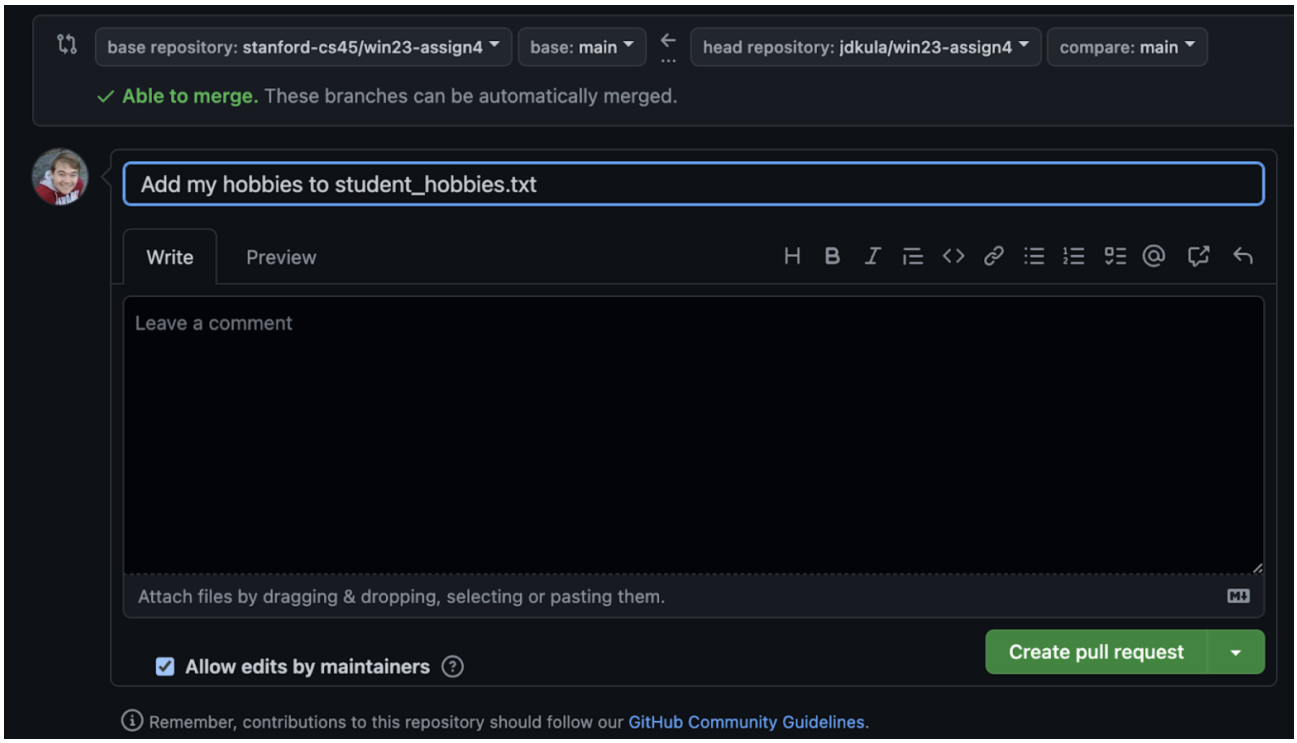
```
Drawing (jdkula)
```

Consider a student whose SUNet is `karel` who enjoys coding and also enjoys languages. After the student has completed this part of the assignment, the file would look as follows:

```
Surfing (adrazen)
Coding (karel)
Drawing (jdkula)
Languages (akshay01 karel)
```

^Note that "Languages" was re-ordered to the bottom. The file is sorted by number-of-people.

Finally, you'll be **submitting a pull request to the shared repository**. You can do this from the GitHub Website by going to your forked repository and clicking the "Contribute" then "Open Pull Request" buttons. You'll get a screen that looks like the following– fill out a title and a comment below, and then click "Create Pull Request"!



Here are some additional instructions from GitHub about creating pull requests, if useful.

Sleuthing for a Fact

We have a text file listing a bunch of facts called `facts.txt` on the repository— but oh no!! There are supposed to be 3090 fun facts in the file, but there are only 3089 currently. One must have been deleted at some point!

Your goal is to **find this missing fact** and **who deleted it**! To do this, you can use the `git log` and `git diff` commands to inspect the repository.

To start your sleuthing, you can *provide a file name* to git log to show only the commits that affected that file. This will give you the **commit hashes** of all commits that affect it. Do you see any strange or suspicious commit messages?

Once you identify a suspicious commit (or pair), you can use `git diff` to see what files changed between two commits. Recall from Part I, if you want to see what a single commit changed, you can compare a commit with the commit right before it. Recall that you can use `a1b2c3^` to refer to "the commit just before `a1b2c3`".

Once you've found the missing fun fact, put it in `missing_fact.txt` – then, place it in with your rock-paper-scissors game and commit it as an additional file.

Feedback Survey (0.5 points)

Once you have completed the assignment, you can earn an additional 0.5 points by completing our anonymous feedback survey. Given this is the first offering of the course, we want to collect as much feedback as possible to improve the course in the future. You can complete the survey here.

Once you complete the survey, you will receive a completion code which you should place in a text file named `survey.txt`. The survey is anonymous so submitting the completion code is the only way we can verify that you completed the survey. Please do not share this code with anyone, as that would constitute a breach of the honor code.

Submitting Your Assignment

Once you have finished this assignment, you will need to upload your files to Gradescope **AS A TAR.GZ FILE** using the command below. Make sure to upload all files to the Assignment 4 submission page. You should also upload survey.txt if you completed the survey.

Run this command in your assignment directory to submit it:

```
tar -cvzf ./assignment4.tar.gz .
```

Your Assignment 4 directory should have the following structure:

```
.
├── missing_fact.txt
├── screenshot.png
├── survey.txt
└── rock-paper-scissors/
    ├── .git/
    └── rock_paper_scissors.py
```

Here is what you need to submit for each part:

- Part I: A screenshot of your completion of the required parts of the tutorial. The filename of the screenshot doesn't matter, as long as it has a `.png` or `.jpg` extension.
- Part II: Your entire directory containing the Rock Paper Scissors game.
- Part III: On GitHub, make sure you've created a pull request with the appropriate changes to the main repository. On GradeScope, submit a `missing_fact.txt` file that contains the missing fact.
- Survey: `survey.txt`

Assignment 5: It all `make`s sense!

Due May 17th, 2023 at 11:59pm // [Gradescope]

Table of Contents

Overview

This assignment consists of two different components:

1. Practice using `make` and CI tools
2. Practice with profiling tools

We expect this assignment to take 1-3 hours depending on your proficiency level with the tools. If you find yourself unproductively stuck or unproductively struggling, ask on Ed and/or go to office hours!

