课程概览与 shell

动机

作为计算机科学家，我们都知道计算机最擅长帮助我们完成重复性的工作。 但是我们却常常忘记这一点也适用于我们使用计算机的方式，而不仅仅是利用计算机程序去帮我们求解问题。 在从事与计算机相关的工作时，我们有很多触手可及的工具可以帮助我们更高效的解决问题。 但是我们中的大多数人实际上只利用了这些工具中的很少一部分，我们常常只是死记硬背一些如咒语般的命令， 或是当我们卡住的时候，盲目地从网上复制粘贴一些命令。

本课程意在帮你解决这一问题。

我们希望教会您如何挖掘现有工具的潜力，并向您介绍一些新的工具。也许我们还可以促使您想要去探索（甚至是去开发）更多的工具。 我们认为这是大多数计算机科学相关课程中缺少的重要一环。

课程结构

本课程包含 11 个时长在一小时左右的讲座，每一个讲座都会关注一个 特定的主题。尽管这些讲座之间基本上是各自独立的，但随着课程的进行，我们会假定您已经掌握了之前的内容。 每个讲座都有在线笔记供查阅，但是课上的很多内容并不会包含在笔记中。因此我们也会把课程录制下来发布到互联网上供大家观看学习。

我们希望能在这 11 个一小时讲座中涵盖大部分必须的内容，因此课程的信息密度是相当大的。为了能帮助您以自己的节奏来掌握讲座内容，每次课程都包含一组练习来帮助您掌握本节课的重点。 课后我们会安排答疑的时间来回答您的问题。如果您参加的是在线课程，可以发送邮件到 missing-semester@mit.edu 来联系我们。

由于时长的限制，我们不可能达到那些专门课程一样的细致程度，我们会适时地将您介绍一些优秀的资源，帮助您深入的理解相关的工具或主题。 但是如果您还有一些特别关注的话题，也请联系我们。

主题 1: The Shell

shell 是什么?

如今的计算机有着多种多样的交互接口让我们可以进行指令的的输入，从炫酷的图像用户界面（GUI），语音输入甚至是 AR/VR 都已经无处不在。 这些交互接口可以覆盖 80% 的使用场景，但是它们也从根本上限制了您的操作方式——你不能点击一个不存在的按钮或者是用语音输入一个还没有被录入的指令。 为了充分利用计算机的能力，我们不得不回到最根本的方式，使用文字接口：Shell

几乎所有您能够接触到的平台都支持某种形式的 shell，有些甚至还提供了多种 shell 供您选择。虽然它们之间有些细节上的差异，但是其核心功能都是一样的：它允许你执行程序，输入并获取

某种半结构化的输出。

本节课我们会使用 Bourne Again SHell, 简称 “bash” 。 这是被最广泛使用的一种 shell，它的语法和其他的 shell 都是类似的。打开shell *提示符*（您输入指令的地方），您首先需要打开 *终端* 。您的设备通常都已经内置了终端，或者您也可以安装一个，非常简单。

使用 shell

当您打开终端时，您会看到一个提示符，它看起来一般是这个样子的：

```
missing:~$
```

这是 shell 最主要的文本接口。它告诉你，你的主机名是 `missing` 并且您当前的工作目录（“current working directory”）或者说您当前所在的位置是 `~` (表示 “home”)。 `$` 符号表示您现在的身份不是 root 用户（稍后会介绍）。在这个提示符中，您可以输入 *命令* ，命令最终会被 shell 解析。最简单的命令是执行一个程序：

```
missing:~$ date
Fri 10 Jan 2020 11:49:31 AM EST
missing:~$
```

这里，我们执行了 `date` 这个程序，不出意料地，它打印出了当前的日前和时间。然后，shell 等待我们输入其他命令。我们可以在执行命令的同时向程序传递 *参数* ：

```
missing:~$ echo hello
hello
```

上例中，我们让 shell 执行 `echo` ，同时指定参数 `hello`。 `echo` 程序将该参数打印出来。shell 基于空格分割命令并进行解析，然后执行第一个单词代表的程序，并将后续的单词作为程序可以访问的参数。如果您希望传递的参数中包含空格（例如一个名为 My Photos 的文件夹），您要么用使用单引号，双引号将其包裹起来，要么使用转义符号 `\` 进行处理（`My\ Photos`）。

但是，shell 是如何知道去哪里寻找 `date` 或 `echo` 的呢? 其实，类似于 Python 或 Ruby，shell 是一个编程环境，所以它具备变量、条件、循环和函数（下一课进行讲解）。当你在 shell 中执行命令时，您实际上是在执行一段 shell 可以解释执行的简短代码。如果你要求 shell 执行某个指令，但是该指令并不是 shell 所了解的编程关键字，那么它会去咨询 *环境变量* `$PATH`，它会列出当 shell 接到某条指令时，进行程序搜索的路径：

```
missing:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
missing:~$ which echo
/bin/echo
missing:~$ /bin/echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

当我们执行 `echo` 命令时，shell 了解到需要执行 `echo` 这个程序，随后它便会在 `$PATH` 中搜索由 `:` 所分割的一系列目录，基于名字搜索该程序。当找到该程序时便执行（假定该文件是 *可执行程序*，后续课程将详细讲解）。确定某个程序名代表的是哪个具体的程序，可以使用 `which` 程序。我们也可以绕过 `$PATH`，通过直接指定需要执行的程序的路径来执行该程序

在shell中导航

shell 中的路径是一组被分割的目录，在 Linux 和 macOS 上使用 `/` 分割，而在Windows上是 `\`。路径 `/` 代表的是系统的根目录，所有的文件夹都包括在这个路径之下，在Windows上每个盘都有一个根目录（例如：`C:\`）。 我们假设您在学习本课程时使用的是 Linux 文件系统。如果某个路径以 `/` 开头，那么它是一个 *绝对路径*，其他的都是 *相对路径*。相对路径是指相对于当前工作目录的路径，当前工作目录可以使用 `pwd` 命令来获取。此外，切换目录需要使用 `cd` 命令。在路径中，`.` 表示的是当前目录，而 `..` 表示上级目录：

```
missing:~$ pwd
/home/missing
missing:~$ cd /home
missing:/home$ pwd
/home
missing:/home$ cd ..
missing:/$ pwd
/
missing:/$ cd ./home
missing:/home$ pwd
/home
missing:/home$ cd missing
missing:~$ pwd
/home/missing
missing:~$ ../../bin/echo hello
hello
```

pwd: present working directory

cd - (dash) will go to the previous directory

注意，shell 会实时显示当前的路径信息。您可以通过配置 shell 提示符来显示各种有用的信息，这一内容我们会在后面的课程中进行讨论。

一般来说，当我们运行一个程序时，如果我们没有指定路径，则该程序会在当前目录下执行。例如，我们常常会搜索文件，并在需要时创建文件。

为了查看指定目录下包含哪些文件，我们使用 `ls` 命令：

```
missing:~$ ls
missing:~$ cd ..
missing:/home$ ls
missing
missing:/home$ cd ..
missing:/$ ls
bin
boot
dev
etc
home
...
```

除非我们利用第一个参数指定目录，否则 `ls` 会打印当前目录下的文件。大多数的命令接受标记和选项（带有值的标记），它们以 `-` 开头，并可以改变程序的行为。通常，在执行程序时使用 `-h` 或 `--help` 标记可以打印帮助信息，以便了解有哪些可用的标记或选项。例如，`ls --help` 的输出如下：

```
  -l                         use a long listing format
```

```
missing:~$ ls -l /home
drwxr-xr-x 1 missing  users  4096 Jun 15  2019 missing
```

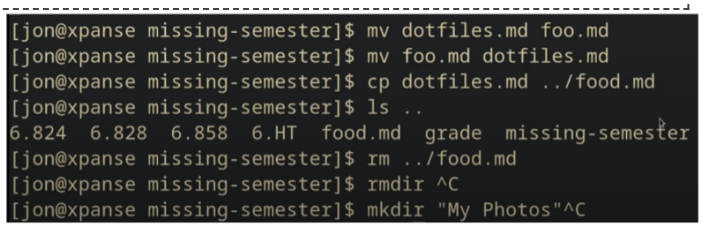
这个参数可以打印出更加详细地列出目录下文件或文件夹的信息。首先，本行第一个字符 `d` 表示 `missing` 是一个目录。然后接下来的九个字符，每三个字符构成一组。（`rwx`）.它们分别代表了文件所有者（`missing`），用户组（`users`）以及其他所有人具有的权限。其中 `-` 表示该用户不具备相应的权限。从上面的信息来看，只有文件所有者可以修改（`w`），`missing` 文件夹（例如，添加或删除文件夹中的文件）。为了进入某个文件夹，用户需要具备该文件夹以及其父文件夹的“搜索”权限（以“可执行”：`x`）权限表示。为了列出它的包含的内容，用户必须对该文件夹具备读权限（`r`）。对于文件来说，权限的意义也是类似的。注意，`/bin` 目录下的程序在最后一组，即表示所有人的用户组中，均包含 `x` 权限，也就是说任何人都可以执行这些程序。

在这个阶段，还有几个趁手的命令是您需要掌握的，例如 `mv`（用于重命名或移动文件）、`cp`（拷贝文件）以及 `mkdir`（新建文件夹）。

如果您想要知道关于程序参数、输入输出的信息，亦或是想要了解它们的工作方式，请试试 `man` 这个程序。它会接受一个程序名作为参数，然后将它的文档（用户手册）展现给您。注意，使用 `q` 可以退出该程序。

man(ual)，手册之意。

```
missing:~$ man ls
```



在程序间创建连接

在 shell 中，程序有两个主要的“流”：它们的输入流和输出流。 当程序尝试读取信息时，它们会从输入流中进行读取，当程序打印信息时，它们会将信息输出到输出流中。 通常，一个程序的输入输出流都是您的终端。也就是，您的键盘作为输入，显示器作为输出。 但是，我们也可以重定向这些流！

最简单的重定向是 `< file` 和 `> file`。这两个命令可以将程序的输入输出流分别重定向到文件：

```
missing:~$ echo hello > hello.txt
missing:~$ cat hello.txt
hello
missing:~$ cat < hello.txt
hello
missing:~$ cat < hello.txt > hello2.txt
missing:~$ cat hello2.txt
hello
```

您还可以使用 `>>` 来向一个文件追加内容。使用管道（*pipes*），我们能够更好的利用文件重定向。 `|` 操作符允许我们将一个程序的输出和另外一个程序的输入连接起来：

```
missing:~$ ls -l / | tail -n1
drwxr-xr-x 1 root  root  4096 Jun 20  2019 var
missing:~$ curl --head --silent google.com | grep --ignore-case content-l
219
```

我们会在数据清理一章中更加详细的探讨如何更好的利用管道。

一个功能全面又强大的工具

对于大多数的类 Unix 系统，有一类用户是非常特殊的，那就是：根用户（root user）。 您应该已经注意到了，在上面的输出结果中，根用户几乎不受任何限制，他可以创建、读取、更新和删除系统中的任何文件。 通常在我们并不会以根用户的身份直接登录系统，因为这样可能会因为某些错误的操作而破坏系统。 取而代之的是我们会在需要的时候使用 `sudo` 命令。顾名思义，它的作用是让您可以以 su（super user 或 root 的简写）的身份执行一些操作。 当您遇到拒绝访问（permission denied）的错误时，通常是因为此时您必须是根用户才能操作。然而，请再次确认您是真的要执行此操作。

有一件事情是您必须作为根用户才能做的，那就是向 `sysfs` 文件写入内容。系统被挂载在 `/sys` 下，`sysfs` 文件则暴露了一些内核（kernel）参数。 因此，您不需要借助任何专用的工具，就可以轻松地在运行期间配置系统内核。**注意 Windows 和 macOS 没有这个文件**

例如，您笔记本电脑的屏幕亮度写在 `brightness` 文件中，它位于

```
/sys/class/backlight
```

通过将数值写入该文件，我们可以改变屏幕的亮度。现在，蹦到您脑袋里的第一个想法可能是：

```
$ sudo find -L /sys/class/backlight -maxdepth 2 -name '*brightness*'
/sys/class/backlight/thinkpad_screen/brightness
$ cd /sys/class/backlight/thinkpad_screen
$ sudo echo 3 > brightness
An error occurred while redirecting file 'brightness'
open: Permission denied
```

出乎意料的是，我们还是得到了一个错误信息。毕竟，我们已经使用了 `sudo` 命令！关于 shell，有件事我们必须要知道。 `|` 、 `>` 、和 `<` 是通过 shell 执行的，而不是被各个程序单独执行。 `echo` 等程序并不知道 `|` 的存在，它们只知道从自己的输入输出流中进行读写。 对于上面这种情况， *shell* (权限为您的当前用户) 在设置 `sudo echo` 前尝试打开 brightness 文件并写入，但是系统拒绝了 shell 的操作因为此时 shell 不是根用户。

明白这一点后，我们可以这样操作：

```
$ echo 3 | sudo tee brightness
```

因为打开 `/sys` 文件的是 `tee` 这个程序，并且该程序以 `root` 权限在运行，因此操作可以进行。 这样您就可以在 `/sys` 中愉快地玩耍了，例如修改系统中各种LED的状态（路径可能会有所不同）：