Part I: Practice with Makefiles and CI Tools (2 points)

In this part of the assignment, you will practice writing a `Makefile` that compiles C files. Don't worry, you won't have to write any C code! Instead, we will provide you with the source code for a super secret, super advanced calculator program.

The program is exceptionally complex, and therefore the code consists of 6 (!!) separate files. Compiling 6 files individually is far too much work... We are soliciting your compiling expertise to help us compile the program.

The six files in the program are:

- `main.c`: which implements the core functionality of the calculator, such as prompting the user to enter an operation (+, -, /, *) and calling the appropriate helper function to execute the operation
- `addition.c`: which houses the `add` function
- `subtraction.c`: which houses the `subtract` function
- `multiplication.c`: which houses the `multiply` function
- `division.c`: which houses the `divide` function
- `operations.h`: which defines the prototypes for the `add`, `subtract`, `multiply` and `divide` functions

You can fork then clone these files from this repository on GitHub. You should make your own **fork** (just like you did in Assignment 4) and then work within that forked repository. For this part of the assignment, you'll be working entirely within the `calculator` directory. **Please make sure to enable Workflows on your forked repository before continuing: instructions are at the bottom of this section.**

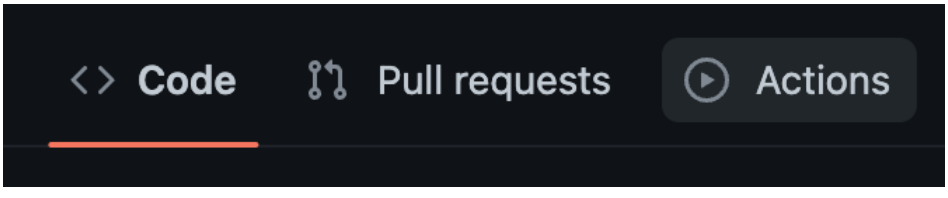
Your task is to (1) write a `Makefile` that will compile these files and allow us to run our program in an executable called `calculator`, then (2) write some tests that will automatically be run by CI. We would like to be able to run the following commands:

- `make`: which should produce an executable called `calculator`
- `make clean`: which should remove any previous binaries (e.g. `.o` files).

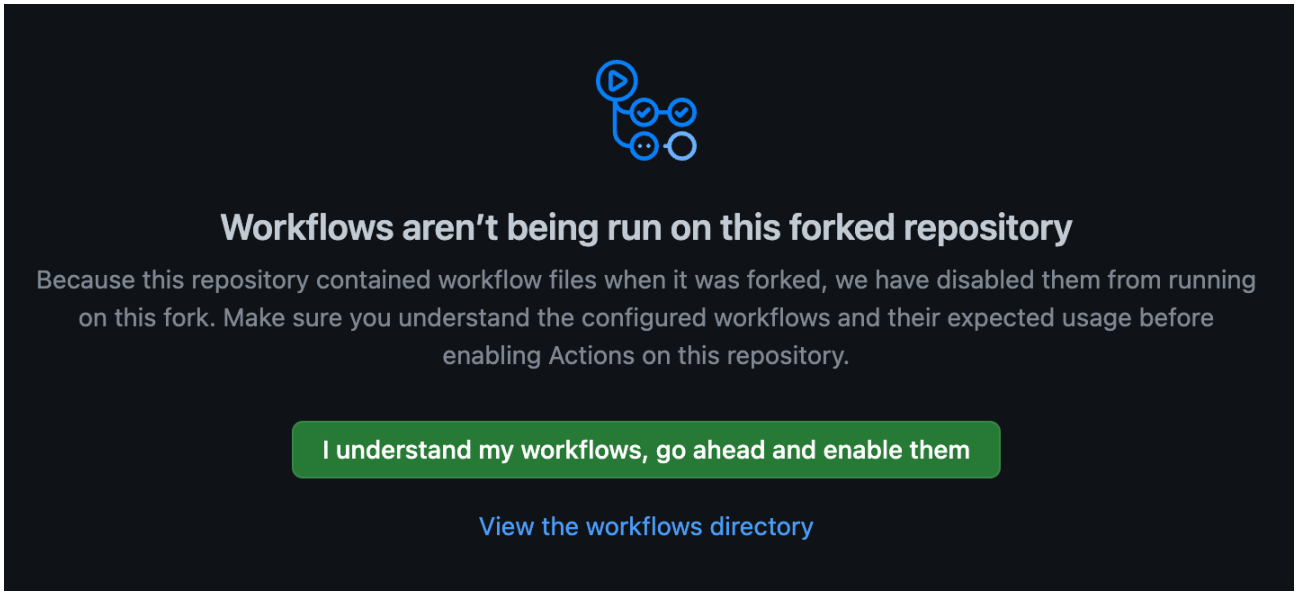
To test that your `Makefile` works, you should run make and then invoke the program using `./calculator` and the appropriate arguments. You should also try running `make clean && make` and then `./calculator` (with arguments). In both cases, you should successfully get the `calculator` program to run.

Remember that, when you run `make` without specifying a "target", it runs the first rule in the Makefile by default.

Then, add **CI Tests** in the `tests/your_tests.py` file. Don't worry if you're rusty on Python, we have some premade examples for you in the `tests/example_tests.py` file! Our CI file (which you can view at `.github/workflows` in the repository if you want!) will automatically run your tests. Please add two new tests to this file, so that when you push to your GitHub repository, you get a green checkmark ✅. You'll have to enable our workflows to run on your forked repository – go to "Actions" –



And then enable them by pressing the green “I understand and enable” button:



Put your Github username in a file called `github.txt`. We will use this to check that your CI tests are working.

Part II: Profiling Code (1 point)

For this part of the assignment, we will practice profiling some code. Our newest endeavor to produce cutting edge CS algorithmic research to find the most efficient implementation of sorting! The `sorts.py` file includes three different sort implementations. It’s available in the `profiling` directory of the repo you forked and cloned for Part I.

Our issue: we can’t figure out which one is fastest! Your task is to help us by using profiling tools to figure out which one is fastest in terms of CPU usage.

First, you should use `cProfile` to figure out which sorting implementation takes the most time. Remember that you can use the `-s tottime` flag to sort by the amount of time. Place your answer (the name of the sorting algorithm) on the first line of a file called `profiling.txt`.