```
$ echo 1 | sudo tee /sys/class/leds/input6::scrolllock/brightness
```

接下来.....

学到这里，您掌握的 shell 知识已经可以完成一些基础的任务了。您应该已经可以查找感兴趣的文件并使用大多数程序的基本功能了。 在下一场讲座中，我们会探讨如何利用 shell 及其他工具执行并自动化更复杂的任务。

课后练习

习题解答 本课程中的每节课都包含一系列练习题。有些题目是有明确目的的，另外一些则是开放题，例如“尝试使用 X 和 Y”，我们强烈建议您一定要动手实践，用于尝试这些内容。 此外，我们没有为这些练习题提供答案。如果有任何困难，您可以发送邮件给我们并描述你已经做出的尝试，我们会设法帮您解答。

1. 本课程需要使用类Unix shell，例如 Bash 或 ZSH。如果您在 Linux 或者 MacOS 上面完成本课程的练习，则不需要做任何特殊的操作。如果您使用的是 Windows，则您不应该使用 cmd 或是 Powershell；您可以使用Windows Subsystem for Linux或者是 Linux 虚拟机。使用 `echo $SHELL` 命令可以查看您的 shell 是否满足要求。如果打印结果为 `/bin/bash` 或 `/usr/bin/zsh` 则是可以的。
2. 在 `/tmp` 下新建一个名为 `missing` 的文件夹。

3. 用 `man` 查看程序 `touch` 的使用手册。
4. 用 `touch` 在 `missing` 文件夹中新建一个叫 `semester` 的文件。
5. 将以下内容一行一行地写入 `semester` 文件：

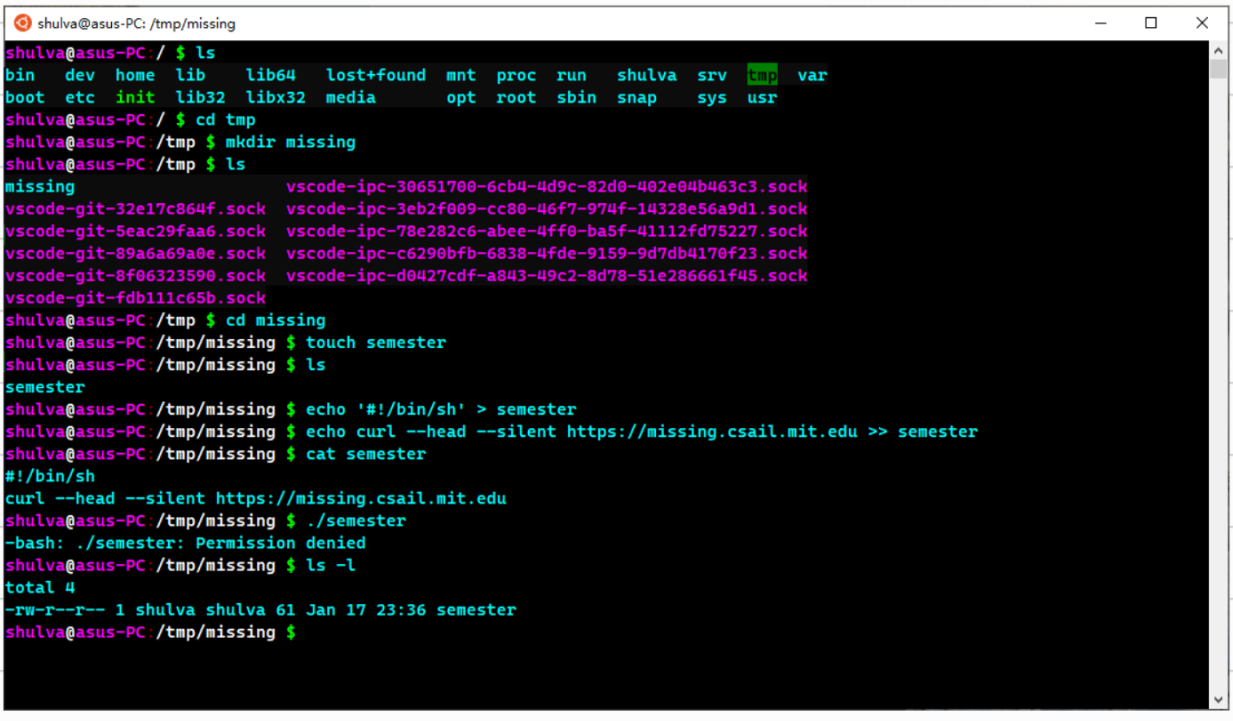
```
#!/bin/sh
curl --head --silent https://missing.csail.mit.edu
```

   第一行可能有点棘手， `#` 在Bash中表示注释，而 `!` 即使被双引号（`"`）包裹也具有特殊的含义。 单引号（`'`）则不一样，此处利用这一点解决输入问题。更多信息请参考 Bash quoting 手册

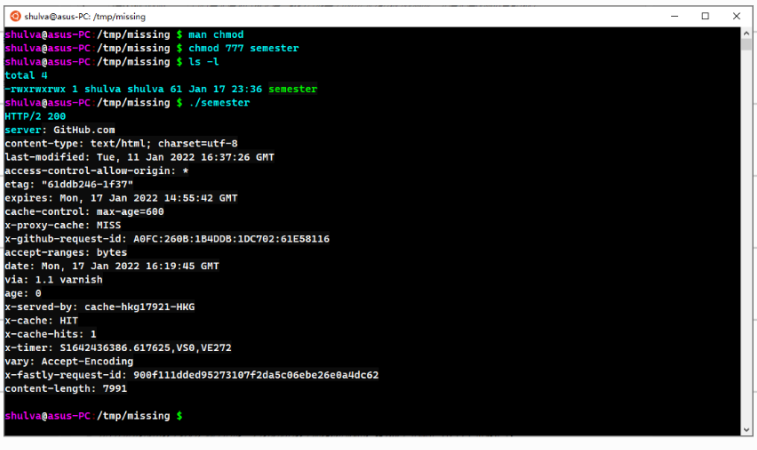
6. 尝试执行这个文件。例如，将该脚本的路径（`./semester`）输入到您的shell中并回车。如果程序无法执行，请使用 `ls` 命令来获取信息并理解其不能执行的原因。
7. 查看 `chmod` 的手册(例如，使用 `man chmod` 命令)
8. 使用 `chmod` 命令改变权限，使 `./semester` 能够成功执行，不要使用 `sh semester` 来执行该程序。您的 shell 是如何知晓这个文件需要使用 `sh` 来解析呢？更多信息请参考：shebang
9. 使用 `|` 和 `>`，将 `semester` 文件输出的最后更改日期信息，写入主目录下的 `last-modified.txt` 的文件中
10. 写一段命令来从 `/sys` 中获取笔记本的电量信息，或者台式机 CPU 的温度。注意：macOS 并没有 sysfs，所以 Mac 用户可以跳过这一题。

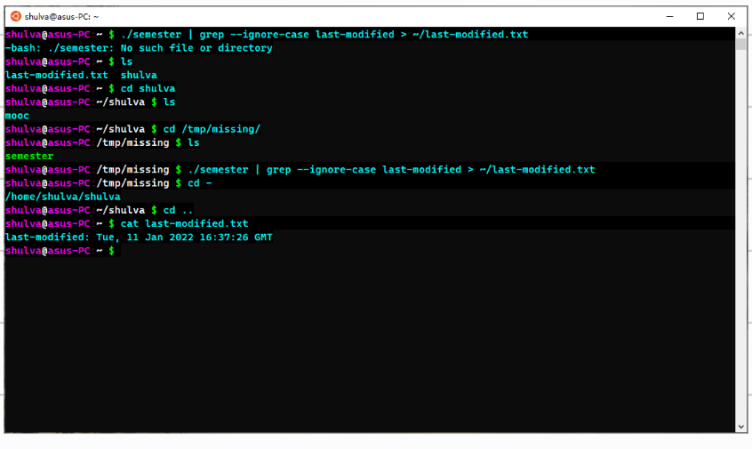
homework:

1~6



7~9





10.

由于我用的是 WSL 2 的 Linux，似乎没有 cpu 温度的文件模块，故作罢，其实不难。

Shell Tools and Scripting

In this lecture, we will present some of the basics of using bash as a scripting language along with a number of shell tools that cover several of the most common tasks that you will be constantly performing in the command line.

Shell Scripting

So far we have seen how to execute commands in the shell and pipe them together. However, in many scenarios you will want to perform a series of commands and make use of control flow expressions like conditionals or loops.

Shell scripts are the next step in complexity. Most shells have their own scripting language with variables, control flow and its own syntax. What makes shell scripting different from other scripting programming language is that it is optimized for performing shell-related tasks. Thus, creating command pipelines, saving results into files, and reading from standard input are primitives in shell scripting, which makes it easier to use than general purpose scripting languages. For this section we will focus on bash scripting since it is the most common.

To assign variables in bash, use the syntax `foo=bar` and access the value of the variable with `$foo`. Note that `foo = bar` will not work since it is interpreted as calling the `foo` program with arguments `=` and `bar`. In general, in shell scripts the space character will perform argument splitting. This behavior can be confusing to use at first, so always check for that.

Strings in bash can be defined with `'` and `"` delimiters, but they are not equivalent. Strings delimited with `'` are literal strings and will not substitute variable values whereas `"` delimited strings will.

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

As with most programming languages, bash supports control flow techniques including `if`, `case`, `while` and `for`. Similarly, `bash` has functions that take arguments and can operate with them. Here is an example of a function that creates a directory and `cd`s into it.

```
mcd () {
    mkdir -p "$1"
    cd "$1"
}
```

>> Source mcd.sh
mcd will be defined in the shell

Here `$1` is the first argument to the script/function. Unlike other scripting languages, bash uses a variety of special variables to refer to arguments, error codes, and other relevant variables. Below is a list of some of them. A more comprehensive list can be found here.

- `$0` - Name of the script
- `$1` to `$9` - Arguments to the script. `$1` is the first argument and so on.
- `$@` - All the arguments
- `$#` - Number of arguments
- `$?` - Return code of the previous command
- `$$` - Process identification number (PID) for the current script
- `!!` - Entire last command, including arguments. A common pattern is to execute a command only for it to fail due to missing permissions; you can quickly re-execute the command with sudo by doing `sudo !!`
- `$_` - Last argument from the last command. If you are in an interactive shell, you can also quickly get this value by typing `Esc` followed by `.` or `Alt+.`

Commands will often return output using `STDOUT`, errors through `STDERR`, and a Return Code to report errors in a more script-friendly manner. The return code or exit status is the way scripts/commands have to communicate how execution went. A value of 0 usually means everything went OK; anything different from 0 means an error occurred.

Exit codes can be used to conditionally execute commands using `&&` (and operator) and `||` (or operator), both of which are short-circuiting operators. Commands can also be separated within the same line using a semicolon `;`. The `true` program will always have a 0 return code and the `false` command will always have a 1 return code. Let's see some examples

```
false || echo "Oops, fail"
# Oops, fail

true || echo "Will not be printed"
#

true && echo "Things went well"
# Things went well

false && echo "Will not be printed"
#

true ; echo "This will always run"
# This will always run

false ; echo "This will always run"
# This will always run
```

Another common pattern is wanting to get the output of a command as a variable. This can be done with *command substitution*. Whenever you place `$( CMD )` it will execute `CMD`, get the output of the command and substitute it in place. For example, if you do `for file in $(ls)`, the shell will first call `ls` and then iterate over those values. A lesser known similar feature is *process substitution*, `<( CMD )` will execute `CMD` and place the output in a temporary file and substitute the `<()` with that file's name. This is useful when commands expect values to be passed by file instead of by STDIN. For example, `diff <(ls foo) <(ls bar)` will show differences between files in dirs `foo` and `bar`.

Since that was a huge information dump, let's see an example that showcases some of these features. It will iterate through the arguments we provide, `grep` for the string `foobar`, and append it to the file as a comment if it's not found.

```
#!/bin/bash

echo "Starting program at $(date)" # Date will be substituted

echo "Running program $0 with $# arguments with pid $$"

for file in "$@"; do
    grep foobar "$file" > /dev/null 2> /dev/null
    # When pattern is not found, grep has exit status 1
    # We redirect STDOUT and STDERR to a null register since we do not ca
    if [[ $? -ne 0 ]]; then
        echo "File $file does not have any foobar, adding one"
        echo "# foobar" >> "$file"
    fi
done
```

In the comparison we tested whether `$?` was not equal to 0. Bash implements many comparisons of this sort - you can find a detailed list in the manpage for `test`. When performing comparisons in bash, try to use double brackets `[[ ]]` in favor of simple brackets `[ ]`. Chances of making mistakes are lower although it won't be portable to `sh`. A more detailed explanation can be found here.

When launching scripts, you will often want to provide arguments that are similar. Bash has ways of making this easier, expanding expressions by carrying out filename expansion. These techniques are often referred to as shell *globbing*.

- Wildcards - Whenever you want to perform some sort of wildcard matching, you can use `?` and `*` to match one or any amount of characters respectively. For instance, given files `foo`, `foo1`, `foo2`, `foo10` and `bar`, the command `rm foo?` will delete `foo1` and `foo2` whereas `rm foo*` will delete all but `bar`.
- Curly braces `{}` - Whenever you have a common substring in a series of commands, you can use curly braces for bash to expand this automatically. This comes in very handy when moving or converting files.

```
convert image.{png,jpg}
# Will expand to
convert image.png image.jpg

cp /path/to/project/{foo,bar,baz}.sh /newpath
# Will expand to
cp /path/to/project/foo.sh /path/to/project/bar.sh /path/to/project/baz.s

# Globbing techniques can also be combined
mv *{.py,.sh} folder
# Will move all *.py and *.sh files


mkdir foo bar
# This creates files foo/a, foo/b, ... foo/h, bar/a, bar/b, ... bar/h
touch {foo,bar}/{a..h}
touch foo/x bar/y
# Show differences between files in foo and bar
diff <(ls foo) <(ls bar)
# Outputs
# < x
# ---
# > y
```

Writing `bash` scripts can be tricky and unintuitive. There are tools like shellcheck that will help you find errors in your sh/bash scripts.

Note that scripts need not necessarily be written in bash to be called from the terminal. For instance, here's a simple Python script that outputs its arguments in reversed order:

```
#!/usr/local/bin/python
import sys
for arg in reversed(sys.argv[1:]):
    print(arg)
```

The kernel knows to execute this script with a python interpreter instead of a shell command because we included a shebang line at the top of the script. It is good practice to write shebang lines using the `env` command that will resolve to wherever the command lives in the system, increasing the portability of your scripts. To resolve the location, `env` will make use of the `PATH` environment variable we introduced in the first lecture. For this example the shebang line would look like `#!/usr/bin/env python`.

Some differences between shell functions and scripts that you should keep in mind are:

- Functions have to be in the same language as the shell, while scripts can be written in any language. This is why including a shebang for scripts is important.
- Functions are loaded once when their definition is read. Scripts are loaded every time they are executed. This makes functions slightly faster to load, but whenever you change them you will have to reload their definition.
- Functions are executed in the current shell environment whereas scripts execute in their own process. Thus, functions can modify environment variables, e.g. change your current directory, whereas scripts can't. Scripts will be passed by value environment variables that have been exported using `export`
- As with any programming language, functions are a powerful construct to achieve modularity, code reuse, and clarity of shell code. Often shell scripts will include their own function definitions.

Shell Tools

Finding how to use commands

At this point, you might be wondering how to find the flags for the commands in the aliasing section such as `ls -l`, `mv -i` and `mkdir -p`. More generally, given a command, how do you go about finding out what it does and its different options? You could always start googling, but since UNIX predates StackOverflow, there are built-in ways of getting this information.

As we saw in the shell lecture, the first-order approach is to call said command with the `-h` or `--help` flags. A more detailed approach is to use the `man` command. Short for manual, `man` provides a manual page (called manpage) for a command you specify. For example, `man rm` will output the behavior of the `rm` command along with the flags that it takes, including the `-i` flag we showed earlier. In fact, what I have been linking so far for every command is the online version of the Linux manpages for the commands. Even non-native commands that you install will have manpage entries if the developer wrote them and included them as part of the installation process. For interactive tools such as the ones based on ncurses, help for the commands can often be accessed within the program using the `:help` command or typing `?`.

Sometimes manpages can provide overly detailed descriptions of the commands, making it hard to decipher what flags/syntax to use for common use cases. TLDR pages are a nifty complementary solution that focuses on giving example use cases of a command so you can quickly figure out which options to use. For instance, I find myself referring back to the tldr pages for `tar` and `ffmpeg` way more often than the manpages.

Finding files

One of the most common repetitive tasks that every programmer faces is finding files or directories. All UNIX-like systems come packaged with `find`, a great shell tool to find files. `find` will recursively search for files matching some criteria. Some examples:

```
# Find all directories named src
find . -name src -type d
# Find all python files that have a folder named test in their path
find . -path '*/test/*.py' -type f
# Find all files modified in the last day
find . -mtime -1
# Find all zip files with size in range 500k to 10M
find . -size +500k -size -10M -name '*.tar.gz'
```

Beyond listing files, find can also perform actions over files that match your query. This property can be incredibly helpful to simplify what could be fairly monotonous tasks.

```
# Delete all files with .tmp extension
find . -name '*.tmp' -exec rm {} \;
# Find all PNG files and convert them to JPG
find . -name '*.png' -exec convert {} {}.jpg \;
```

Despite `find`'s ubiquitousness, its syntax can sometimes be tricky to remember. For instance, to simply find files that match some pattern `PATTERN` you have to execute `find -name '*PATTERN*'` (or `-iname` if you want the pattern matching to be case insensitive). You could start building aliases for those scenarios, but part of the shell philosophy is that it is good to explore alternatives. Remember, one of the best properties of the shell is that you are just calling programs, so you can find (or even write yourself) replacements for some. For instance, `fd` is a simple, fast, and user-friendly alternative to `find`. It offers some nice defaults like colorized output, default regex matching, and Unicode support. It also has, in my opinion, a more intuitive syntax. For example, the syntax to find a pattern `PATTERN` is `fd PATTERN`.

Most would agree that `find` and `fd` are good, but some of you might be wondering about the efficiency of looking for files every time versus compiling some sort of index or database for quickly searching. That is what `locate` is for. `locate` uses a database that is updated using `updatedb`. In most systems, `updatedb` is updated daily via `cron`. Therefore one trade-off between the two is speed vs freshness. Moreover `find` and similar tools can also find files using attributes such as file size, modification time, or file permissions, while `locate` just uses the file name. A more in-depth comparison can be found here.

Finding code

Finding files by name is useful, but quite often you want to search based on file *content*. A common scenario is wanting to search for all files that contain some pattern, along with where in those files said pattern occurs. To achieve this, most UNIX-like systems provide `grep`, a generic tool for matching patterns from the input text. `grep` is an incredibly valuable shell tool that we will cover in greater detail during the data wrangling lecture.

For now, know that `grep` has many flags that make it a very versatile tool. Some I frequently use are `-C` for getting **C**ontext around the matching line and `-v` for in**v**erting the match, i.e. print all lines that do **not** match the pattern. For example, `grep -C 5` will print 5 lines before and after the match. When it comes to quickly searching through many files, you want to use `-R` since it will **R**ecursively go into directories and look for files for the matching string.

But `grep -R` can be improved in many ways, such as ignoring `.git` folders, using multi CPU support, &c. Many `grep` alternatives have been developed, including ack, ag and rg. All of them are fantastic and pretty much provide the same functionality. For now I am sticking with ripgrep (`rg`), given how fast and intuitive it is. Some examples:

```
# Find all python files where I used the requests library
rg -t py 'import requests'
# Find all files (including hidden files) without a shebang line
rg -u --files-without-match "^#!"
# Find all matches of foo and print the following 5 lines
rg foo -A 5
# Print statistics of matches (# of matched lines and files )
rg --stats PATTERN
```

Note that as with `find`/`fd`, it is important that you know that these problems can be quickly solved using one of these tools, while the specific tools you use are not as important.

Finding shell commands

So far we have seen how to find files and code, but as you start spending more time in the shell, you may want to find specific commands you typed at some point. The first thing to know is that typing the up arrow will give you back your last command, and if you keep pressing it you will slowly go through your shell history.

The `history` command will let you access your shell history programmatically. It will print your shell history to the standard output. If we want to search there we can pipe that output to `grep` and search for patterns. `history | grep find` will print commands that contain the substring "find".

In most shells, you can make use of `Ctrl+R` to perform backwards search through your history. After pressing `Ctrl+R`, you can type a substring you want to match for commands in your history. As you keep pressing it, you will cycle through the matches in your history. This can also be enabled with the UP/DOWN arrows in zsh. A nice addition on top of `Ctrl+R` comes with using fzf bindings. `fzf` is a general-purpose fuzzy finder that can be used with many commands. Here it is used to fuzzily match through your history and present results in a convenient and visually pleasing manner.

Another cool history-related trick I really enjoy is **history-based autosuggestions**. First introduced by the fish shell, this feature dynamically autocompletes your current shell command with the most recent command that you typed that shares a common prefix with it. It can be enabled in zsh and it is a great quality of life trick for your shell.

You can modify your shell's history behavior, like preventing commands with a leading space from being included. This comes in handy when you are typing commands with passwords or other bits of sensitive information. To do this, add `HISTCONTROL=ignorespace` to your `.bashrc` or `setopt HIST_IGNORE_SPACE` to your `.zshrc`. If you make the mistake of not adding the leading space, you can always manually remove the entry by editing your `.bash_history` or `.zhistory`.

Directory Navigation

So far, we have assumed that you are already where you need to be to perform these actions. But how do you go about quickly navigating directories? There are many simple ways that you could do this, such as writing shell aliases or creating symlinks with `ln -s`, but the truth is that developers have figured out quite clever and sophisticated solutions by now.

As with the theme of this course, you often want to optimize for the common case. Finding frequent and/or recent files and directories can be done through tools like `fasd` and `autojump`. Fasd ranks files and directories by *frecency*, that is, by both *frequency* and *recency*. By default, `fasd` adds a `z` command that you can use to quickly `cd` using a substring of a *frecent* directory. For example, if you often go to `/home/user/files/cool_project` you can simply use `z cool` to jump there. Using autojump, this same change of directory could be accomplished using `j cool`.

More complex tools exist to quickly get an overview of a directory structure: `tree`, `broot` or even full fledged file managers like `nnn` or `ranger`.

Exercises

1. Read `man ls` and write an `ls` command that lists files in the following manner
   - Includes all files, including hidden files
   - Sizes are listed in human readable format (e.g. 454M instead of 454279954)
   - Files are ordered by recency
   - Output is colorized

   A sample output would look like this

```
-rw-r--r--   1 user group 1.1M Jan 14 09:53 baz
drwxr-xr-x   5 user group  160 Jan 14 09:53 .
-rw-r--r--   1 user group  514 Jan 14 06:42 bar
-rw-r--r--   1 user group 106M Jan 13 12:12 foo
drwx------+ 47 user group 1.5K Jan 12 18:08 ..
```

2. Write bash functions `marco` and `polo` that do the following. Whenever you execute `marco` the current working directory should be saved in some manner, then when you execute `polo`, no matter what directory you are in, `polo` should `cd` you back to the directory where you executed `marco`. For ease of debugging you can write the code in a file `marco.sh` and (re)load the definitions to your shell by executing `source marco.sh`.

3. Say you have a command that fails rarely. In order to debug it you need to capture its output but it can be time consuming to get a failure run. Write a bash script that runs the following script until it fails and captures its standard output and error streams to files and prints everything at the end. Bonus points if you can also report how many runs it took for the script to fail.

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

n=$(( RANDOM % 100 ))

if [[ n -eq 42 ]]; then
   echo "Something went wrong"
   >&2 echo "The error was using magic numbers"
   exit 1
fi

echo "Everything went according to plan"
```

4. As we covered in the lecture `find`'s `-exec` can be very powerful for performing operations over the files we are searching for. However, what if we want to do something with **all** the files, like creating a zip file? As you have seen so far commands will take input from both arguments and STDIN. When piping commands, we are connecting STDOUT to STDIN, but some commands like `tar` take inputs from arguments. To bridge this disconnect there's the `xargs` command which will execute a command using STDIN as arguments. For example `ls | xargs rm` will delete the files in the current directory.

   Your task is to write a command that recursively finds all HTML files in the folder and makes a zip with them. Note that your command should work even if the files have spaces (hint: check `-d` flag for `xargs`).

If you're on macOS, note that the default BSD `find` is different from the one included in GNU coreutils. You can use `-print0` on `find` and the `-0` flag on `xargs`. As a macOS user, you should be aware that command-line utilities shipped with macOS may differ from the GNU counterparts; you can install the GNU versions if you like by using brew.

5. (Advanced) Write a command or script to recursively find the most recently modified file in a directory. More generally, can you list all files by recency?

Editors (Vim)

Writing English words and writing code are very different activities. When programming, you spend more time switching files, reading, navigating, and editing code compared to writing a long stream. It makes sense that there are different types of programs for writing English words versus code (e.g. Microsoft Word versus Visual Studio Code).

As programmers, we spend most of our time editing code, so it's worth investing time mastering an editor that fits your needs. Here's how you learn a new editor:

- Start with a tutorial (i.e. this lecture, plus resources that we point out)
- Stick with using the editor for all your text editing needs (even if it slows you down initially)
- Look things up as you go: if it seems like there should be a better way to do something, there probably is

If you follow the above method, fully committing to using the new program for all text editing purposes, the timeline for learning a sophisticated text editor looks like this. In an hour or two, you'll learn basic editor functions such as opening and editing files, save/quit, and navigating buffers. Once you're 20 hours in, you should be as fast as you were with your old editor. After that, the benefits start: you will have enough knowledge and muscle memory that using the new editor saves you time. Modern text editors are fancy and powerful tools, so the learning never stops: you'll get even faster as you learn more.

Which editor to learn?

Programmers have strong opinions about their text editors.

Which editors are popular today? See this Stack Overflow survey (there may be some bias because Stack Overflow users may not be representative of programmers as a whole). Visual Studio Code is the most popular editor. Vim is the most popular command-line-based editor.

Vim

All the instructors of this class use Vim as their editor. Vim has a rich history; it originated from the Vi editor (1976), and it's still being developed today. Vim has some really neat ideas behind it, and for this reason, lots of tools support a Vim emulation mode (for example, 1.4 million people have installed Vim emulation for VS code). Vim is probably worth learning even if you finally end up switching to some other text editor.

It's not possible to teach all of Vim's functionality in 50 minutes, so we're going to focus on explaining the philosophy of Vim, teaching you the basics, showing you some of

the more advanced functionality, and giving you the resources to master the tool.

Philosophy of Vim

When programming, you spend most of your time reading/editing, not writing. For this reason, Vim is a *modal* editor: it has different modes for inserting text vs manipulating text. Vim is programmable (with Vimscript and also other languages like Python), and Vim's interface itself is a programming language: keystrokes (with mnemonic names) are commands, and these commands are composable. Vim avoids the use of the mouse, because it's too slow; Vim even avoids using the arrow keys because it requires too much movement.

The end result is an editor that can match the speed at which you think.

Modal editing

Vim's design is based on the idea that a lot of programmer time is spent reading, navigating, and making small edits, as opposed to writing long streams of text. For this reason, Vim has multiple operating modes.

- **Normal**: for moving around a file and making edits
- **Insert**: for inserting text
- **Replace**: for replacing text
- **Visual** (plain, line, or block): for selecting blocks of text
- **Command-line**: for running a command

Keystrokes have different meanings in different operating modes. For example, the letter `x` in Insert mode will just insert a literal character 'x', but in Normal mode, it will delete the character under the cursor, and in Visual mode, it will delete the selection.

In its default configuration, Vim shows the current mode in the bottom left. The initial/default mode is Normal mode. You'll generally spend most of your time between Normal mode and Insert mode.

You change modes by pressing `<ESC>` (the escape key) to switch from any mode back to Normal mode. From Normal mode, enter Insert mode with `i`, Replace mode with `R`, Visual mode with `v`, Visual Line mode with `V`, Visual Block mode with `<C-v>` (Ctrl-V, sometimes also written `^V`), and Command-line mode with `:`.

You use the `<ESC>` key a lot when using Vim: consider remapping Caps Lock to Escape (macOS instructions).

Basics

Inserting text

From Normal mode, press `i` to enter Insert mode. Now, Vim behaves like any other text editor, until you press `<ESC>` to return to Normal mode. This, along with the basics explained above, are all you need to start editing files using Vim (though not particularly efficiently, if you're spending all your time editing from Insert mode).

Buffers, tabs, and windows

Vim maintains a set of open files, called "buffers". A Vim session has a number of tabs, each of which has a number of windows (split panes). Each window shows a single buffer. Unlike other programs you are familiar with, like web browsers, there is not a 1-to-1 correspondence between buffers and windows; windows are merely views. A given buffer may be open in *multiple* windows, even within the same tab. This can be quite handy, for example, to view two different parts of a file at the same time.

By default, Vim opens with a single tab, which contains a single window.

Command-line

Command mode can be entered by typing `:` in Normal mode. Your cursor will jump to the command line at the bottom of the screen upon pressing `:`. This mode has many functionalities, including opening, saving, and closing files, and quitting Vim.

- `:q` quit (close window)
- `:w` save ("write")
- `:wq` save and quit
- `:e {name of file}` open file for editing
- `:ls` show open buffers
- `:help {topic}` open help
  - `:help :w` opens help for the `:w` command
  - `:help w` opens help for the `w` movement

Vim's interface is a programming language

The most important idea in Vim is that Vim's interface itself is a programming language. Keystrokes (with mnemonic names) are commands, and these commands *compose*. This enables efficient movement and edits, especially once the commands become muscle memory.

Movement

You should spend most of your time in Normal mode, using movement commands to navigate the buffer. Movements in Vim are also called "nouns", because they refer to chunks of text.

- Basic movement: `hjkl` (left, down, up, right)
- Words: `w` (next word), `b` (beginning of word), `e` (end of word)

- Lines: `0` (beginning of line), `^` (first non-blank character), `$` (end of line)
- Screen: `H` (top of screen), `M` (middle of screen), `L` (bottom of screen)
- Scroll: `Ctrl-u` (up), `Ctrl-d` (down)
- File: `gg` (beginning of file), `G` (end of file)
- Line numbers: `:{number}<CR>` or `{number}G` (line {number})
- Misc: `%` (corresponding item)
- Find: `f{character}`, `t{character}`, `F{character}`, `T{character}`
  - find/to forward/backward {character} on the current line
  - `,` / `;` for navigating matches
- Search: `/{regex}`, `n` / `N` for navigating matches

Selection

Visual modes:

- Visual: `v`
- Visual Line: `V`
- Visual Block: `Ctrl-v`

Can use movement keys to make selection.

Edits

Everything that you used to do with the mouse, you now do with the keyboard using editing commands that compose with movement commands. Here's where Vim's interface starts to look like a programming language. Vim's editing commands are also called "verbs", because verbs act on nouns.

- `i` enter Insert mode
  - but for manipulating/deleting text, want to use something more than backspace
- `o` / `O` insert line below / above
- `d{motion}` delete {motion}
  - e.g. `dw` is delete word, `d$` is delete to end of line, `d0` is delete to beginning of line
- `c{motion}` change {motion}
  - e.g. `cw` is change word
  - like `d{motion}` followed by `i`
- `x` delete character (equal do `dl`)
- `s` substitute character (equal to `xi`)
- Visual mode + manipulation
  - select text, `d` to delete it or `c` to change it
- `u` to undo, `<C-r>` to redo
- `y` to copy / "yank" (some other commands like `d` also copy)
- `p` to paste
- Lots more to learn: e.g. `~` flips the case of a character

Counts

You can combine nouns and verbs with a count, which will perform a given action a number of times.

- `3w` move 3 words forward
- `5j` move 5 lines down
- `7dw` delete 7 words

Modifiers

You can use modifiers to change the meaning of a noun. Some modifiers are `i`, which means "inner" or "inside", and `a`, which means "around".

- `ci(` change the contents inside the current pair of parentheses
- `ci[` change the contents inside the current pair of square brackets
- `da'` delete a single-quoted string, including the surrounding single quotes

Demo

Here is a broken fizz buzz implementation:

```
def fizz_buzz(limit):
    for i in range(limit):
        if i % 3 == 0:
            print('fizz')
        if i % 5 == 0:
            print('fizz')
        if i % 3 and i % 5:
            print(i)

def main():
    fizz_buzz(10)
```

We will fix the following issues:

- Main is never called
- Starts at 0 instead of 1
- Prints "fizz" and "buzz" on separate lines for multiples of 15
- Prints "fizz" for multiples of 5
- Uses a hard-coded argument of 10 instead of taking a command-line argument

See the lecture video for the demonstration. Compare how the above changes are made using Vim to how you might make the same edits using another program. Notice how very few keystrokes are required in Vim, allowing you to edit at the speed you think.

Customizing Vim

Vim is customized through a plain-text configuration file in `~/.vimrc` (containing Vimscript commands). There are probably lots of basic settings that you want to turn on.

We are providing a well-documented basic config that you can use as a starting point. We recommend using this because it fixes some of Vim's quirky default behavior. **Download our config here and save it to `~/.vimrc`.**

Vim is heavily customizable, and it's worth spending time exploring customization options. You can look at people's dotfiles on GitHub for inspiration, for example, your instructors' Vim configs (Anish, Jon (uses neovim), Jose). There are lots of good blog posts on this topic too. Try not to copy-and-paste people's full configuration, but read it, understand it, and take what you need.

Extending Vim

There are tons of plugins for extending Vim. Contrary to outdated advice that you might find on the internet, you do *not* need to use a plugin manager for Vim (since Vim 8.0). Instead, you can use the built-in package management system. Simply create the directory `~/.vim/pack/vendor/start/`, and put plugins in there (e.g. via `git clone`).

Here are some of our favorite plugins:

- ctrlp.vim: fuzzy file finder
- ack.vim: code search
- nerdtree: file explorer
- vim-easymotion: magic motions

We're trying to avoid giving an overwhelmingly long list of plugins here. You can check out the instructors' dotfiles (Anish, Jon, Jose) to see what other plugins we use. Check out Vim Awesome for more awesome Vim plugins. There are also tons of blog posts on this topic: just search for "best Vim plugins".

Vim-mode in other programs

Many tools support Vim emulation. The quality varies from good to great; depending on the tool, it may not support the fancier Vim features, but most cover the basics pretty well.

Shell

If you're a Bash user, use `set -o vi`. If you use Zsh, `bindkey -v`. For Fish, `fish_vi_key_bindings`. Additionally, no matter what shell you use, you can `export EDITOR=vim`. This is the environment variable used to decide which editor is launched

when a program wants to start an editor. For example, `git` will use this editor for commit messages.

Readline

Many programs use the GNU Readline library for their command-line interface. Readline supports (basic) Vim emulation too, which can be enabled by adding the following line to the `~/.inputrc` file:

```
set editing-mode vi
```

With this setting, for example, the Python REPL will support Vim bindings.

Others

There are even vim keybinding extensions for web browsers - some popular ones are Vimium for Google Chrome and Tridactyl for Firefox. You can even get Vim bindings in Jupyter notebooks. Here is a long list of software with vim-like keybindings.

Advanced Vim

Here are a few examples to show you the power of the editor. We can't teach you all of these kinds of things, but you'll learn them as you go. A good heuristic: whenever you're using your editor and you think "there must be a better way of doing this", there probably is: look it up online.

Search and replace

`:s` (substitute) command (documentation).

- `%s/foo/bar/g`
  - replace foo with bar globally in file
- `%s/\[.*\](\(.*\))/\1/g`
  - replace named Markdown links with plain URLs

Multiple windows

- `:sp` / `:vsp` to split windows
- Can have multiple views of the same buffer.