Next you should figure out the bottleneck for each sort implementation using `line-profiler`. You will need to install line-profiler if you don’t have it already. You can install it using:

```
pip3 install line-profiler
```

To run the `line-profiler`, you will need to add the `@profile` decorator to each function that you want to profile. Once you’ve added the decorator(s), you can run:

```
kernprof -l -v sorts.py
```

Below are the first few lines of the output for the analysis for `quicksort`:

```
Total time: 0.058354 s
File: sorts.py
Function: quicksort at line 22

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    22                                           @profile
    23                                           def quicksort(array):
    24     17566       3791.0      0.2      6.5      if len(array) <= 1:
    25     17566       2720.0      0.2      4.7          return array
    26     16566       2670.0      0.2      4.6      pivot = array[0]
[ CONTINUED ]
```

You will want to identify which line takes the longest amount of time in the program as a whole, and which line takes the longest amount of time per execution. (If there is a tie, you can list the first line that has the longest time per execution.) For the sample output above, we see that the line `if len(array) <= 1:` takes the longest amount of time as a percentage of total time (6.5%).

However, in terms of time per execution (labeled `Time per Hit`), all three lines take the same amount of time. They each take 0.2 units of time, where a unit of time is 1µs (1 microsecond, i.e. 10^-6 seconds).

For your results for this part, you should include two answers (each on a separate line): the line that takes the longest in terms of percentage of total time, and the line that takes the longest in terms of time per execution.

Your final `profiling.txt` file should look as follows:

```
<slowest-algorithm>
<line-of-code-from-insertionsort-longest-percent-tottime>
<line-of-code-from-insertionsort-longest-time-per-hit>
<line-of-code-from-quicksort-longest-percent-tottime>
<line-of-code-from-quicksort-longest-time-per-hit>
<line-of-code-from-quicksort_inplace-longest-percent-tottime>
<line-of-code-from-quicksort_inplace-longest-time-per-hit>
```

For instance, if we used the sample results from above (which are wrong as they only look at a subset of the results for `insertionsort`), we could start populating our file as follows:

```
<slowest-algorithm>
if len(array) <= 1:
if len(array) <= 1:
<line-of-code-quicksort-longest-percent-tottime>
<line-of-code-quicksort-longest-time-per-hit>
<line-of-code-quicksort_inplace-longest-percent-tottime>
<line-of-code-quicksort_inplace-longest-time-per-hit>
```

Feedback Survey (0.5 points)

Once you have completed the assignment, you can earn an additional 0.5 points by completing our anonymous feedback survey. Given this is the first offering of the course, we want to collect as much feedback as possible to improve the course in the future. You can complete the survey here.

Once you complete the survey, you will receive a completion code which you should place in a text file named `survey.txt`. The survey is anonymous so submitting the completion code is the only way to verify that you completed the survey. *Please do not share this code with anyone, as that would constitute a breach of the honor code.*

Submitting Your Assignment

Once you have finished this assignment, you will need to upload your files to Gradescope. Make sure to upload all files to the Assignment 5 submission page. You should also upload `survey.txt` if you completed the survey.

This assignment will be autograded for Part I, Task I (the Makefile) and Part II (Profiling). Part I, Task II (CI tests) will be manually graded. You will thus see that Part I, Task II will not be graded until one of us goes through and checks your tests.

```
zip -jv assign5_submission.zip ./calculator/Makefile ./profiling/profiling.txt ./github.txt ./survey.txt ./cal
```

All files must have the same name as specified above.

Assignment 6: The (Public) Key to My Heart

Due May 24th, 2023 at 11:59pm // [Gradescope]

Table of Contents

Overview

This assignment consists of two different components:

1. Spoof an email
2. Set up SSH keys to sign a message

We expect this assignment to take 1-3 hours depending on your proficiency level with the tools. If you find yourself unproductively stuck or unproductively struggling, ask on Ed and/or go to office hours!

> ***Warning**: This assignment teaches you a technique for email spoofing that could be used to compromise another's security. You should not use this technique outside of this assignment. Neither CS45 course staff nor Stanford will take liability for any legal repercussions resulting from using this technique outside the context and parameters of this assignment.*

Part I: Spoof an Email (1.5 points)

For this part of the assignment, you will get to spoof an email.

The way email spoofing works is that we are leveraging the absence of authentication that should be present to ensure that the sender of the email is authorized to send an email on behalf of that sender. You might expect that you would first have to authenticate into an account (i.e. log in and provide a password), before being able to send an email from that account.

Instead, we find that most email is sent using SMTP (Simple Mail Transfer Protocol), which is an insecure protocol that doesn't actually check that the sender is authorized to send from that address.

Please note that we are showing you this technique as part of an assignment, with the purpose of teaching you about security techniques. Read the disclaimer above before starting this part of the assignment.

Thank you to Keith Winstein for inspiration for this part of the assignment.

Exercise 1: Send an Email to Yourself

1. To get started, you will want to log into our CS45 server. To log in, open a Terminal and type the following two lines, replacing `<SUNet>` with your SUNet:

```
export SUNET=<SUNet>
ssh s-cs45-assign6-$SUNET@138.2.228.214
```

Once you are logged in, you will automatically be directed to run the Simple Mail Transfer Protocol (`smtp`). (We do this to ensure that you can't "explore" around our server and "accidentally" take it down... Security 😁).

If all goes well, you should get immediately get some output that looks like the following immediately after logging in:

```
Trying 127.0.0.1...
Connected to localhost.
```

```
Escape character is '^]'.
220 honeypot.vcn.oraclevcn.com ESMTP Postfix (Ubuntu)
```

2. Next, you'll want to specify who the email is from (which in this case, is yourself). To do so, you will want to type in `MAIL FROM:`, a space, your email address, and then hit the Enter key. This should look as follows:

```
MAIL FROM: <SUNET>@stanford.edu ↵
```

If all goes well, you will see `250 2.1.0 Sender ok`.

If you are running into an issue where the sender address is not being accepted (i.e. you are getting an error), make sure you don't have extra characters anywhere. You can also try encapsulating the address with `<` and `>` as in: `MAIL FROM: <adrazen@stanford.edu>`

3. Now, you will want to specify who the email is to (which in this case, is yourself). To do so, you will want to type in `RCPT TO:`, a space, your email address, and then hit the Enter key. This should look as follows:

```
RCPT TO: <SUNET>@stanford.edu ↵
```

If all goes well, you will see `250 2.1.0 Recipient ok`.

If you are running into an issue where the recipient address is not being accepted (i.e. you are getting an error), make sure you don't have extra characters anywhere. You can also try encapsulating the address with `<` and `>` as in:

```
RCPT TO: <adrazen@stanford.edu>
```

4. Now it's time to construct the message itself. Type `DATA` and then hit the Enter key.

```
DATA ↵
```

If all goes well, you will see `354 End data with <CR><LF.<CR><LF>`.