Macros

- `q{character}` to start recording a macro in register `{character}`
- `q` to stop recording
- `@{character}` replays the macro
- Macro execution stops on error
- `{number}@{character}` executes a macro {number} times

- Macros can be recursive
  - first clear the macro with `q{character}q`
  - record the macro, with `@{character}` to invoke the macro recursively (will be a no-op until recording is complete)
- Example: convert xml to json (file)
  - Array of objects with keys "name" / "email"
  - Use a Python program?
  - Use sed / regexes
    - `g/people/d`
    - `%s/<person>/{/g`
    - `%s/<name>\(.*\)<\/name>/"name": "\1",/g`
    - ...
  - Vim commands / macros
    - `Gdd, ggdd` delete first and last lines
    - Macro to format a single element (register `e`)
      - Go to line with `<name>`
      - `qe^r"f>s": "<ESC>f<C"<ESC>q`
    - Macro to format a person
      - Go to line with `<person>`
      - `qpS{<ESC>j@eA,<ESC>j@ejS},<ESC>q`
    - Macro to format a person and go to the next person
      - Go to line with `<person>`
      - `qq@pjq`
    - Execute macro until end of file
      - `999@q`
    - Manually remove last `,` and add `[` and `]` delimiters

Resources

- `vimtutor` is a tutorial that comes installed with Vim - if Vim is installed, you should be able to run `vimtutor` from your shell
- Vim Adventures is a game to learn Vim
- Vim Tips Wiki
- Vim Advent Calendar has various Vim tips
- Vim Golf is code golf, but where the programming language is Vim's UI
- Vi/Vim Stack Exchange
- Vim Screencasts
- Practical Vim (book)

Exercises

1. Complete `vimtutor`. Note: it looks best in a 80x24 (80 columns by 24 lines) terminal window.

2. Download our basic vimrc and save it to `~/.vimrc`. Read through the well-commented file (using Vim!), and observe how Vim looks and behaves slightly differently with the new config.
3. Install and configure a plugin: ctrlp.vim.
   1. Create the plugins directory with `mkdir -p ~/.vim/pack/vendor/start`
   2. Download the plugin: `cd ~/.vim/pack/vendor/start; git clone https://github.com/ctrlpvim/ctrlp.vim`
   3. Read the documentation for the plugin. Try using CtrlP to locate a file by navigating to a project directory, opening Vim, and using the Vim command-line to start `:CtrlP`.
   4. Customize CtrlP by adding configuration to your `~/.vimrc` to open CtrlP by pressing Ctrl-P.
4. To practice using Vim, re-do the Demo from lecture on your own machine.
5. Use Vim for *all* your text editing for the next month. Whenever something seems inefficient, or when you think "there must be a better way", try Googling it, there probably is. If you get stuck, come to office hours or send us an email.
6. Configure your other tools to use Vim bindings (see instructions above).
7. Further customize your `~/.vimrc` and install more plugins.
8. (Advanced) Convert XML to JSON (example file) using Vim macros. Try to do this on your own, but you can look at the macros section above if you get stuck.

Data Wrangling

Have you ever wanted to take data in one format and turn it into a different format? Of course you have! That, in very general terms, is what this lecture is all about. Specifically, massaging data, whether in text or binary format, until you end up with exactly what you wanted.

We've already seen some basic data wrangling in past lectures. Pretty much any time you use the `|` operator, you are performing some kind of data wrangling. Consider a command like `journalctl | grep -i intel`. It finds all system log entries that mention Intel (case insensitive). You may not think of it as wrangling data, but it is going from one format (your entire system log) to a format that is more useful to you (just the intel log entries). Most data wrangling is about knowing what tools you have at your disposal, and how to combine them.

Let's start from the beginning. To wrangle data, we need two things: data to wrangle, and something to do with it. Logs often make for a good use-case, because you often want to investigate things about them, and reading the whole thing isn't feasible. Let's figure out who's trying to log into my server by looking at my server's log:

```
ssh myserver journalctl
```

That's far too much stuff. Let's limit it to ssh stuff:

```
ssh myserver journalctl | grep sshd
```

Notice that we're using a pipe to stream a *remote* file through `grep` on our local computer! `ssh` is magical, and we will talk more about it in the next lecture on the command-line environment. This is still way more stuff than we wanted though. And pretty hard to read. Let's do better:

```
ssh myserver 'journalctl | grep sshd | grep "Disconnected from"' | less
```

Why the additional quoting? Well, our logs may be quite large, and it's wasteful to stream it all to our computer and then do the filtering. Instead, we can do the filtering on the remote server, and then massage the data locally. `less` gives us a "pager" that allows us to scroll up and down through the long output. To save some additional traffic while we debug our command-line, we can even stick the current filtered logs into a file so that we don't have to access the network while developing:

```
$ ssh myserver 'journalctl | grep sshd | grep "Disconnected from"' > ssh.
$ less ssh.log
```

There's still a lot of noise here. There are *a lot* of ways to get rid of that, but let's look at one of the most powerful tools in your toolkit: `sed`.

`sed` is a "stream editor" that builds on top of the old `ed` editor. In it, you basically give short commands for how to modify the file, rather than manipulate its contents directly (although you can do that too). There are tons of commands, but one of the most common ones is `s`: substitution. For example, we can write:

```
ssh myserver journalctl
  | grep sshd
  | grep "Disconnected from"
  | sed 's/.*Disconnected from //'
```

What we just wrote was a simple *regular expression*; a powerful construct that lets you match text against patterns. The `s` command is written on the form: `s/REGEX/SUBSTITUTION/`, where `REGEX` is the regular expression you want to search for, and `SUBSTITUTION` is the text you want to substitute matching text with.

(You may recognize this syntax from the "Search and replace" section of our Vim lecture notes! Indeed, Vim uses a syntax for searching and replacing that is similar to `sed`'s substitution command. Learning one tool often helps you become more proficient with others.)

Regular expressions

Regular expressions are common and useful enough that it's worthwhile to take some time to understand how they work. Let's start by looking at the one we used above: `/.*Disconnected from /`. Regular expressions are usually (though not always) surrounded by `/`. Most ASCII characters just carry their normal meaning, but some characters have "special" matching behavior. Exactly which characters do what vary somewhat between different implementations of regular expressions, which is a source of great frustration. Very common patterns are:

- `.` means "any single character" except newline
- `*` zero or more of the preceding match
- `+` one or more of the preceding match
- `[abc]` any one character of `a`, `b`, and `c`
- `(RX1|RX2)` either something that matches `RX1` or `RX2`
- `^` the start of the line
- `$` the end of the line

`sed`'s regular expressions are somewhat weird, and will require you to put a `\` before most of these to give them their special meaning. Or you can pass `-E`.

So, looking back at `/.*Disconnected from /`, we see that it matches any text that starts with any number of characters, followed by the literal string "Disconnected from ". Which is what we wanted. But

beware, regular expressions are trixy. What if someone tried to log in with the username "Disconnected from"? We'd have:

```
Jan 17 03:13:00 thesquareplanet.com sshd[2631]: Disconnected from invalid
```

What would we end up with? Well, `*` and `+` are, by default, "greedy". They will match as much text as they can. So, in the above, we'd end up with just

```
46.97.239.16 port 55920 [preauth]
```

Which may not be what we wanted. In some regular expression implementations, you can just suffix `*` or `+` with a `?` to make them non-greedy, but sadly `sed` doesn't support that. We *could* switch to perl's command-line mode though, which *does* support that construct:

```
perl -pe 's/.*?Disconnected from //'
```

We'll stick to `sed` for the rest of this, because it's by far the more common tool for these kinds of jobs. `sed` can also do other handy things like print lines following a given match, do multiple substitutions per invocation, search for things, etc. But we won't cover that too much here. `sed` is basically an entire topic in and of itself, but there are often better tools.

Okay, so we also have a suffix we'd like to get rid of. How might we do that? It's a little tricky to match just the text that follows the username, especially if the username can have spaces and such! What we need to do is match the *whole* line:

```
  | sed -E 's/.*Disconnected from (invalid |authenticating )?user .* [^ ]+
```

Let's look at what's going on with a regex debugger. Okay, so the start is still as before. Then, we're matching any of the "user" variants (there are two prefixes in the logs). Then we're matching on any string of characters where the username is. Then we're matching on any single word (`[^ ]+`; any non-empty sequence of non-space characters). Then the word "port" followed by a sequence of digits. Then possibly the suffix `[preauth]`, and then the end of the line.

Notice that with this technique, as username of "Disconnected from" won't confuse us any more. Can you see why?

There is one problem with this though, and that is that the entire log becomes empty. We want to *keep* the username after all. For this, we can use "capture groups". Any text matched by a regex surrounded by parentheses is stored in a numbered capture group. These are available in the substitution (and in some engines, even in the pattern itself!) as `\1`, `\2`, `\3`, etc. So:

```
  | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^
```

As you can probably imagine, you can come up with *really* complicated regular expressions. For example, here's an article on how you might match an e-mail address. It's not easy. And there's lots of discussion. And people have written tests. And test matrices. You can even write a regex for determining if a given number is a prime number.

Regular expressions are notoriously hard to get right, but they are also very handy to have in your toolbox!

Back to data wrangling

Okay, so we now have

```
ssh myserver journalctl
  | grep sshd
  | grep "Disconnected from"
  | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^
```

`sed` can do all sorts of other interesting things, like injecting text (with the `i` command), explicitly printing lines (with the `p` command), selecting lines by index, and lots of other things. Check `man sed`!

Anyway. What we have now gives us a list of all the usernames that have attempted to log in. But this is pretty unhelpful. Let's look for common ones:

```
ssh myserver journalctl
  | grep sshd
  | grep "Disconnected from"
  | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^
  | sort | uniq -c
```

`sort` will, well, sort its input. `uniq -c` will collapse consecutive lines that are the same into a single line, prefixed with a count of the number of occurrences. We probably want to sort that too and only keep the most common usernames:

```
ssh myserver journalctl
  | grep sshd
  | grep "Disconnected from"
  | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^
  | sort | uniq -c
  | sort -nk1,1 | tail -n10
```

`sort -n` will sort in numeric (instead of lexicographic) order. `-k1,1` means "sort by only the first whitespace-separated column". The `,n` part says "sort until the `n`th field, where the default is the end of the line. In this *particular* example, sorting by the whole line wouldn't matter, but we're here to learn!

If we wanted the *least* common ones, we could use `head` instead of `tail`. There's also `sort -r`, which sorts in reverse order.

Okay, so that's pretty cool, but what if we'd like these extract only the usernames as a comma-separated list instead of one per line, perhaps for a config file?

```
ssh myserver journalctl
  | grep sshd
  | grep "Disconnected from"
  | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^
  | sort | uniq -c
  | sort -nk1,1 | tail -n10
  | awk '{print $2}' | paste -sd,
```

If you're using macOS: note that the command as shown won't work with the BSD `paste` shipped with macOS. See exercise 4 from the shell tools lecture for more on the difference between BSD and GNU coreutils and instructions for how to install GNU coreutils on macOS.

Let's start with `paste`: it lets you combine lines (`-s`) by a given single-character delimiter (`-d,` in this case). But what's this `awk` business?

awk – another editor

`awk` is a programming language that just happens to be really good at processing text streams. There is *a lot* to say about `awk` if you were to learn it properly, but as with many other things here, we'll just go through the basics.

First, what does `{print $2}` do? Well, `awk` programs take the form of an optional pattern plus a block saying what to do if the pattern matches a given line. The default pattern (which we used above) matches all lines. Inside the block, `$0` is set to the entire line's contents, and `$1` through `$n` are set to the `n`th *field* of that line, when separated by the `awk` field separator (whitespace by default, change with `-F`). In this case, we're saying that, for every line, print the contents of the second field, which happens to be the username!

Let's see if we can do something fancier. Let's compute the number of single-use usernames that start with `c` and end with `e`:

```
  | awk '$1 == 1 && $2 ~ /^c[^ ]*e$/ { print $2 }' | wc -l
```

There's a lot to unpack here. First, notice that we now have a pattern (the stuff that goes before `{...}`). The pattern says that the first field of the line should be equal to 1 (that's the count from `uniq -c`), and that the second field should match the given regular expression. And the block just says to print the username. We then count the number of lines in the output with `wc -l`.

However, `awk` is a programming language, remember?

```
BEGIN { rows = 0 }
$1 == 1 && $2 ~ /^c[^ ]*e$/ { rows += $1 }
END { print rows }
```

`BEGIN` is a pattern that matches the start of the input (and `END` matches the end). Now, the per-line block just adds the count from the first field (although it'll always be 1 in this case), and then we print it out at the end. In fact, we *could* get rid of `grep` and `sed` entirely, because `awk` can do it all, but we'll leave that as an exercise to the reader.

Analyzing data

You can do math directly in your shell using `bc`, a calculator that can read from STDIN! For example, add the numbers on each line together by concatenating them together, delimited by `+`:

```
  | paste -sd+ | bc -l
```

Or produce more elaborate expressions:

```
echo "2*($(data | paste -sd+))" | bc -l
```

You can get stats in a variety of ways. `st` is pretty neat, but if you already have R:

```
ssh myserver journalctl
  | grep sshd
  | grep "Disconnected from"
  | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^
  | sort | uniq -c
  | awk '{print $1}' | R --no-echo -e 'x <- scan(file="stdin", quiet=TRUE)
```

R is another (weird) programming language that's great at data analysis and plotting. We won't go into too much detail, but suffice to say that `summary` prints summary statistics for a vector, and we created a vector containing the input stream of numbers, so R gives us the statistics we wanted!

If you just want some simple plotting, `gnuplot` is your friend:

```
ssh myserver journalctl
  | grep sshd
  | grep "Disconnected from"
  | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^
  | sort | uniq -c
  | sort -nk1,1 | tail -n10
  | gnuplot -p -e 'set boxwidth 0.5; plot "-" using 1:xtic(2) with boxes'
```

Data wrangling to make arguments

Sometimes you want to do data wrangling to find things to install or remove based on some longer list. The data wrangling we've talked about so far + `xargs` can be a powerful combo.

For example, as seen in lecture, I can use the following command to uninstall old nightly builds of Rust from my system by extracting the old build names using data wrangling tools and then passing them via `xargs` to the uninstaller:

```
rustup toolchain list | grep nightly | grep -vE "nightly-x86" | sed 's/-x
```

Wrangling binary data

So far, we have mostly talked about wrangling textual data, but pipes are just as useful for binary data. For example, we can use ffmpeg to capture an image from our camera, convert it to grayscale, compress it, send it to a remote machine over SSH, decompress it there, make a copy, and then display it.

```
ffmpeg -loglevel panic -i /dev/video0 -frames 1 -f image2 -
 | convert - -colorspace gray -
 | gzip
 | ssh mymachine 'gzip -d | tee copy.jpg | env DISPLAY=:0 feh -'
```

Exercises

1. Take this short interactive regex tutorial.
2. Find the number of words (in `/usr/share/dict/words`) that contain at least three `a`s and don't have a `'s` ending. What are the three most common last two letters of those words? `sed`'s `y` command, or the `tr` program, may help you with case insensitivity. How many of those two-letter combinations are there? And for a challenge: which combinations do not occur?
3. To do in-place substitution it is quite tempting to do something like `sed s/REGEX/SUBSTITUTION/ input.txt > input.txt`. However this is a bad idea, why? Is this particular to `sed`? Use `man sed` to find out how to accomplish this.
4. Find your average, median, and max system boot time over the last ten boots. Use `journalctl` on Linux and `log show` on macOS, and look for log timestamps near the beginning and end of each boot. On Linux, they may look something like:

   ```
   Logs begin at ...
   ```

   and

   ```
   systemd[577]: Startup finished in ...
   ```

   On macOS, look for:

   ```
   === system boot:
   ```

   and

   ```
   Previous shutdown cause: 5
   ```

5. Look for boot messages that are *not* shared between your past three reboots (see `journalctl`'s `-b` flag). Break this task down into multiple steps. First, find a way to get just the logs from the past three boots. There may be an applicable flag on the tool you use to extract the boot logs, or you can use `sed '0,/STRING/d'` to remove all lines previous to one that matches `STRING`. Next, remove any parts of the line that *always* varies (like the timestamp). Then, de-duplicate the input lines and keep a count of each one (`uniq` is your friend). And finally, eliminate any line whose count is 3 (since it *was* shared among all the boots).

6. Find an online data set like this one, this one, or maybe one from here. Fetch it using `curl` and extract out just two columns of numerical data. If you're fetching HTML data, `pup` might be helpful. For JSON data, try `jq`. Find the min and max of one column in a single command, and the difference of the sum of each column in another.

Command-line Environment

In this lecture we will go through several ways in which you can improve your workflow when using the shell. We have been working with the shell for a while now, but we have mainly focused on executing different commands. We will now see how to run several processes at the same time while keeping track of them, how to stop or pause a specific process and how to make a process run in the background.