First, we will add the headers. This may feel redundant to the `MAIL FROM:` and `RCPT TO:` that you entered above. However, the `MAIL FROM:` and `RCPT TO:` from above are part of the envelope, and are used by the SMTP protocol to specify specific delivery instructions for where the email needs to go.

Meanwhile, the headers that we will add below are used by email clients (i.e. the sender and receiver), but are not used for actually routing and delivering the email. You should add a `From`, `To` and `Subject` header. Make sure to add a blank line at the end of the headers. This should look as follows:

```
From: <SUNET>@stanford.edu ↵
To: <SUNET>@stanford.edu ↵
Subject: Hello from CS45! ↵
↵
```

5. Now you can construct the email body. You can write anything you want! When you are done constructing the email body, press the Enter key. Then add a line with just a `.` (period) on it and then press Enter. This should look something like this:

```
354 End data with <CR><LF.<CR><LF>
Hi this is my email! ↵
. ↵
```

If all goes well, you should see a confirmation message: `250 2.0.0 33h24dpdsr-1 Message accepted for delivery.`

6. When you are done, type `QUIT` and hit Enter to exit out of the email server.

7. If you go to your Stanford email inbox, you should see a new email from yourself! (If you don't see it after a few minutes, check your spam folder.)

Exercise 2: Send a Spoofed Email to Yourself

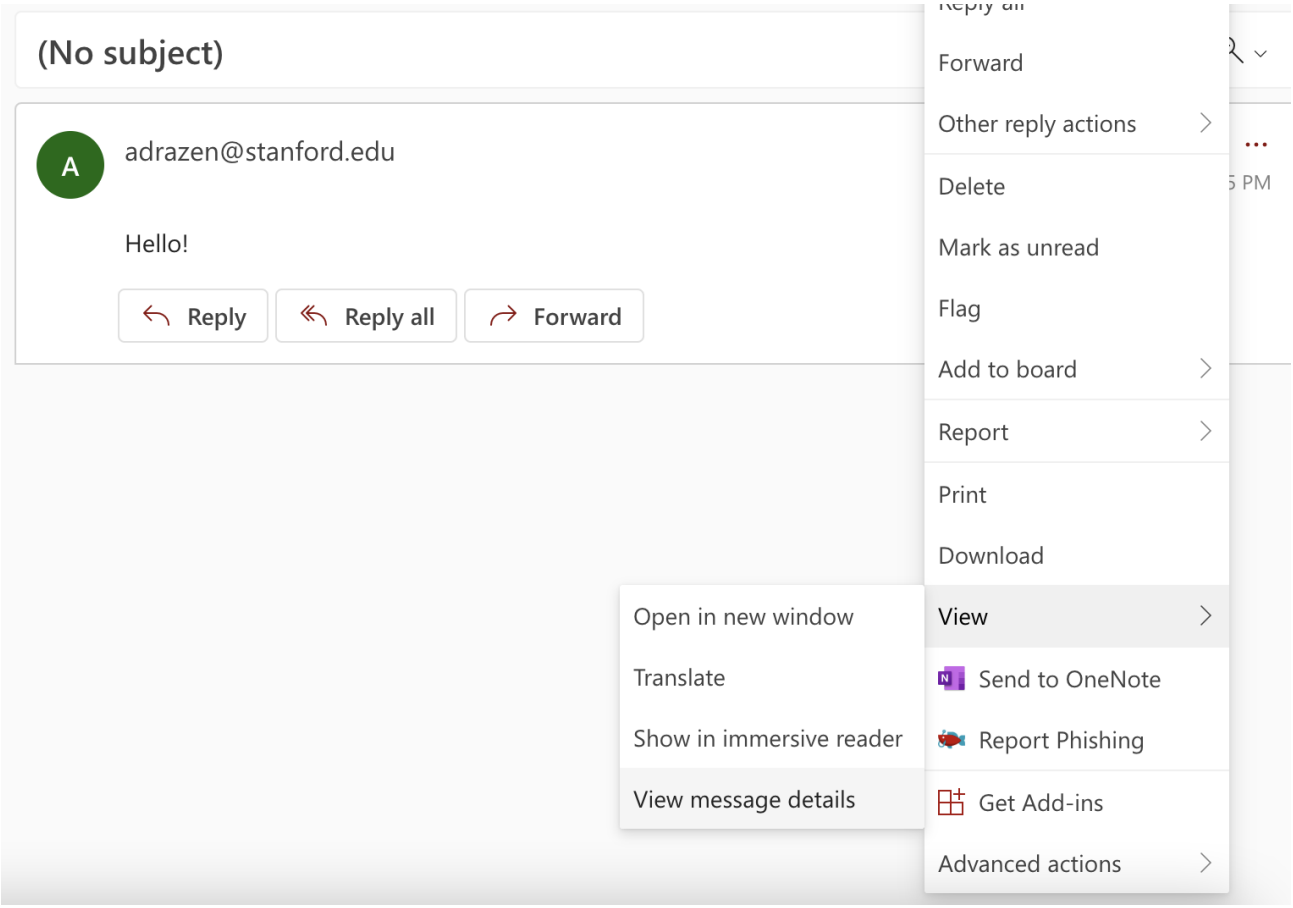
To send a spoofed email to yourself, you will want to repeat the process from Exercise 1.

However, this time you should change the `MAIL FROM:` address in step 2 to be `cs45-admin@stanford.edu`. (This is a fake email address that we are allowing you to use as part of this assignment.) You will also want to change the `From:` header in Step 4 to be

`cs45-admin@stanford.edu`.

What happens if you change one and not the other? Try changing just the `MAIL FROM:` address in step 2 and not the `From:` header in Step 4. Now try changing just the `From:` header in Step 4 and not the `MAIL FROM:` address in step 2. Can you see a difference?

To explore what the difference is, try opening the spoofed emails in the Outlook mail server. Open the "More Actions" menu (three dots) and then select "View > View Message Details". Can you see anything suspicious about the email here?



Exercise 3: Send a Spoofed Email to Us

Now you'll want to send us a spoofed email from `cs45-admin@stanford.edu`. You'll want to repeat the steps in Exercise 2 and send the email to `cs45-spr2223-staff@mailman.stanford.edu`.

In order to confirm that your email gets successfully delivered, we recommend CCing yourself on the email. In order to do that, you will want to add an additional recipient using `RCPT TO:`. In other words, you should repeat Step 3 from above twice. The first time, you should add the recipient as `cs45-spr2223-staff@mailman.stanford.edu` and the second time you should add the recipient as your own email address. You can set the `To:` header to just be `cs45-spr2223-staff@mailman.stanford.edu`.

Make sure to include your SUNet somewhere in the message body as this is how you will get credit for this part of the assignment.