We will also learn about different ways to improve your shell and other tools, by defining aliases and configuring them using dotfiles. Both of these can help you save time, e.g. by using the same configurations in all your machines without having to type long commands. We will look at how to work with remote machines using SSH.

Job Control

In some cases you will need to interrupt a job while it is executing, for instance if a command is taking too long to complete (such as a `find` with a very large directory structure to search through). Most of the time, you can do `Ctrl-C` and the command will stop. But how does this actually work and why does it sometimes fail to stop the process?

Killing a process

Your shell is using a UNIX communication mechanism called a *signal* to communicate information to the process. When a process receives a signal it stops its execution, deals with the signal and potentially changes the flow of execution based on the information that the signal delivered. For this reason, signals are *software interrupts*.

In our case, when typing `Ctrl-C` this prompts the shell to deliver a `SIGINT` signal to the process.

Here's a minimal example of a Python program that captures `SIGINT` and ignores it, no longer stopping. To kill this program we can now use the `SIGQUIT` signal instead, by typing `Ctrl-\`.

```python
#!/usr/bin/env python
import signal, time

def handler(signum, time):
    print("\nI got a SIGINT, but I am not stopping")

signal.signal(signal.SIGINT, handler)
i = 0
while True:
    time.sleep(.1)
    print("\r{}".format(i), end="")
    i += 1
```

Here's what happens if we send `SIGINT` twice to this program, followed by `SIGQUIT`. Note that `^` is how `Ctrl` is displayed when typed in the terminal.

```
$ python sigint.py
24^C
I got a SIGINT, but I am not stopping
26^C
I got a SIGINT, but I am not stopping
30^\[1]    39913 quit       python sigint.py
```

While `SIGINT` and `SIGQUIT` are both usually associated with terminal related requests, a more generic signal for asking a process to exit gracefully is the `SIGTERM` signal. To send this signal we can use the `kill` command, with the syntax `kill -TERM <PID>`.

Pausing and backgrounding processes

Signals can do other things beyond killing a process. For instance, `SIGSTOP` pauses a process. In the terminal, typing `Ctrl-Z` will prompt the shell to send a `SIGTSTP` signal, short for Terminal Stop (i.e. the terminal's version of `SIGSTOP`).

We can then continue the paused job in the foreground or in the background using `fg` or `bg`, respectively.

The `jobs` command lists the unfinished jobs associated with the current terminal session. You can refer to those jobs using their pid (you can use `pgrep` to find that out). More intuitively, you can also refer to a process using the percent symbol followed by its job number (displayed by `jobs`). To refer to the last backgrounded job you can use the `$!` special parameter.

One more thing to know is that the `&` suffix in a command will run the command in the background, giving you the prompt back, although it will still use the shell's STDOUT which can be annoying (use shell redirections in that case).

To background an already running program you can do `Ctrl-Z` followed by `bg`. Note that backgrounded processes are still children processes of your terminal and will die if you close the terminal (this will send yet another signal, `SIGHUP`). To prevent that from happening you can run the program with `nohup` (a wrapper to ignore `SIGHUP`), or use `disown` if the process has already been started. Alternatively, you can use a terminal multiplexer as we will see in the next section.

Below is a sample session to showcase some of these concepts.

```
$ sleep 1000
^Z
[1]  + 18653 suspended  sleep 1000

$ nohup sleep 2000 &
[2] 18745
appending output to nohup.out

$ jobs
[1]  + suspended  sleep 1000
[2]  - running    nohup sleep 2000

$ bg %1
[1]  - 18653 continued  sleep 1000

$ jobs
[1]  - running    sleep 1000
[2]  + running    nohup sleep 2000

$ kill -STOP %1
[1]  + 18653 suspended (signal)  sleep 1000

$ jobs
[1]  + suspended (signal)  sleep 1000
[2]  - running    nohup sleep 2000

$ kill -SIGHUP %1
[1]  + 18653 hangup     sleep 1000

$ jobs
[2]  + running    nohup sleep 2000

$ kill -SIGHUP %2

$ jobs
[2]  + running    nohup sleep 2000

$ kill %2
[2]  + 18745 terminated  nohup sleep 2000

$ jobs
```

A special signal is `SIGKILL` since it cannot be captured by the process and it will always terminate it immediately. However, it can have bad side effects such as leaving orphaned children processes.

You can learn more about these and other signals here or typing `man signal` or `kill -l`.

Terminal Multiplexers

When using the command line interface you will often want to run more than one thing at once. For instance, you might want to run your editor and your program side by side. Although this can be achieved by opening new terminal windows, using a terminal multiplexer is a more versatile solution.

Terminal multiplexers like `tmux` allow you to multiplex terminal windows using panes and tabs so you can interact with multiple shell sessions. Moreover, terminal multiplexers let you detach a current terminal session and reattach at some point later in time. This can make your workflow much better when working with remote machines since it avoids the need to use `nohup` and similar tricks.

The most popular terminal multiplexer these days is `tmux`. `tmux` is highly configurable and by using the associated keybindings you can create multiple tabs and panes and quickly navigate through them.

`tmux` expects you to know its keybindings, and they all have the form `<C-b> x` where that means (1) press `Ctrl+b`, (2) release `Ctrl+b`, and then (3) press `x`. `tmux` has the following hierarchy of objects:

- **Sessions** - a session is an independent workspace with one or more windows
  - `tmux` starts a new session.
  - `tmux new -s NAME` starts it with that name.
  - `tmux ls` lists the current sessions
  - Within `tmux` typing `<C-b> d` detaches the current session
  - `tmux a` attaches the last session. You can use `-t` flag to specify which
- **Windows** - Equivalent to tabs in editors or browsers, they are visually separate parts of the same session
  - `<C-b> c` Creates a new window. To close it you can just terminate the shells doing `<C-d>`
  - `<C-b> N` Go to the *N* th window. Note they are numbered
  - `<C-b> p` Goes to the previous window
  - `<C-b> n` Goes to the next window
  - `<C-b> ,` Rename the current window
  - `<C-b> w` List current windows
- **Panes** - Like vim splits, panes let you have multiple shells in the same visual display.
  - `<C-b> "` Split the current pane horizontally
  - `<C-b> %` Split the current pane vertically
  - `<C-b> <direction>` Move to the pane in the specified *direction*. Direction here means arrow keys.
  - `<C-b> z` Toggle zoom for the current pane

- `<C-b> [` Start scrollback. You can then press `<space>` to start a selection and `<enter>` to copy that selection.
- `<C-b> <space>` Cycle through pane arrangements.

For further reading, here is a quick tutorial on `tmux` and this has a more detailed explanation that covers the original `screen` command. You might also want to familiarize yourself with `screen`, since it comes installed in most UNIX systems.

Aliases

It can become tiresome typing long commands that involve many flags or verbose options. For this reason, most shells support *aliasing*. A shell alias is a short form for another command that your shell will replace automatically for you. For instance, an alias in bash has the following structure:

```
alias alias_name="command_to_alias arg1 arg2"
```

Note that there is no space around the equal sign `=`, because `alias` is a shell command that takes a single argument.

Aliases have many convenient features:

```
# Make shorthands for common flags
alias ll="ls -lh"

# Save a lot of typing for common commands
alias gs="git status"
alias gc="git commit"
alias v="vim"

# Save you from mistyping
alias sl=ls

# Overwrite existing commands for better defaults
alias mv="mv -i"           # -i prompts before overwrite
alias mkdir="mkdir -p"     # -p make parent dirs as needed
alias df="df -h"           # -h prints human readable format

# Alias can be composed
alias la="ls -A"
alias lla="la -l"

# To ignore an alias run it prepended with \
\ls
# Or disable an alias altogether with unalias
unalias la

# To get an alias definition just call it with alias
alias ll
# Will print ll='ls -lh'
```

Note that aliases do not persist shell sessions by default. To make an alias persistent you need to include it in shell startup files, like `.bashrc` or `.zshrc`, which we are going to introduce in the next section.

Dotfiles

Many programs are configured using plain-text files known as *dotfiles* (because the file names begin with a `.`, e.g. `~/.vimrc`, so that they are hidden in the directory listing `ls` by default).

Shells are one example of programs configured with such files. On startup, your shell will read many files to load its configuration. Depending on the shell, whether you are starting a login and/or interactive the entire process can be quite complex. Here is an excellent resource on the topic.

For `bash`, editing your `.bashrc` or `.bash_profile` will work in most systems. Here you can include commands that you want to run on startup, like the alias we just

described or modifications to your `PATH` environment variable. In fact, many programs will ask you to include a line like `export PATH="$PATH:/path/to/program/bin"` in your shell configuration file so their binaries can be found.

Some other examples of tools that can be configured through dotfiles are:

- `bash - ~/.bashrc, ~/.bash_profile`
- `git - ~/.gitconfig`
- `vim - ~/.vimrc` and the `~/.vim` folder
- `ssh - ~/.ssh/config`
- `tmux - ~/.tmux.conf`

How should you organize your dotfiles? They should be in their own folder, under version control, and **symlinked** into place using a script. This has the benefits of:

- **Easy installation**: if you log in to a new machine, applying your customizations will only take a minute.
- **Portability**: your tools will work the same way everywhere.
- **Synchronization**: you can update your dotfiles anywhere and keep them all in sync.
- **Change tracking**: you're probably going to be maintaining your dotfiles for your entire programming career, and version history is nice to have for long-lived projects.

What should you put in your dotfiles? You can learn about your tool's settings by reading online documentation or man pages. Another great way is to search the internet for blog posts about specific programs, where authors will tell you about their preferred customizations. Yet another way to learn about customizations is to look through other people's dotfiles: you can find tons of dotfiles repositories on Github — see the most popular one here (we advise you not to blindly copy configurations though). Here is another good resource on the topic.

All of the class instructors have their dotfiles publicly accessible on GitHub: Anish, Jon, Jose.

Portability

A common pain with dotfiles is that the configurations might not work when working with several machines, e.g. if they have different operating systems or shells. Sometimes you also want some configuration to be applied only in a given machine.

There are some tricks for making this easier. If the configuration file supports it, use the equivalent of if-statements to apply machine specific customizations. For example, your shell could have something like:

```
if [[ "$(uname)" == "Linux" ]]; then {do_something}; fi

# Check before using shell-specific features
if [[ "$SHELL" == "zsh" ]]; then {do_something}; fi

# You can also make it machine-specific
if [[ "$(hostname)" == "myServer" ]]; then {do_something}; fi
```

If the configuration file supports it, make use of includes. For example, a `~/.gitconfig` can have a setting:

```
[include]
    path = ~/.gitconfig_local
```

And then on each machine, `~/.gitconfig_local` can contain machine-specific settings. You could even track these in a separate repository for machine-specific settings.

This idea is also useful if you want different programs to share some configurations. For instance, if you want both `bash` and `zsh` to share the same set of aliases you can write them under `.aliases` and have the following block in both:

```
# Test if ~/.aliases exists and source it
if [ -f ~/.aliases ]; then
    source ~/.aliases
fi
```

Remote Machines

It has become more and more common for programmers to use remote servers in their everyday work. If you need to use remote servers in order to deploy backend software or you need a server with higher computational capabilities, you will end up using a Secure Shell (SSH). As with most tools covered, SSH is highly configurable so it is worth learning about it.

To `ssh` into a server you execute a command as follows

```
ssh foo@bar.mit.edu
```

Here we are trying to ssh as user `foo` in server `bar.mit.edu`. The server can be specified with a URL (like `bar.mit.edu`) or an IP (something like `foobar@192.168.1.42`). Later we will see that if we modify ssh config file you can access just using something like `ssh bar`.

Executing commands

An often overlooked feature of `ssh` is the ability to run commands directly. `ssh foobar@server ls` will execute `ls` in the home folder of foobar. It works with pipes, so `ssh foobar@server ls | grep PATTERN` will grep locally the remote output of `ls` and `ls | ssh foobar@server grep PATTERN` will grep remotely the local output of `ls`.

SSH Keys

Key-based authentication exploits public-key cryptography to prove to the server that the client owns the secret private key without revealing the key. This way you do not need to reenter your password every time. Nevertheless, the private key (often `~/.ssh/id_rsa` and more recently `~/.ssh/id_ed25519`) is effectively your password, so treat it like so.

Key generation

To generate a pair you can run `ssh-keygen`.

```
ssh-keygen -o -a 100 -t ed25519 -f ~/.ssh/id_ed25519
```

You should choose a passphrase, to avoid someone who gets hold of your private key to access authorized servers. Use `ssh-agent` or `gpg-agent` so you do not have to type your passphrase every time.

If you have ever configured pushing to GitHub using SSH keys, then you have probably done the steps outlined here and have a valid key pair already. To check if you have a passphrase and validate it you can run `ssh-keygen -y -f /path/to/key`.

Key based authentication

`ssh` will look into `.ssh/authorized_keys` to determine which clients it should let in. To copy a public key over you can use:

```
cat .ssh/id_ed25519.pub | ssh foobar@remote 'cat >> ~/.ssh/authorized_key
```

A simpler solution can be achieved with `ssh-copy-id` where available:

```
ssh-copy-id -i .ssh/id_ed25519.pub foobar@remote
```

Copying files over SSH

There are many ways to copy files over ssh:

- `ssh+tee`, the simplest is to use `ssh` command execution and STDIN input by doing `cat localfile | ssh remote_server tee serverfile`. Recall that `tee` writes the output from STDIN into a file.
- `scp` when copying large amounts of files/directories, the secure copy `scp` command is more convenient since it can easily recurse over paths. The syntax is `scp path/to/local_file remote_host:path/to/remote_file`
- `rsync` improves upon `scp` by detecting identical files in local and remote, and preventing copying them again. It also provides more fine grained control over symlinks, permissions and has extra features like the `--partial` flag that can resume from a previously interrupted copy. `rsync` has a similar syntax to `scp`.

Port Forwarding

In many scenarios you will run into software that listens to specific ports in the machine. When this happens in your local machine you can type `localhost:PORT` or `127.0.0.1:PORT`, but what do you do with a remote server that does not have its ports directly available through the network/internet?.

This is called *port forwarding* and it comes in two flavors: Local Port Forwarding and Remote Port Forwarding (see the pictures for more details, credit of the pictures from this StackOverflow post).

Local Port Forwarding

Remote Port Forwarding

The most common scenario is local port forwarding, where a service in the remote machine listens in a port and you want to link a port in your local machine to forward to the remote port. For example, if we execute `jupyter notebook` in the remote server that listens to the port `8888`. Thus, to forward that to the local port `9999`, we would do `ssh -L 9999:localhost:8888 foobar@remote_server` and then navigate to `locahost:9999` in our local machine.

SSH Configuration

We have covered many many arguments that we can pass. A tempting alternative is to create shell aliases that look like

```
alias my_server="ssh -i ~/.id_ed25519 --port 2222 -L 9999:localhost:8888
```

However, there is a better alternative using `~/.ssh/config`.

```
Host vm
    User foobar
    HostName 172.16.174.141
    Port 2222
    IdentityFile ~/.ssh/id_ed25519
    LocalForward 9999 localhost:8888

# Configs can also take wildcards
Host *.mit.edu
    User foobaz
```

An additional advantage of using the `~/.ssh/config` file over aliases is that other programs like `scp`, `rsync`, `mosh`, &c are able to read it as well and convert the settings into the corresponding flags.

Note that the `~/.ssh/config` file can be considered a dotfile, and in general it is fine for it to be included with the rest of your dotfiles. However, if you make it public, think about the information that you are potentially providing strangers on the internet: addresses of your servers, users, open ports, &c. This may facilitate some types of attacks so be thoughtful about sharing your SSH configuration.

Server side configuration is usually specified in `/etc/ssh/sshd_config`. Here you can make changes like disabling password authentication, changing ssh ports, enabling X11 forwarding, &c. You can specify config settings on a per user basis.

Miscellaneous

A common pain when connecting to a remote server are disconnections due to shutting down/sleeping your computer or changing a network. Moreover if one has a connection with significant lag using ssh can become quite frustrating. Mosh, the mobile shell, improves upon ssh, allowing roaming connections, intermittent connectivity and providing intelligent local echo.

Sometimes it is convenient to mount a remote folder. sshfs can mount a folder on a remote server locally, and then you can use a local editor.

Shells & Frameworks

During shell tool and scripting we covered the `bash` shell because it is by far the most ubiquitous shell and most systems have it as the default option. Nevertheless, it is not the only option.

For example, the `zsh` shell is a superset of `bash` and provides many convenient features out of the box such as:

– Smarter globbing, `**`

- Inline globbing/wildcard expansion
- Spelling correction
- Better tab completion/selection
- Path expansion (`cd /u/lo/b` will expand as `/usr/local/bin`)

Frameworks can improve your shell as well. Some popular general frameworks are prezto or oh-my-zsh, and smaller ones that focus on specific features such as zsh-syntax-highlighting or zsh-history-substring-search. Shells like fish include many of these user-friendly features by default. Some of these features include:

- Right prompt
- Command syntax highlighting
- History substring search
- manpage based flag completions
- Smarter autocompletion
- Prompt themes

One thing to note when using these frameworks is that they may slow down your shell, especially if the code they run is not properly optimized or it is too much code. You can always profile it and disable the features that you do not use often or value over speed.

Terminal Emulators

Along with customizing your shell, it is worth spending some time figuring out your choice of **terminal emulator** and its settings. There are many many terminal emulators out there (here is a comparison).

Since you might be spending hundreds to thousands of hours in your terminal it pays off to look into its settings. Some of the aspects that you may want to modify in your terminal include:

- Font choice
- Color Scheme
- Keyboard shortcuts
- Tab/Pane support
- Scrollback configuration
- Performance (some newer terminals like Alacritty or kitty offer GPU acceleration).

Exercises

Job control

1. From what we have seen, we can use some `ps aux | grep` commands to get our jobs' pids and then kill them, but there are better ways to do it. Start a `sleep 10000` job in a terminal, background it with `Ctrl-Z` and continue its execution with `bg`. Now

use `pgrep` to find its pid and `pkill` to kill it without ever typing the pid itself. (Hint: use the `-af` flags).

2. Say you don't want to start a process until another completes. How would you go about it? In this exercise, our limiting process will always be `sleep 60 &`. One way to achieve this is to use the `wait` command. Try launching the sleep command and having an `ls` wait until the background process finishes.

   However, this strategy will fail if we start in a different bash session, since `wait` only works for child processes. One feature we did not discuss in the notes is that the `kill` command's exit status will be zero on success and nonzero otherwise. `kill -0` does not send a signal but will give a nonzero exit status if the process does not exist. Write a bash function called `pidwait` that takes a pid and waits until the given process completes. You should use `sleep` to avoid wasting CPU unnecessarily.

Terminal multiplexer

1. Follow this `tmux` tutorial and then learn how to do some basic customizations following these steps.

Aliases

1. Create an alias `dc` that resolves to `cd` for when you type it wrongly.

2. Run `history | awk '{$1="";print substr($0,2)}' | sort | uniq -c | sort -n | tail -n 10` to get your top 10 most used commands and consider writing shorter aliases for them. Note: this works for Bash; if you're using ZSH, use `history 1` instead of just `history`.

Dotfiles

Let's get you up to speed with dotfiles.

1. Create a folder for your dotfiles and set up version control.
2. Add a configuration for at least one program, e.g. your shell, with some customization (to start off, it can be something as simple as customizing your shell prompt by setting `$PS1`).
3. Set up a method to install your dotfiles quickly (and without manual effort) on a new machine. This can be as simple as a shell script that calls `ln -s` for each file, or you could use a specialized utility.
4. Test your installation script on a fresh virtual machine.
5. Migrate all of your current tool configurations to your dotfiles repository.
6. Publish your dotfiles on GitHub.

Remote Machines

Install a Linux virtual machine (or use an already existing one) for this exercise. If you are not familiar with virtual machines check out this tutorial for installing one.

1. Go to `~/.ssh/` and check if you have a pair of SSH keys there. If not, generate them with `ssh-keygen -o -a 100 -t ed25519`. It is recommended that you use a password and use `ssh-agent`, more info here.
2. Edit `.ssh/config` to have an entry as follows

```
Host vm
    User username_goes_here
    HostName ip_goes_here
    IdentityFile ~/.ssh/id_ed25519
    LocalForward 9999 localhost:8888
```

3. Use `ssh-copy-id vm` to copy your ssh key to the server.
4. Start a webserver in your VM by executing `python -m http.server 8888`. Access the VM webserver by navigating to `http://localhost:9999` in your machine.
5. Edit your SSH server config by doing `sudo vim /etc/ssh/sshd_config` and disable password authentication by editing the value of `PasswordAuthentication`. Disable root login by editing the value of `PermitRootLogin`. Restart the `ssh` service with `sudo service sshd restart`. Try sshing in again.
6. (Challenge) Install `mosh` in the VM and establish a connection. Then disconnect the network adapter of the server/VM. Can mosh properly recover from it?
7. (Challenge) Look into what the `-N` and `-f` flags do in `ssh` and figure out a command to achieve background port forwarding.

Version Control (Git)

Version control systems (VCSs) are tools used to track changes to source code (or other collections of files and folders). As the name implies, these tools help maintain a history of changes; furthermore, they facilitate collaboration. VCSs track changes to a folder and its contents in a series of snapshots, where each snapshot encapsulates the entire state of files/folders within a top-level directory. VCSs also maintain metadata like who created each snapshot, messages associated with each snapshot, and so on.

Why is version control useful? Even when you're working by yourself, it can let you look at old snapshots of a project, keep a log of why certain changes were made, work on parallel branches of development, and much more. When working with others, it's an invaluable tool for seeing what other people have changed, as well as resolving conflicts in concurrent development.

Modern VCSs also let you easily (and often automatically) answer questions like:

- Who wrote this module?
- When was this particular line of this particular file edited? By whom? Why was it edited?
- Over the last 1000 revisions, when/why did a particular unit test stop working?

While other VCSs exist, **Git** is the de facto standard for version control. This XKCD comic captures Git's reputation:

Because Git's interface is a leaky abstraction, learning Git top-down (starting with its interface / command-line interface) can lead to a lot of confusion. It's possible to memorize a handful of commands and think of them as magic incantations, and follow the approach in the comic above whenever anything goes wrong.

While Git admittedly has an ugly interface, its underlying design and ideas are beautiful. While an ugly interface has to be *memorized*, a beautiful design can be *understood*. For this reason, we give a bottom-up explanation of Git, starting with its data model and later covering the command-line interface. Once the data model is understood, the commands can be better understood in terms of how they manipulate the underlying data model.

Git's data model

There are many ad-hoc approaches you could take to version control. Git has a well-thought-out model that enables all the nice features of version control, like maintaining history, supporting branches, and enabling collaboration.

Snapshots

Git models the history of a collection of files and folders within some top-level directory as a series of snapshots. In Git terminology, a file is called a "blob", and it's just a bunch of bytes. A directory is called a "tree", and it maps names to blobs or trees (so directories can contain other directories). A snapshot is the top-level tree that is being tracked. For example, we might have a tree as follows:

```
<root> (tree)
|
+- foo (tree)
|  |
|  + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

The top-level tree contains two elements, a tree "foo" (that itself contains one element, a blob "bar.txt"), and a blob "baz.txt".

Modeling history: relating snapshots

How should a version control system relate snapshots? One simple model would be to have a linear history. A history would be a list of snapshots in time-order. For many reasons, Git doesn't use a simple model like this.

In Git, a history is a directed acyclic graph (DAG) of snapshots. That may sound like a fancy math word, but don't be intimidated. All this means is that each snapshot in Git refers to a set of "parents", the snapshots that preceded it. It's a set of parents rather than a single parent (as would be the case in a linear history) because a snapshot might descend from multiple parents, for example, due to combining (merging) two parallel branches of development.

Git calls these snapshots "commit"s. Visualizing a commit history might look something like this:

```
o <-- o <-- o <-- o
            ^
             \
              --- o <-- o
```

In the ASCII art above, the `o`s correspond to individual commits (snapshots). The arrows point to the parent of each commit (it's a "comes before" relation, not "comes after"). After the third commit, the history branches into two separate branches. This might correspond to, for example, two separate features being developed in parallel, independently from each other. In the future, these branches may be merged to create a

new snapshot that incorporates both of the features, producing a new history that looks like this, with the newly created merge commit shown in bold:

Commits in Git are immutable. This doesn't mean that mistakes can't be corrected, however; it's just that "edits" to the commit history are actually creating entirely new commits, and references (see below) are updated to point to the new ones.

Data model, as pseudocode

It may be instructive to see Git's data model written down in pseudocode:

```
// a file is a bunch of bytes
type blob = array<byte>

// a directory contains named files and directories
type tree = map<string, tree | blob>

// a commit has parents, metadata, and the top-level tree
type commit = struct {
    parents: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

It's a clean, simple model of history.

Objects and content-addressing

An "object" is a blob, tree, or commit:

```
type object = blob | tree | commit
```

In Git data store, all objects are content-addressed by their SHA-1 hash.

```
objects = map<string, object>

def store(object):
    id = sha1(object)
    objects[id] = object

def load(id):
    return objects[id]
```

Blobs, trees, and commits are unified in this way: they are all objects. When they reference other objects, they don't actually *contain* them in their on-disk representation, but have a reference to them by their hash.

For example, the tree for the example directory structure above (visualized using `git cat-file -p 698281bc680d1995c5f4caaf3359721a5a58d48d`), looks like this:

```
100644 blob 4448adbf7ecd394f42ae135bbeed9676e894af85    baz.txt
040000 tree c68d233a33c5c06e0340e4c224f0afca87c8ce87    foo
```

The tree itself contains pointers to its contents, `baz.txt` (a blob) and `foo` (a tree). If we look at the contents addressed by the hash corresponding to baz.txt with `git cat-file -p 4448adbf7ecd394f42ae135bbeed9676e894af85`, we get the following:

```
git is wonderful
```

References

Now, all snapshots can be identified by their SHA-1 hashes. That's inconvenient, because humans aren't good at remembering strings of 40 hexadecimal characters.

Git's solution to this problem is human-readable names for SHA-1 hashes, called "references". References are pointers to commits. Unlike objects, which are immutable, references are mutable (can be updated to point to a new commit). For example, the `master` reference usually points to the latest commit in the main branch of development.

```
references = map<string, string>

def update_reference(name, id):
    references[name] = id

def read_reference(name):
    return references[name]

def load_reference(name_or_id):
    if name_or_id in references:
        return load(references[name_or_id])
    else:
        return load(name_or_id)
```

With this, Git can use human-readable names like "master" to refer to a particular snapshot in the history, instead of a long hexadecimal string.

One detail is that we often want a notion of "where we currently are" in the history, so that when we take a new snapshot, we know what it is relative to (how we set the `parents` field of the commit). In Git, that "where we currently are" is a special reference called "HEAD".

Repositories

Finally, we can define what (roughly) is a Git *repository*: it is the data `objects` and `references`.

On disk, all Git stores are objects and references: that's all there is to Git's data model. All `git` commands map to some manipulation of the commit DAG by adding objects and adding/updating references.

Whenever you're typing in any command, think about what manipulation the command is making to the underlying graph data structure. Conversely, if you're trying to make a particular kind of change to the commit DAG, e.g. "discard uncommitted changes and make the 'master' ref point to commit `5d83f9e`", there's probably a command to do it (e.g. in this case, `git checkout master; git reset --hard 5d83f9e`).

Staging area

This is another concept that's orthogonal to the data model, but it's a part of the interface to create commits.

One way you might imagine implementing snapshotting as described above is to have a "create snapshot" command that creates a new snapshot based on the *current state* of the working directory. Some version control tools work like this, but not Git. We want clean snapshots, and it might not always be ideal to make a snapshot from the current

state. For example, imagine a scenario where you've implemented two separate features, and you want to create two separate commits, where the first introduces the first feature, and the next introduces the second feature. Or imagine a scenario where you have debugging print statements added all over your code, along with a bugfix; you want to commit the bugfix while discarding all the print statements.

Git accommodates such scenarios by allowing you to specify which modifications should be included in the next snapshot through a mechanism called the "staging area".

Git command-line interface

To avoid duplicating information, we're not going to explain the commands below in detail. See the highly recommended Pro Git for more information, or watch the lecture video.

Basics

- `git help <command>`: get help for a git command
- `git init`: creates a new git repo, with data stored in the `.git` directory
- `git status`: tells you what's going on
- `git add <filename>`: adds files to staging area
- `git commit`: creates a new commit
  - Write good commit messages!
  - Even more reasons to write good commit messages!
- `git log`: shows a flattened log of history
- `git log --all --graph --decorate`: visualizes history as a DAG
- `git diff <filename>`: show changes you made relative to the staging area
- `git diff <revision> <filename>`: shows differences in a file between snapshots
- `git checkout <revision>`: updates HEAD and current branch

Branching and merging

- `git branch`: shows branches
- `git branch <name>`: creates a branch
- `git checkout -b <name>`: creates a branch and switches to it
  - same as `git branch <name>; git checkout <name>`
- `git merge <revision>`: merges into current branch
- `git mergetool`: use a fancy tool to help resolve merge conflicts
- `git rebase`: rebase set of patches onto a new base

Remotes

- `git remote`: list remotes
- `git remote add <name> <url>`: add a remote

- `git push <remote> <local branch>:<remote branch>`: send objects to remote, and update remote reference
- `git branch --set-upstream-to=<remote>/<remote branch>`: set up correspondence between local and remote branch
- `git fetch`: retrieve objects/references from a remote
- `git pull`: same as `git fetch; git merge`
- `git clone`: download repository from remote

Undo

- `git commit --amend`: edit a commit's contents/message
- `git reset HEAD <file>`: unstage a file
- `git checkout -- <file>`: discard changes

Advanced Git

- `git config`: Git is highly customizable
- `git clone --depth=1`: shallow clone, without entire version history
- `git add -p`: interactive staging
- `git rebase -i`: interactive rebasing
- `git blame`: show who last edited which line
- `git stash`: temporarily remove modifications to working directory
- `git bisect`: binary search history (e.g. for regressions)
- `.gitignore`: specify intentionally untracked files to ignore

Miscellaneous

- **GUIs**: there are many GUI clients out there for Git. We personally don't use them and use the command-line interface instead.
- **Shell integration**: it's super handy to have a Git status as part of your shell prompt (zsh, bash). Often included in frameworks like Oh My Zsh.
- **Editor integration**: similarly to the above, handy integrations with many features. fugitive.vim is the standard one for Vim.
- **Workflows**: we taught you the data model, plus some basic commands; we didn't tell you what practices to follow when working on big projects (and there are many different approaches).
- **GitHub**: Git is not GitHub. GitHub has a specific way of contributing code to other projects, called pull requests.
- **Other Git providers**: GitHub is not special: there are many Git repository hosts, like GitLab and BitBucket.

Resources

- Pro Git is **highly recommended reading**. Going through Chapters 1–5 should teach you most of what you need to use Git proficiently, now that you understand the data model. The later chapters have some interesting, advanced material.
- Oh Shit, Git!?! is a short guide on how to recover from some common Git mistakes.
- Git for Computer Scientists is a short explanation of Git's data model, with less pseudocode and more fancy diagrams than these lecture notes.
- Git from the Bottom Up is a detailed explanation of Git's implementation details beyond just the data model, for the curious.
- How to explain git in simple words
- Learn Git Branching is a browser-based game that teaches you Git.

Exercises

1. If you don't have any past experience with Git, either try reading the first couple chapters of Pro Git or go through a tutorial like Learn Git Branching. As you're working through it, relate Git commands to the data model.
2. Clone the repository for the class website.
   1. Explore the version history by visualizing it as a graph.
   2. Who was the last person to modify `README.md`? (Hint: use `git log` with an argument).
   3. What was the commit message associated with the last modification to the `collections:` line of `_config.yml`? (Hint: use `git blame` and `git show`).
3. One common mistake when learning Git is to commit large files that should not be managed by Git or adding sensitive information. Try adding a file to a repository, making some commits and then deleting that file from history (you may want to look at this).
4. Clone some repository from GitHub, and modify one of its existing files. What happens when you do `git stash`? What do you see when running `git log --all --oneline`? Run `git stash pop` to undo what you did with `git stash`. In what scenario might this be useful?
5. Like many command line tools, Git provides a configuration file (or dotfile) called `~/.gitconfig`. Create an alias in `~/.gitconfig` so that when you run `git graph`, you get the output of `git log --all --graph --decorate --oneline`. Information about git aliases can be found here.
6. You can define global ignore patterns in `~/.gitignore_global` after running `git config --global core.excludesfile ~/.gitignore_global`. Do this, and set up your global gitignore file to ignore OS-specific or editor-specific temporary files, like `.DS_Store`.
7. Fork the repository for the class website, find a typo or some other improvement you can make, and submit a pull request on GitHub (you may want to look at this).

Debugging and Profiling

A golden rule in programming is that code does not do what you expect it to do, but what you tell it to do. Bridging that gap can sometimes be a quite difficult feat. In this lecture we are going to cover useful techniques for dealing with buggy and resource hungry code: debugging and profiling.