Part II: Generate SSH Key to Sign A File (1.5 points)

For this next part of the assignment, we'll get to use asymmetric cryptography! We'll set up SSH (Secure Shell) keys and then see how we can use these keys to sign a file.

Step 1: Setting up SSH Keys

To begin, we need to set up our SSH keys. You may have set up SSH keys before in the past. If you already have SSH keys, you can skip to Step 2.

SSH keys are incredibly useful. To check whether you have existing SSH keys, you can run the following command:

```
ls -al ~/.ssh
```

The output of this command will list all of your SSH keys. If you have existing SSH keys, the output will contain two lines that look something like this:

```
-r--------   1 adrazen  staff   411 Sep 22  2021 id_ed25519
```

```
-rw-r--r--   1 adrazen  staff   102 Sep 22  2021 id_ed25519.pub
```

Notice the different file permissions between the first line (containing the private key) and the second line (containing the public key).

If you don't have SSH keys, you'll want to run the following command, replacing `your_email@example.com` with your Stanford email address:

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

Once you hit Enter, the command will tell you that it's generating your Public/Private key pair with the message Generating public/private `ed25519` key pair.

It will then prompt you asking you where to save your SSH keys:

```
Enter a file in which to save the key (/Users/your_username/.ssh/id_ed25519): [Press enter]
```

You should hit the Enter button in order to use the default location (`/Users/your_username/.ssh/id_ed25519`).

You will then be prompted to provide a secure passphrase to protect your SSH keys. Choose a passphrase that you won't forget.

Congratulations! You've successfully generated your SSH keys. You can now examine your newly-minted SSH keys by navigating to ~/.ssh. Inside of this directory you should see two files: id_ed25519 and id_ed25519.pub. These are your private and public keys respectively. Feel free to print them out using cat to examine their contents (in particular, you'll need the contents of the `id_ed25519.pub` file in the next part). Remember, it's okay to share your public key with others, but your private key should always be a secret. If someone has your private key, your key is "compromised" (they can now pretend to be you and take actions on your behalf).

Step 2: Signing A File

Let's sign a file using your SSH key! First you will need to create a file. Create a file called file.txt with a fun message or fact that you want to sign and send to us!

To sign a file using your SSH key, you will want to run the following command:

```
ssh-keygen -Y sign -n file -f ~/.ssh/id_ed25519 file.txt
```

If you're interested in learning about the flags we are using with ssh-keygen, you can run man ssh-keygen to learn more. If all goes well, running this command should sign the provided file and generate a signature file ending in `.sig`.

Now, you'll want to provide us with all of the necessary information so we can verify that you indeed signed this file.

To do so, you'll want to create a file called `allowed_signer` that contains your email, the cryptography algorithm you used (`ed25519`) and your public key. The contents of `allowed_signer` file should look as follows:

```
<email_address> ssh-ed25519 <your-public-key>
```

Make sure to give us your public key (from the `.pub` file), not your private key. No one should ever see your private key except you!

Now it's time to submit! You will need to submit your signature file (ending in `.sig`), your `allowed_signer` file, and your original file. We'll verify your signature to make sure it's really your file.

Feedback Survey (0.5 points)

Once you have completed the assignment, you can earn an additional 0.5 points by completing our anonymous feedback survey. Given this is the first offering of the course, we want to collect as much feedback as possible to improve the course in the future. You can complete the survey here.

Once you complete the survey, you will receive a completion code which you should place in a text file named `survey.txt`. The survey is anonymous so submitting the completion code is the only way to verify that you completed the survey. *Please do not share this code with anyone, as that would constitute a breach of the honor code.*

Submitting Your Assignment

Once you have finished this assignment, you will need to upload your files to Gradescope in addition to sending the spoofed email in Part 1. Make sure to upload all files to the Assignment 6 submission page. You should also upload `survey.txt` if you completed

the survey. The autograder for this assignment will grade Part II and your survey code.

```
zip -jv assign6_submission.zip ./survey.txt ./file.txt ./file.txt.sig ./allowed_signer
```

All files must have the same name as specified above.

Assignment 7: On Cloud Nine

Due May 31st, 2023 at 11:59pm // [Gradescope]

Table of Contents

Overview

This assignment consists of two interconnected components:

1. Create and publish a Docker image on DockerHub
2. Launch a free tier instance on AWS and run a server on it (the same server you built in part 1!)

We expect this assignment to take 1-3 hours depending on your proficiency level with the tools. If you find yourself unproductively stuck or unproductively struggling, ask on Ed and/or go to office hours!

Part I: Build and Publish a Docker Image (2 points)

In this part of the assignment, you'll download some starter code we've provided for you, and build an optimized Dockerfile around that code, which will (1) build the code, then (2) package it in a slim docker image, and (3) publish it on DockerHub. You'll get practice with (and learn about!) several Dockerfile constructs – `FROM`, `COPY`, `RUN`, `ENV`, `WORKDIR`, `VOLUME`, and `EXPOSE` – as well as the process of building and publishing Docker images. This is pretty much everything you need in order to build docker images in practice.

Launch Docker Desktop before proceeding; on Windows, you should have access to Docker through WSL. If you get any firewall prompts, allow it.

Step 0: Sign Up for DockerHub

DockerHub is the canonical registry where all images are stored and where Docker looks for images when it can't find it locally. (When we've used e.g. `ubuntu:latest` in lecture, or `jdkula/calculator:latest`, it was downloading images published here).

In order to publish your own images, you'll need an account. Sign up here: https://hub.docker.com/

Step 1: Understand the code we're building

The code is available at https://github.com/stanford-cs45/win23-a8 – no need to fork this repository, but go ahead and clone it to your local computer. Read the `README.md` file, which is also available by just scrolling down on the repository's page above. This will give us an idea of what we'll need to do in our `Dockerfile`. I'll highlight a few things here:

- As part of making our docker image, we'll need to **build** the code, and then tell Docker **how to run the code**. The README tells us how to do both.
- It mentions that the code needs Node.JS in order to run.
    - We need to pick a base image to use. We could use ubuntu:latest like we did in lecture and install everything we need manually, but it turns out there's already a ready-made image for doing things with Node.js. You should use `node:18` as your base image.
- It mentions a couple commands we'll need in order to build. These are good candidates for `RUN` commands in our Dockerfile.
- In the part about running the server, it mentions a few things it needs in order to run:

- It mentions that we need to create and persist a `/data` directory– this is where the messages are stored. This would be a great opportunity to use a **volume**. Read up on the `VOLUME` command on Docker's documentation to know how to use it.
- It mentions we need to use an environment variable called `PASSWORD` in order for anyone to be able to post anything. Read up on the `ENV` command on Docker's documentation, which allows you to specify a default environment variable, and then decide whether or not it's a good idea to include a default password in your Dockerfile.
- It mentions that the server will be hosted on port `3000`. Read up on the `EXPOSE` command on Docker's documentation, which allows you to specify network ports the container exposes (that might want to be passed through from the host).
- It mentions that it needs to use the `node build` command to run the server. That would make a good start for the `ENTRYPOINT` command, which you can read about on Docker's documentation here.

Step 2: Create the Dockerfile

Now that we have some idea of what we want to do, we'll create a Dockerfile in the `win23-a8` directory we cloned and fill it out. Your `Dockerfile` should have the following structure:

- Set the base image to node:18
- Establish a working directory of your choice where the app will live
- Copy in everything from the current directory
- Install dependencies and build the application
- Inform docker of the volume for `/data`, expose the port, and (optionally, if you think it's a good idea), give a default for the `PASSWORD` environment variable.
- Set the entrypoint.

Step 3: Build a Docker image

You'll need your DockerHub username for this step. To build your Docker image, run the following command:

```
docker build . -t USERNAME/messageboard:latest
```

For example, my DockerHub username is `jdkula`. I would run the following command: `docker build . -t jdkula/messageboard:latest`

This will build your docker image and tag it as `USERNAME/messageboard:latest` – this allows you to refer to the image in later `docker` commands.

Step 3.5: Test your image

You can test your image out like so:

```
docker run -d -e PASSWORD=yourpassword -v "$(pwd)/data:/data" -p 8080:3000 USERNAME/messageboard:latest
```

Let's break this down:

- `-d` tells Docker to run it in the background.
- `-e PASSWORD=yourpassword` sets the environment variable `PASSWORD` to `yourpassword`
- `-v $(pwd)/data:/data` creates a folder called data in the current folder, and attaches it to `/data` inside the container.
  - This is the weirdest line to understand! Let's break it down:
  - The format is `-v HOST:CONTAINER`. It says, take the folder in the `HOST` side, and make it so that the folder on the `CONTAINER` side is connected to it.
  - `$(pwd)/data` will expand to your current working directory with `/data` appended – for example, if I'm at `/Users/jdkula/win23-a8`, then `$(pwd)/data` would be `/Users/jdkula/win23-a8/data`
  - `/data` is the path *within the container* where `$(pwd)/data` will be mounted.
  - This all means that if something in the container writes something to `/data` – for example, `/data/messages.json` – it will appear outside the container inside `$(pwd)/data`, and vice versa!
- `-p 8080:3000` tells Docker to map the port `8080` on the host (i.e. your computer!) to port `3000` inside the container. (In a nutshell, this lets you access the server running inside the container, from outside the container. It runs on port `3000` inside the container, but is accessible via port `8080` outside the container.).

While the container is running, you can go to `localhost:8080` in your web browser to see it and make sure it works!

The docker run command above will output a long hash for the container. You can stop it from running by running

```
docker kill INSERT_SHA256HASH_HERE
```

```
# Example:
docker kill aa64b0cd137e9c0d991d1a872c9fb21fa79450126fa6b23f0aa4f094e861dbd9
```

Step 4: Publish your Docker image

You can run the following command to publish it on DockerHub:

```
# only once, to login:
docker login
# then, to push the image:
docker push USERNAME/messageboard:latest  # replace USERNAME with your DockerHub username.
```

If you're on an M1 mac, use the following command to build and push an image that works for all major architectures:

```
# Run once to set up
docker buildx create --use

# replace USERNAME with your DockerHub username
docker buildx build --platform linux/amd64,linux/arm64 -t USERNAME/messageboard:latest . --push
```

For this part, create a file called `image_tag.txt` and put your `USERNAME/messageboard:latest` tag in that file. We'll pull and run your image to grade it, as well as build your image from your `Dockerfile`.

Part II: Launch a free tier instance on AWS and run a server on it (1 point)

In this part of the assignment, you'll launch an instance on the free tier of AWS, install Docker on it, and use Docker to run a server on your instance. You'll get practice with launching and the bare basics of administering an instance on AWS; as well as get practice with running Docker containers, using port forwarding, modifying environment variables, and Docker volumes (specifically, bind mounts). This will give you what you need to use most Docker images!

Step 0: Sign up for AWS

Go to https://aws.amazon.com/ and sign up for a new account. You may need to enter in some credit card information to access this service– let us know if this is a problem for you and we can provide you with an instance.

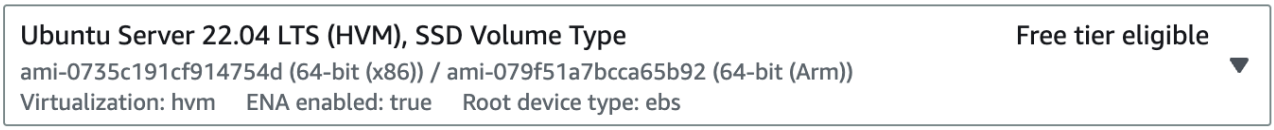
Step 1: Create a Keypair

Once you're in AWS, go to the EC2 dashboard and find the "Key Pairs" page. Then, click "Actions" and then "Import Key Pair." Name it whatever you'd like, and then paste in the public key you made in Assignment 7. We'll use this to log into your EC2 instance.

Step 2: Launch your instance

Go back to the EC2 Dashboard, and press the "Launch Instance" button, then "Launch Instance" again.

Name your instance whatever you'd like. Then, under "Application and OS Images," select Ubuntu and ensure you're using an Amazon Machine Image ("AMI") that says "Free tier eligible."



For Instance Type, choose `t2.micro`, or any other free tier eligible instance (preferably the cheapest possible– although you should not be charged in any case, as you'll have 750 free hours of `t2.micro` for the first year of your AWS account).

Then, choose the keypair you created earlier.

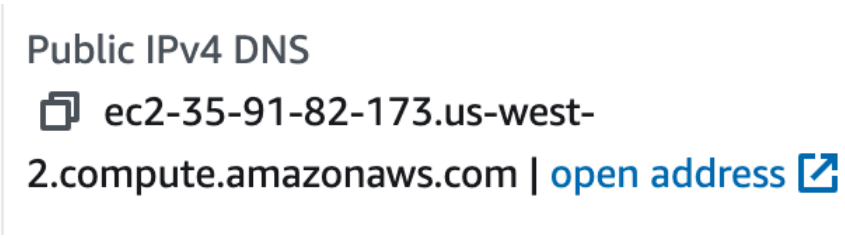
Under Network Settings, check the boxes to allow SSH, HTTPS, and HTTP traffic from the internet.

Finally, under Configure Storage, change it to 25 GiB of storage (gp2 as the storage type is OK).

Then, press "Launch Instance." Amazon will launch your instance– boom! Computer created!