Debugging

Printf debugging and Logging

"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements" — Brian Kernighan, *Unix for Beginners*.

A first approach to debug a program is to add print statements around where you have detected the problem, and keep iterating until you have extracted enough information to understand what is responsible for the issue.

A second approach is to use logging in your program, instead of ad hoc print statements. Logging is better than regular print statements for several reasons:

- You can log to files, sockets or even remote servers instead of standard output.
- Logging supports severity levels (such as INFO, DEBUG, WARN, ERROR, &c), that allow you to filter the output accordingly.
- For new issues, there's a fair chance that your logs will contain enough information to detect what is going wrong.

Here is an example code that logs messages:

```
$ python logger.py
# Raw output as with just prints
$ python logger.py log
# Log formatted output
$ python logger.py log ERROR
# Print only ERROR levels and above
$ python logger.py color
# Color formatted output
```

One of my favorite tips for making logs more readable is to color code them. By now you probably have realized that your terminal uses colors to make things more readable. But how does it do it? Programs like `ls` or `grep` are using ANSI escape codes, which are special sequences of characters to indicate your shell to change the color of the output. For example, executing `echo -e "\e[38;2;255;0;0mThis is red\e[0m"` prints the message `This is red` in red on your terminal, as long as it supports true color. If your

terminal doesn't support this (e.g. macOS's Terminal.app), you can use the more universally supported escape codes for 16 color choices, for example `echo -e "\e[31;1mThis is red\e[0m"`.

The following script shows how to print many RGB colors into your terminal (again, as long as it supports true color).

```
#!/usr/bin/env bash
for R in $(seq 0 20 255); do
    for G in $(seq 0 20 255); do
        for B in $(seq 0 20 255); do
            printf "\e[38;2;${R};${G};${B}m\e[0m";
        done
    done
done
```

Third party logs

As you start building larger software systems you will most probably run into dependencies that run as separate programs. Web servers, databases or message brokers are common examples of this kind of dependencies. When interacting with these systems it is often necessary to read their logs, since client side error messages might not suffice.

Luckily, most programs write their own logs somewhere in your system. In UNIX systems, it is commonplace for programs to write their logs under `/var/log`. For instance, the NGINX webserver places its logs under `/var/log/nginx`. More recently, systems have started using a **system log**, which is increasingly where all of your log messages go. Most (but not all) Linux systems use `systemd`, a system daemon that controls many things in your system such as which services are enabled and running. `systemd` places the logs under `/var/log/journal` in a specialized format and you can use the `journalctl` command to display the messages. Similarly, on macOS there is still `/var/log/system.log` but an increasing number of tools use the system log, that can be displayed with `log show`. On most UNIX systems you can also use the `dmesg` command to access the kernel log.

For logging under the system logs you can use the `logger` shell program. Here's an example of using `logger` and how to check that the entry made it to the system logs. Moreover, most programming languages have bindings logging to the system log.

```
logger "Hello Logs"
# On macOS
log show --last 1m | grep Hello
# On Linux
journalctl --since "1m ago" | grep Hello
```

As we saw in the data wrangling lecture, logs can be quite verbose and they require some level of processing and filtering to get the information you want. If you find yourself heavily filtering through `journalctl` and `log show` you can consider using their flags, which can perform a first pass of filtering of their output. There are also some tools like `lnav`, that provide an improved presentation and navigation for log files.

Debuggers

When printf debugging is not enough you should use a debugger. Debuggers are programs that let you interact with the execution of a program, allowing the following:

- Halt execution of the program when it reaches a certain line.
- Step through the program one instruction at a time.
- Inspect values of variables after the program crashed.
- Conditionally halt the execution when a given condition is met.
- And many more advanced features

Many programming languages come with some form of debugger. In Python this is the Python Debugger `pdb`.

Here is a brief description of some of the commands `pdb` supports:

- **l**(ist) - Displays 11 lines around the current line or continue the previous listing.
- **s**(tep) - Execute the current line, stop at the first possible occasion.
- **n**(ext) - Continue execution until the next line in the current function is reached or it returns.
- **b**(reak) - Set a breakpoint (depending on the argument provided).
- **p**(rint) - Evaluate the expression in the current context and print its value. There's also **pp** to display using `pprint` instead.
- **r**(eturn) - Continue execution until the current function returns.
- **q**(uit) - Quit the debugger.

Let's go through an example of using `pdb` to fix the following buggy python code. (See the lecture video).

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n):
            if arr[j] > arr[j+1]:
                arr[j] = arr[j+1]
                arr[j+1] = arr[j]
    return arr

print(bubble_sort([4, 2, 1, 8, 7, 6]))
```

Note that since Python is an interpreted language we can use the `pdb` shell to execute commands and to execute instructions. `ipdb` is an improved `pdb` that uses the `IPython` REPL enabling tab completion, syntax highlighting, better tracebacks, and better introspection while retaining the same interface as the `pdb` module.

For more low level programming you will probably want to look into `gdb` (and its quality of life modification `pwndbg`) and `lldb`. They are optimized for C-like language debugging but will let you probe pretty much any process and get its current machine state: registers, stack, program counter, &c.

Specialized Tools

Even if what you are trying to debug is a black box binary there are tools that can help you with that. Whenever programs need to perform actions that only the kernel can, they use System Calls. There are commands that let you trace the syscalls your program makes. In Linux there's `strace` and macOS and BSD have `dtrace`. `dtrace` can be tricky to use because it uses its own `D` language, but there is a wrapper called `dtruss` that provides an interface more similar to `strace` (more details here).

Below are some examples of using `strace` or `dtruss` to show `stat` syscall traces for an execution of `ls`. For a deeper dive into `strace`, this article and this zine are good reads.

```
# On Linux
sudo strace -e lstat ls -l > /dev/null
4
# On macOS
sudo dtruss -t lstat64_extended ls -l > /dev/null
```

Under some circumstances, you may need to look at the network packets to figure out the issue in your program. Tools like `tcpdump` and Wireshark are network packet analyzers that let you read the contents of network packets and filter them based on different criteria.

For web development, the Chrome/Firefox developer tools are quite handy. They feature a large number of tools, including:

- Source code - Inspect the HTML/CSS/JS source code of any website.
- Live HTML, CSS, JS modification - Change the website content, styles and behavior to test (you can see for yourself that website screenshots are not valid proofs).
- Javascript shell - Execute commands in the JS REPL.
- Network - Analyze the requests timeline.
- Storage - Look into the Cookies and local application storage.

Static Analysis

For some issues you do not need to run any code. For example, just by carefully looking at a piece of code you could realize that your loop variable is shadowing an already existing variable or function name; or that a program reads a variable before defining it. Here is where static analysis tools come into play. Static analysis programs take source code as input and analyze it using coding rules to reason about its correctness.

In the following Python snippet there are several mistakes. First, our loop variable `foo` shadows the previous definition of the function `foo`. We also wrote `baz` instead of `bar` in the last line, so the program will crash after completing the `sleep` call (which will take one minute).

```
import time

def foo():
    return 42

for foo in range(5):
    print(foo)
bar = 1
bar *= 0.2
time.sleep(60)
print(baz)
```

Static analysis tools can identify this kind of issues. When we run `pyflakes` on the code we get the errors related to both bugs. `mypy` is another tool that can detect type checking issues. Here, `mypy` will warn us that `bar` is initially an `int` and is then casted to a `float`. Again, note that all these issues were detected without having to run the code.

```
$ pyflakes foobar.py
foobar.py:6: redefinition of unused 'foo' from line 3
foobar.py:11: undefined name 'baz'

$ mypy foobar.py
foobar.py:6: error: Incompatible types in assignment (expression has type
foobar.py:9: error: Incompatible types in assignment (expression has type
foobar.py:11: error: Name 'baz' is not defined
Found 3 errors in 1 file (checked 1 source file)
```

In the shell tools lecture we covered `shellcheck`, which is a similar tool for shell scripts.

Most editors and IDEs support displaying the output of these tools within the editor itself, highlighting the locations of warnings and errors. This is often called **code linting** and it can also be used to display other types of issues such as stylistic violations or insecure constructs.

In vim, the plugins `ale` or `syntastic` will let you do that. For Python, `pylint` and `pep8` are examples of stylistic linters and `bandit` is a tool designed to find common security issues. For other languages people have compiled comprehensive lists of useful static analysis tools, such as Awesome Static Analysis (you may want to take a look at the *Writing* section) and for linters there is Awesome Linters.

A complementary tool to stylistic linting are code formatters such as `black` for Python, `gofmt` for Go, `rustfmt` for Rust or `prettier` for JavaScript, HTML and CSS. These tools autoformat your code so that it's consistent with common stylistic patterns for the given programming language. Although you might be unwilling to give stylistic control about your code, standardizing code format will help other people read your code and will make you better at reading other people's (stylistically standardized) code.

Profiling

Even if your code functionally behaves as you would expect, that might not be good enough if it takes all your CPU or memory in the process. Algorithms classes often teach big *O* notation but not how to find hot spots in your programs. Since premature optimization is the root of all evil, you should learn about profilers and monitoring tools. They will help you understand which parts of your program are taking most of the time and/or resources so you can focus on optimizing those parts.

Timing

Similarly to the debugging case, in many scenarios it can be enough to just print the time it took your code between two points. Here is an example in Python using the `time` module.

```
import time, random
n = random.randint(1, 10) * 100

# Get current time
start = time.time()

# Do some work
print("Sleeping for {} ms".format(n))
time.sleep(n/1000)

# Compute time between start and now
print(time.time() - start)

# Output
# Sleeping for 500 ms
# 0.5713930130004883
```

However, wall clock time can be misleading since your computer might be running other processes at the same time or waiting for events to happen. It is common for tools to make a distinction between *Real*, *User* and *Sys* time. In general, *User* + *Sys* tells you how much time your process actually spent in the CPU (more detailed explanation here).

- *Real* - Wall clock elapsed time from start to finish of the program, including the time taken by other processes and time taken while blocked (e.g. waiting for I/O or network)
- *User* - Amount of time spent in the CPU running user code
- *Sys* - Amount of time spent in the CPU running kernel code

For example, try running a command that performs an HTTP request and prefixing it with `time`. Under a slow connection you might get an output like the one below. Here it took over 2 seconds for the request to complete but the process only took 15ms of CPU user time and 12ms of kernel CPU time.

```
$ time curl https://missing.csail.mit.edu &> /dev/null
real    0m2.561s
user    0m0.015s
sys     0m0.012s
```

Profilers

CPU

Most of the time when people refer to *profilers* they actually mean *CPU profilers*, which are the most common. There are two main types of CPU profilers: *tracing* and *sampling* profilers. Tracing profilers keep a record of every function call your program makes whereas sampling profilers probe your program periodically (commonly every millisecond) and record the program' s stack. They use these records to present aggregate statistics of what your program spent the most time doing. Here is a good intro article if you want more detail on this topic.

Most programming languages have some sort of command line profiler that you can use to analyze your code. They often integrate with full fledged IDEs but for this lecture we are going to focus on the command line tools themselves.

In Python we can use the `cProfile` module to profile time per function call. Here is a simple example that implements a rudimentary grep in Python:

```python
#!/usr/bin/env python

import sys, re

def grep(pattern, file):
    with open(file, 'r') as f:
        print(file)
        for i, line in enumerate(f.readlines()):
            pattern = re.compile(pattern)
            match = pattern.search(line)
            if match is not None:
                print("{}: {}".format(i, line), end="")

if __name__ == '__main__':
    times = int(sys.argv[1])
    pattern = sys.argv[2]
    for i in range(times):
        for file in sys.argv[3:]:
            grep(pattern, file)
```

We can profile this code using the following command. Analyzing the output we can see that IO is taking most of the time and that compiling the regex takes a fair amount of time as well. Since the regex only needs to be compiled once, we can factor it out of the for.

```
$ python -m cProfile -s tottime grep.py 1000 '^(import|\s*def)[^,]*$' *.p

[omitted program output]

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     8000    0.266    0.000    0.292    0.000 {built-in method io.open}
     8000    0.153    0.000    0.894    0.000 grep.py:5(grep)
    17000    0.101    0.000    0.101    0.000 {built-in method builtins.p
     8000    0.100    0.000    0.129    0.000 {method 'readlines' of '_io
    93000    0.097    0.000    0.111    0.000 re.py:286(_compile)
    93000    0.069    0.000    0.069    0.000 {method 'search' of '_sre.S
    93000    0.030    0.000    0.141    0.000 re.py:231(compile)
    17000    0.019    0.000    0.029    0.000 codecs.py:318(decode)
        1    0.017    0.017    0.911    0.911 grep.py:3(<module>)

[omitted lines]
```

A caveat of Python's `cProfile` profiler (and many profilers for that matter) is that they display time per function call. That can become unintuitive really fast, especially if you are using third party libraries in your code since internal function calls are also accounted for.

A more intuitive way of displaying profiling information is to include the time taken per line of code, which is what *line profilers* do.

For instance, the following piece of Python code performs a request to the class website and parses the response to get all URLs in the page:

```
#!/usr/bin/env python
import requests
from bs4 import BeautifulSoup

# This is a decorator that tells line_profiler
# that we want to analyze this function
@profile
def get_urls():
    response = requests.get('https://missing.csail.mit.edu')
    s = BeautifulSoup(response.content, 'lxml')
    urls = []
    for url in s.find_all('a'):
        urls.append(url['href'])

if __name__ == '__main__':
    get_urls()
```

If we used Python's `cProfile` profiler we'd get over 2500 lines of output, and even with sorting it'd be hard to understand where the time is being spent. A quick run with `line_profiler` shows the time taken per line:

```
$ kernprof -l -v a.py
Wrote profile results to urls.py.lprof
Timer unit: 1e-06 s

Total time: 0.636188 s
File: a.py
Function: get_urls at line 5

Line #  Hits         Time  Per Hit   % Time  Line Contents
==============================================================
 5                                           @profile
 6                                           def get_urls():
 7         1     613909.0 613909.0     96.5      response = requests.get(
 8         1      21559.0  21559.0      3.4      s = BeautifulSoup(respon
 9         1          2.0      2.0      0.0      urls = []
10        25        685.0     27.4      0.1      for url in s.find_all('a
11        24         33.0      1.4      0.0          urls.append(url['hre
```

Memory

In languages like C or C++ memory leaks can cause your program to never release memory that it doesn't need anymore. To help in the process of memory debugging you can use tools like Valgrind that will help you identify memory leaks.

In garbage collected languages like Python it is still useful to use a memory profiler because as long as you have pointers to objects in memory they won't be garbage collected. Here's an example program and its associated output when running it with memory-profiler (note the decorator like in `line-profiler`).

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

if __name__ == '__main__':
    my_func()
```

```
$ python -m memory_profiler example.py
Line #    Mem usage    Increment   Line Contents
================================================
     3                             @profile
     4      5.97 MB    0.00 MB   def my_func():
     5     13.61 MB    7.64 MB       a = [1] * (10 ** 6)
     6    166.20 MB  152.59 MB       b = [2] * (2 * 10 ** 7)
     7     13.61 MB -152.59 MB       del b
     8     13.61 MB    0.00 MB       return a
```

Event Profiling

As it was the case for `strace` for debugging, you might want to ignore the specifics of the code that you are running and treat it like a black box when profiling. The `perf` command abstracts CPU differences away and does not report time or memory, but instead it reports system events related to your programs. For example, `perf` can easily report poor cache locality, high amounts of page faults or livelocks. Here is an overview of the command:

- `perf list` - List the events that can be traced with perf
- `perf stat COMMAND ARG1 ARG2` - Gets counts of different events related a process or command
- `perf record COMMAND ARG1 ARG2` - Records the run of a command and saves the statistical data into a file called `perf.data`
- `perf report` - Formats and prints the data collected in `perf.data`

Visualization

Profiler output for real world programs will contain large amounts of information because of the inherent complexity of software projects. Humans are visual creatures and are quite terrible at reading large amounts of numbers and making sense of them. Thus there are many tools for displaying profiler's output in an easier to parse way.

One common way to display CPU profiling information for sampling profilers is to use a Flame Graph, which will display a hierarchy of function calls across the Y axis and time taken proportional to the X axis. They are also interactive, letting you zoom into specific parts of the program and get their stack traces (try clicking in the image below).

Call graphs or control flow graphs display the relationships between subroutines within a program by including functions as nodes and functions calls between them as directed edges. When coupled with profiling information such as the number of calls and time taken, call graphs can be quite useful for interpreting the flow of a program. In Python you can use the `pycallgraph` library to generate them.