Step 3: Connect to your instance

Go to the Instances page inside the EC2 dashboard, and find the instance you just launched. Click its id (which will look like `i-03264e498b6ad8ff4`) to be taken to its instance summary. Find its Public IPv4 DNS address:



Copy it, and go to your terminal. Connect to it like so:

```
ssh ubuntu@ADDRESS

# Example:
ssh ubuntu@ec2-35-91-82-173.us-west-2.compute.amazonaws.com
```

It will ask you about host authenticity. Type `"yes"` and then press enter.

And... congratulations! You're in!

Step 4: Install Docker

This step's easy. Just copy-paste in the following command to your new EC2 instance:

```
curl -fsSL https://get.docker.com | bash
```

Step 5: Run your Messageboard!

Let's run your message board from before! We can use nearly the same command as before (in part 1 step 3.5) to run it:

```
sudo docker run -d -e PASSWORD=yourpassword -v "$(pwd)/data:/data" -p 80:3000 USERNAME/messageboard:latest
```

We've changed two things: first, now we use `sudo` since we need superuser access to do things with Docker; and we've changed our HOST port to `80` instead of `8080`, since `80` is the standard web port!

Once you do this, you can go to your address (e.g. ec2-35-91-82-173.us-west-2.compute.amazonaws.com) in your browser to see it running!

(Optional Step 6: Add HTTPS)

We can use a program called `caddy` to easily add HTTPS to our server. It works by acting as a reverse proxy– this means that it will stand in front of our server, and add HTTPS for us. It'll acquire a certificate automatically from a service called ZeroSSL.

First, kill the server you started in Step 5– we need to change how we're running it. (You can use the same commands from Part II, Step 3.5, just make sure to use sudo now). Then, let's run it again (note we're back to using port 8080 again).

```
sudo docker run -d -e PASSWORD=yourpassword -v "$(pwd)/data:/data" -p 8080:3000 USERNAME/messageboard:latest
```

Then, let's run `caddy` in another docker container(!!), being sure to replace `YOUR_DNS_NAME` with the name of your EC2 instance (e.g. ec2-35-91-82-173.us-west-2.compute.amazonaws.com):

```
sudo docker run --net host -d -v caddy_data:/data caddy caddy reverse-proxy --from YOUR_DNS_NAME --to localhos

# Example:
sudo docker run --net host -d -v caddy_data:/data caddy caddy reverse-proxy --from ec2-35-91-82-173.us-west-2.
```

Wait a few minutes for it to negotiate the HTTPS certificate with ZeroSSL/Let's Encrypt, and then visit your EC2 instance in your web browser and see that sweet, sweet lock icon!

For this part, create a file called `server_address.txt` and put your EC2 address in it, e.g. `ec2-35-91-82-173.us-west-2.compute.amazonaws.com`. Keep the server running during autograding, which will open the webpage to make sure it's running. You do not need to set up HTTPS in order to get full marks on this part, but it's a useful exercise!

> *While your server needs to remain up while grading, **please remember to terminate your AWS instance**– otherwise you may end up being charged at some point in the future.*

Feedback Survey (0.5 points)

Once you have completed the assignment, you can earn an additional 0.5 points by completing our anonymous feedback survey. Given this is the first offering of the course, we want to collect as much feedback as possible to improve the course in the future. You can complete the survey here.

Once you complete the survey, you will receive a completion code which you should place in a text file named `survey.txt`. The survey is anonymous so submitting the completion code is the only way to verify that you completed the survey. Please do not share this code with anyone, as that would constitute a breach of the honor code.

Submitting Your Assignment

Once you have finished this assignment, you will need to upload your files to Gradescope in addition to publishing your Docker image and keeping your AWS instance running. Make sure to upload all files to the Assignment 8 submission page. You should also upload `survey.txt` if you completed the survey.

This autograder for this assignment will grade parts I and II and your survey code.

```
zip -jv assign8_submission.zip ./survey.txt ./image_tag.txt ./server_address.txt ./Dockerfile
```

All files must have the same name as specified above.

Assignment 8: Codec of Conduct

Due June 7th, 2023 at 11:59pm // [Gradescope] [Files]

Table of Contents

Overview

This week's assignment will be shorter than usual to give you time to work on final projects.

This assignment consists of four short components:

1. Practice with common errors when writing code that deals with numbers.
2. Practice with various forms of text encoding.
3. Practice with various forms of image encoding.
4. Practice with various forms of video encoding.

We expect this assignment to take 0-1 hours depending on your proficiency level with these concepts. If you find yourself unproductively stuck or unproductively struggling, ask on Ed and/or go to office hours!

You'll probably want to install FFmpeg using the instructions from the software page.

This assignment should be completed on Gradescope. The questions are repeated here for completeness. You can download the files for the assignment here.

Part I: Number Encoding (0.3 points)

Unsigned Integers (0.05 Points)

What would the result of this code be?

```
// uint8_t is an 8-bit unsigned integer
uint8_t x = 45;
uint8_t y = 90;
return x - y;
```

Signed Integers (0.05 Points)

What would the result of this code be?

```
// int8_t is an 8-bit signed integer
int8_t x = 45;
```

```
int8_t y = 90;
return x - y;
```

Floats (0.1 Points)

What would the result of this code be?

```
// float is a 32-bit floating point number
float x = 0.1;
return 0.3 - (3 * x);
```

- 0
- 0.1
- 0.2
- 0.3
- none of the above

Endianness (0.1 Points)

Assume I'm using an Intel x86-powered CPU (x86 CPUs are little-endian).

I hexdump a file and see that it contains these four bytes (represented as hexadecimal numbers); the byte on the left is at the lowest address and the byte on the right is at the highest address:

```
43 53 34 35
```

Which of the following could be the true contents of the file?

- The 8-bit numbers 0x43, 0x53, 0x34, and 0x35.
- The 16-bit numbers 0x4353 and 0x3435.
- The 16-bit numbers 0x5343 and 0x3534.
- The 32-bit number 0x43533435.
- The 32-bit number 0x35345343.
- The ASCII or UTF-8 string "CS45".
- The ASCII or UTF-8 string "54SC".

Part II: Text Encoding (0.7 Points)

You can download the files for the assignment here.

Useful commands:

- `ls -l` will print the sizes of each file
- `ls -lS` will sort the files by size (this may only work on some computers)
- `file` can try to figure out the text encoding of a file automatically
- `iconv [filename] -f [encoding] -t UTF-8` will try to decode the file `filename` as if it were using the encoding `encoding`.

The encodings used are: ASCII, UTF-8, UTF-16LE, UTF-16BE, UTF-32LE, and UTF-32BE.

Text Encoding Identification (0.05 Points)

- Which file is ASCII?
- Which file is UTF-8?
- Which files are UTF-16?
- Which files are Little Endian (either UTF-16 or UTF-32)?
- Which files are Big Endian (either UTF-16 or UTF-32)?

File Size (0.2 Points)

Assuming you want to optimize for size while preserving 100% accuracy with the original file, which encoding should you choose?

- ASCII
- UTF-8
- UTF-16
- UTF-32

If you want to really optimize for size, even if it means losing a little data, which encoding should you choose?

- ASCII
- UTF-8

- UTF-16
- UTF-32

Morse Code (0.15 Points)

One very old, very widespread form of text encoding is Morse Code. However, Morse Code has some issues that make it unfeasible as a text encoding for computer use.

Which of these issues with Morse Code are solved by UTF-8? Which are solved by ASCII? Which are solved by UTF-32?

- different characters have different length encodings
- it only supports basic English characters without country-specific extensions
- its encodings are not 32 bits long, and therefore are less efficient to access on modern CPUs
- it requires three states (short, long, and pause) to represent, not just two
- there are many language-specific variants

Image Encoding (1 Point)

You can download the files for the assignment here.

Useful commands:

- `ls -l` will print the sizes of each file
- `ls -lS` will sort the files by size (this may only work on some computers)
- `file` can try to figure out the image format of a file automatically

We accidentally erased the file extensions from these images! Help us figure out which format is which.

Matching Files and Encodings (0.3 Points)

Match the files `a.dat`, `b.dat`, `c.dat`, `e.dat`, and `f.dat` with the encodings BMP, JPEG, TIFF, GIF, PNG, HEIF.

Image Size (0.35 Points)

If we wanted to optimize for smallest file size, which image format should we choose?

Size Ratio (0.35 Points)

What is the ratio between the sizes of the smallest and largest images?

Your answer should be (size of smallest) ÷ (size of largest). Provide the answer as a decimal number (e.g., 0.01), **not** a percentage.

Video Encoding (1 Point)

You can download the files for the assignment here.

Useful commands:

- `ls -l` will print the sizes of each file
- `ls -lS` will sort the files by size (this may only work on some computers)
- `ffprobe -hide_banner -show_entries stream=codec_name [filename]` will print out information about the codecs used within a video file. You'll have to install ffmpeg to use this (`sudo apt install ffmpeg`, `brew install ffmpeg`, or follow the instructions from https://ffmpeg.org/download.html).

We encoded the same video with a bunch of codecs, but we forgot which file is which codec! Help us sort them out.

`ffprobe`'s output can be a lot of information. Remember, you know lots of shell tools to help you wrangle data!

Matching Files and Codecs (0.25 points)

Match the files `{a,b,c,d,e}.mkv` with the encodings H.265, H.264, Theora, VP9, and VP8.

Audio (0.05 Points)

Which audio codec do all of these files use?

- MP3
- FLAC
- Opus
- Vorbis
- AAC

Quality: Frame Rate (0.2 Points)

One of these files was encoded with the wrong settings, so the video ended looking very choppy (it has an effective framerate of around 1 frame per second instead of the expected 30). Which codec was used on this file?

Quality: Image (0.2 Points)

One of these files was encoded with too much compression, so the image quality within the video dropped. Which codec was this?

Hint: this is different than the answer to Q4.7

Video Compression (0.3 Points)

At its core, a video is just a sequence of images shown at a specific speed. In fact, when you decode a video and play it on a screen, it turns into exactly this. Since we already have compressed image formats, what's the point of having video-specific formats? Why can't we just compress the images individually and wrap them up in a container?

Select all of the following which are true:

- video compression can take advantage of similarities between the video and the audio, but image compression can't
- video compression doesn't need the same level of detail in images as image compression does
- video compression can take advantage of similarities in images over time, but image compression can't
- video compression has to be lossless, but image compression is lossy
- video decompression needs to happen quickly (in real-time), but image decompression doesn't
- video formats are designed to be sent over the internet, but image formats are designed to be read from a hard drive

Feedback Survey (0.5 points)

Once you have completed the assignment, you can earn an additional 0.5 points by completing our anonymous feedback survey. Given this is the first offering of the course, we want to collect as much feedback as possible to improve the course in the future. You can complete the survey here.

Once you complete the survey, you will receive a completion code which you should place in a text file named `survey.txt`. The survey is anonymous so submitting the completion code is the only way to verify that you completed the survey. *Please do not share this code with anyone, as that would constitute a breach of the honor code.*

Submitting Your Assignment

Once you have finished this assignment, you will need to upload your files to Gradescope. Make sure to upload your files to the Assignment 7 submission page. You should also upload `survey.txt` if you completed the survey.

```
zip -jv assign7_submission.zip ./Makefile ./survey.txt
```

All files must have the same name as specified above.

Final Project

Due: June 9, 2023 at 11:59pm

For your final project, we want you to explore a tool or concept from the course (or a tool that we didn't discuss in the course but that you are interested in!) in further detail. Your role is to do a deep dive into this tool/concept, and summarize what you've learned in order to help others learn about how and when to use it. We want you to become an expert in one of these tools and teach the rest of us about it :)

You don't have to choose a totally new tool for this project—you could go deeper into one we already covered. For example, you may be interested in diving into the usage of `sed`. While we only covered using it for simple searching and replacing, it can do a lot more cool/useful things. In that case, you could explore `sed` in more depth and provide use cases beyond those discussed in class.

In terms of deliverables, we expect you to provide the following:

If you're researching a tool:

1. A written guide for how to use the tool, including tips, tricks and neat features
2. 3-4 slides describing the tool and giving examples of when it's useful

If you're researching a concept:

1. A writeup describing the concept to readers who are unfamiliar with it, and defining any important terms and/or acronyms
2. 3-4 slides explaining the concept at a high level, and providing key takeaways about it

Please submit both your written guide and your slides on Gradescope. There are two separate "Assignments" for the written guide and the slides. Both need to be submitted as PDFs.

Our goal is to create a compendium of tools that we can then share with everyone in the class.

Optionally, if you would like to present your topic to the class during our last meeting on Wednesday of Week 10, let us know and we'll reserve some time for you.

If you're struggling to think of ideas, here are some ideas of what you could cover:

- `sed`
- `awk`
- Perl
- Jupyter Notebooks
- Globbing
- `jq`
- Version control systems other than Git (e.g. Perforce, Mercurial)
- CI/CD tools

- `emacs`
- Web debugging
- VSCode extensions
- Alternative shells (e.g. Windows PowerShell)

If you want more suggestions, post on Ed and mention topics you're interested in learning more about—we can give you pointers for things to explore in those topics.