Resource Monitoring

Sometimes, the first step towards analyzing the performance of your program is to understand what its actual resource consumption is. Programs often run slowly when they are resource constrained, e.g. without enough memory or on a slow network connection. There are a myriad of command line tools for probing and displaying different system resources like CPU usage, memory usage, network, disk usage and so on.

- **General Monitoring** - Probably the most popular is `htop`, which is an improved version of `top`. `htop` presents various statistics for the currently running processes on the system. `htop` has a myriad of options and keybinds, some useful ones are: `<F6>` to sort processes, `t` to show tree hierarchy and `h` to toggle threads. See also `glances` for similar implementation with a great UI. For getting aggregate measures across all processes, `dstat` is another nifty tool that computes real-time resource metrics for lots of different subsystems like I/O, networking, CPU utilization, context switches, &c.
- **I/O operations** - `iotop` displays live I/O usage information and is handy to check if a process is doing heavy I/O disk operations
- **Disk Usage** - `df` displays metrics per partitions and `du` displays **d**isk **u**sage per file for the current directory. In these tools the `-h` flag tells the program to print with **h**uman readable format. A more interactive version of `du` is `ncdu` which lets you navigate folders and delete files and folders as you navigate.
- **Memory Usage** - `free` displays the total amount of free and used memory in the system. Memory is also displayed in tools like `htop`.

- **Open Files** - `lsof` lists file information about files opened by processes. It can be quite useful for checking which process has opened a specific file.
- **Network Connections and Config** - `ss` lets you monitor incoming and outgoing network packets statistics as well as interface statistics. A common use case of `ss` is figuring out what process is using a given port in a machine. For displaying routing, network devices and interfaces you can use `ip`. Note that `netstat` and `ifconfig` have been deprecated in favor of the former tools respectively.
- **Network Usage** - `nethogs` and `iftop` are good interactive CLI tools for monitoring network usage.

If you want to test these tools you can also artificially impose loads on the machine using the `stress` command.

Specialized tools

Sometimes, black box benchmarking is all you need to determine what software to use. Tools like `hyperfine` let you quickly benchmark command line programs. For instance, in the shell tools and scripting lecture we recommended `fd` over `find`. We can use `hyperfine` to compare them in tasks we run often. E.g. in the example below `fd` was 20x faster than `find` in my machine.

```
$ hyperfine --warmup 3 'fd -e jpg' 'find . -iname "*.jpg"'
Benchmark #1: fd -e jpg
  Time (mean ± σ):      51.4 ms ±   2.9 ms    [User: 121.0 ms, System: 16
  Range (min … max):    44.2 ms …  60.1 ms    56 runs

Benchmark #2: find . -iname "*.jpg"
  Time (mean ± σ):      1.126 s ±  0.101 s    [User: 141.1 ms, System: 95
  Range (min … max):    0.975 s …  1.287 s    10 runs

Summary
  'fd -e jpg' ran
   21.89 ± 2.33 times faster than 'find . -iname "*.jpg"'
```

As it was the case for debugging, browsers also come with a fantastic set of tools for profiling webpage loading, letting you figure out where time is being spent (loading, rendering, scripting, &c). More info for Firefox and Chrome.

Exercises

Debugging

1. Use `journalctl` on Linux or `log show` on macOS to get the super user accesses and commands in the last day. If there aren't any you can execute some harmless commands such as `sudo ls` and check again.

2. Do this hands on `pdb` tutorial to familiarize yourself with the commands. For a more in depth tutorial read this.

3. Install `shellcheck` and try checking the following script. What is wrong with the code? Fix it. Install a linter plugin in your editor so you can get your warnings automatically.

```
#!/bin/sh
## Example: a typical script with several problems
for f in $(ls *.m3u)
do
  grep -qi hq.*mp3 $f \
    && echo -e 'Playlist $f contains a HQ file in mp3 format'
done
```

4. (Advanced) Read about reversible debugging and get a simple example working using `rr` or `RevPDB`.

Profiling

5. Here are some sorting algorithm implementations. Use `cProfile` and `line_profiler` to compare the runtime of insertion sort and quicksort. What is the bottleneck of each algorithm? Use then `memory_profiler` to check the memory consumption, why is insertion sort better? Check now the inplace version of quicksort. Challenge: Use `perf` to look at the cycle counts and cache hits and misses of each algorithm.

6. Here's some (arguably convoluted) Python code for computing Fibonacci numbers using a function for each number.

```
#!/usr/bin/env python
def fib0(): return 0

def fib1(): return 1

s = """def fib{}(): return fib{}() + fib{}()"""

if __name__ == '__main__':

    for n in range(2, 10):
        exec(s.format(n, n-1, n-2))
    # from functools import lru_cache
    # for n in range(10):
    #     exec("fib{} = lru_cache(1)(fib{})".format(n, n))
    print(eval("fib9()"))
```

Put the code into a file and make it executable. Install prerequisites: `pycallgraph` and `graphviz`. (If you can run `dot`, you already have GraphViz.) Run the code as is with `pycallgraph graphviz -- ./fib.py` and check the `pycallgraph.png` file. How many times is `fib0` called?. We can do better than that by memoizing the functions. Uncomment the commented lines and regenerate the images. How many times are we calling each `fibN` function now?

7. A common issue is that a port you want to listen on is already taken by another process. Let's learn how to discover that process pid. First execute `python -m http.server 4444` to start a minimal web server listening on port `4444`. On a separate terminal run `lsof | grep LISTEN` to print all listening processes and ports. Find that process pid and terminate it by running `kill <PID>`.

8. Limiting processes resources can be another handy tool in your toolbox. Try running `stress -c 3` and visualize the CPU consumption with `htop`. Now, execute `taskset --cpu-list 0,2 stress -c 3` and visualize it. Is `stress` taking three CPUs? Why not? Read `man taskset`. Challenge: achieve the same using `cgroups`. Try limiting the memory consumption of `stress -m`.

9. (Advanced) The command `curl ipinfo.io` performs a HTTP request and fetches information about your public IP. Open Wireshark and try to sniff the request and reply packets that `curl` sent and received. (Hint: Use the `http` filter to just watch HTTP packets).

Metaprogramming

What do we mean by "metaprogramming"? Well, it was the best collective term we could come up with for the set of things that are more about *process* than they are about writing code or working more efficiently. In this lecture, we will look at systems for building and testing your code, and for managing dependencies. These may seem like they are of limited importance in your day-to-day as a student, but the moment you interact with a larger code base through an internship or once you enter the "real world", you will see this everywhere. We should note that "metaprogramming" can also mean "programs that operate on programs", whereas that is not quite the definition we are using for the purposes of this lecture.

Build systems

If you write a paper in LaTeX, what are the commands you need to run to produce your paper? What about the ones used to run your benchmarks, plot them, and then insert that plot into your paper? Or to compile the code provided in the class you're taking and then running the tests?

For most projects, whether they contain code or not, there is a "build process". Some sequence of operations you need to do to go from your inputs to your outputs. Often, that process might have many steps, and many branches. Run this to generate this plot, that to generate those results, and something else to produce the final paper. As with so many of the things we have seen in this class, you are not the first to encounter this annoyance, and luckily there exist many tools to help you!

These are usually called "build systems", and there are *many* of them. Which one you use depends on the task at hand, your language of preference, and the size of the project. At their core, they are all very similar though. You define a number of *dependencies*, a number of *targets*, and *rules* for going from one to the other. You tell the build system that you want a particular target, and its job is to find all the transitive dependencies of that target, and then apply the rules to produce intermediate targets all the way until the final target has been produced. Ideally, the build system does this without unnecessarily executing rules for targets whose dependencies haven't changed and where the result is available from a previous build.

`make` is one of the most common build systems out there, and you will usually find it installed on pretty much any UNIX-based computer. It has its warts, but works quite well for simple-to-moderate projects. When you run `make`, it consults a file called `Makefile` in the current directory. All the targets, their dependencies, and the rules are defined in that file. Let's take a look at one:

```
paper.pdf: paper.tex plot-data.png
        pdflatex paper.tex

plot-%.png: %.dat plot.py
        ./plot.py -i $*.dat -o $@
```

Each directive in this file is a rule for how to produce the left-hand side using the right-hand side. Or, phrased differently, the things named on the right-hand side are dependencies, and the left-hand side is the target. The indented block is a sequence of programs to produce the target from those dependencies. In `make`, the first directive also defines the default goal. If you run `make` with no arguments, this is the target it will build. Alternatively, you can run something like `make plot-data.png`, and it will build that target instead.

The `%` in a rule is a "pattern", and will match the same string on the left and on the right. For example, if the target `plot-foo.png` is requested, `make` will look for the dependencies `foo.dat` and `plot.py`. Now let's look at what happens if we run `make` with an empty source directory.

```
$ make
make: *** No rule to make target 'paper.tex', needed by 'paper.pdf'.  Sto
```

`make` is helpfully telling us that in order to build `paper.pdf`, it needs `paper.tex`, and it has no rule telling it how to make that file. Let's try making it!

```
$ touch paper.tex
$ make
make: *** No rule to make target 'plot-data.png', needed by 'paper.pdf'.
```

Hmm, interesting, there *is* a rule to make `plot-data.png`, but it is a pattern rule. Since the source files do not exist (`data.dat`), `make` simply states that it cannot make that file. Let's try creating all the files:

```
$ cat paper.tex
\documentclass{article}
\usepackage{graphicx}
\begin{document}
\includegraphics[scale=0.65]{plot-data.png}
\end{document}
$ cat plot.py
#!/usr/bin/env python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-i', type=argparse.FileType('r'))
parser.add_argument('-o')
args = parser.parse_args()

data = np.loadtxt(args.i)
plt.plot(data[:, 0], data[:, 1])
plt.savefig(args.o)
$ cat data.dat
1 1
2 2
3 3
4 4
5 8
```

Now what happens if we run `make`?

```
$ make
./plot.py -i data.dat -o plot-data.png
pdflatex paper.tex
... lots of output ...
```

And look, it made a PDF for us! What if we run `make` again?

```
$ make
make: 'paper.pdf' is up to date.
```

It didn't do anything! Why not? Well, because it didn't need to. It checked that all of the previously-built targets were still up to date with respect to their listed dependencies. We can test this by modifying `paper.tex` and then re-running `make`:

```
$ vim paper.tex
$ make
pdflatex paper.tex
...
```

Notice that `make` did *not* re-run `plot.py` because that was not necessary; none of `plot-data.png`'s dependencies changed!

Dependency management

At a more macro level, your software projects are likely to have dependencies that are themselves projects. You might depend on installed programs (like `python`), system packages (like `openssl`), or libraries within your programming language (like `matplotlib`). These days, most dependencies will be available through a *repository* that hosts a large number of such dependencies in a single place, and provides a convenient mechanism for installing them. Some examples include the Ubuntu package repositories for Ubuntu system packages, which you access through the `apt` tool, RubyGems for Ruby libraries, PyPi for Python libraries, or the Arch User Repository for Arch Linux user-contributed packages.

Since the exact mechanisms for interacting with these repositories vary a lot from repository to repository and from tool to tool, we won't go too much into the details of any specific one in this lecture. What we *will* cover is some of the common terminology they all use. The first among these is *versioning*. Most projects that other projects depend on issue a *version number* with every release. Usually something like 8.1.3 or 64.1.20192004. They are often, but not always, numerical. Version numbers serve many purposes, and one of the most important of them is to ensure that software keeps working. Imagine, for example, that I release a new version of my library where I have renamed a particular function. If someone tried to build some software that depends on my library after I release that update, the build might fail because it calls a function that no longer exists! Versioning attempts to solve this problem by letting a project say that it depends on a particular version, or range of versions, of some other project. That way, even if the underlying library changes, dependent software continues building by using an older version of my library.

That also isn't ideal though! What if I issue a security update which does *not* change the public interface of my library (its "API"), and which any project that depended on the old version should immediately start using? This is where the different groups of numbers in a version come in. The exact meaning of each one varies between projects, but one relatively common standard is *semantic versioning*. With semantic versioning, every version number is of the form: major.minor.patch. The rules are:

- If a new release does not change the API, increase the patch version.
- If you *add* to your API in a backwards-compatible way, increase the minor version.

– If you change the API in a non-backwards-compatible way, increase the major version.

This already provides some major advantages. Now, if my project depends on your project, it *should* be safe to use the latest release with the same major version as the one I built against when I developed it, as long as its minor version is at least what it was back then. In other words, if I depend on your library at version `1.3.7`, then it *should* be fine to build it with `1.3.8`, `1.6.1`, or even `1.3.0`. Version `2.2.4` would probably not be okay, because the major version was increased. We can see an example of semantic versioning in Python's version numbers. Many of you are probably aware that Python 2 and Python 3 code do not mix very well, which is why that was a *major* version bump. Similarly, code written for Python 3.5 might run fine on Python 3.7, but possibly not on 3.4.

When working with dependency management systems, you may also come across the notion of *lock files*. A lock file is simply a file that lists the exact version you are *currently* depending on of each dependency. Usually, you need to explicitly run an update program to upgrade to newer versions of your dependencies. There are many reasons for this, such as avoiding unnecessary recompiles, having reproducible builds, or not automatically updating to the latest version (which may be broken). An extreme version of this kind of dependency locking is *vendoring*, which is where you copy all the code of your dependencies into your own project. That gives you total control over any changes to it, and lets you introduce your own changes to it, but also means you have to explicitly pull in any updates from the upstream maintainers over time.

Continuous integration systems

As you work on larger and larger projects, you'll find that there are often additional tasks you have to do whenever you make a change to it. You might have to upload a new version of the documentation, upload a compiled version somewhere, release the code to pypi, run your test suite, and all sort of other things. Maybe every time someone sends you a pull request on GitHub, you want their code to be style checked and you want some benchmarks to run? When these kinds of needs arise, it's time to take a look at continuous integration.

Continuous integration, or CI, is an umbrella term for "stuff that runs whenever your code changes", and there are many companies out there that provide various types of CI, often for free for open-source projects. Some of the big ones are Travis CI, Azure Pipelines, and GitHub Actions. They all work in roughly the same way: you add a file to your repository that describes what should happen when various things happen to that repository. By far the most common one is a rule like "when someone pushes code, run the test suite". When the event triggers, the CI provider spins up a virtual machines (or more), runs the commands in your "recipe", and then usually notes down the results somewhere. You might set it up so that you are notified if the test suite stops passing, or so that a little badge appears on your repository as long as the tests pass.

As an example of a CI system, the class website is set up using GitHub Pages. Pages is a CI action that runs the Jekyll blog software on every push to `master` and makes the built site available on a particular GitHub domain. This makes it trivial for us to update the website! We just make our changes locally, commit them with git, and then push. CI takes care of the rest.

A brief aside on testing

Most large software projects come with a "test suite". You may already be familiar with the general concept of testing, but we thought we'd quickly mention some approaches to testing and testing terminology that you may encounter in the wild:

- Test suite: a collective term for all the tests
- Unit test: a "micro-test" that tests a specific feature in isolation
- Integration test: a "macro-test" that runs a larger part of the system to check that different feature or components work *together*.
- Regression test: a test that implements a particular pattern that *previously* caused a bug to ensure that the bug does not resurface.
- Mocking: to replace a function, module, or type with a fake implementation to avoid testing unrelated functionality. For example, you might "mock the network" or "mock the disk".

Exercises

1. Most makefiles provide a target called `clean`. This isn't intended to produce a file called `clean`, but instead to clean up any files that can be re-built by make. Think of it as a way to "undo" all of the build steps. Implement a `clean` target for the `paper.pdf` `Makefile` above. You will have to make the target phony. You may find the `git ls-files` subcommand useful. A number of other very common make targets are listed here.
2. Take a look at the various ways to specify version requirements for dependencies in Rust's build system. Most package repositories support similar syntax. For each one (caret, tilde, wildcard, comparison, and multiple), try to come up with a use-case in which that particular kind of requirement makes sense.
3. Git can act as a simple CI system all by itself. In `.git/hooks` inside any git repository, you will find (currently inactive) files that are run as scripts when a particular action happens. Write a `pre-commit` hook that runs `make paper.pdf` and refuses the commit if the `make` command fails. This should prevent any commit from having an unbuildable version of the paper.
4. Set up a simple auto-published page using GitHub Pages. Add a GitHub Action to the repository to run `shellcheck` on any shell files in that repository (here is one way to do it). Check that it works!
5. Build your own GitHub action to run `proselint` or `write-good` on all the `.md` files in the repository. Enable it in your repository, and check that it works by filing a pull

request with a typo in it.

Security and Cryptography

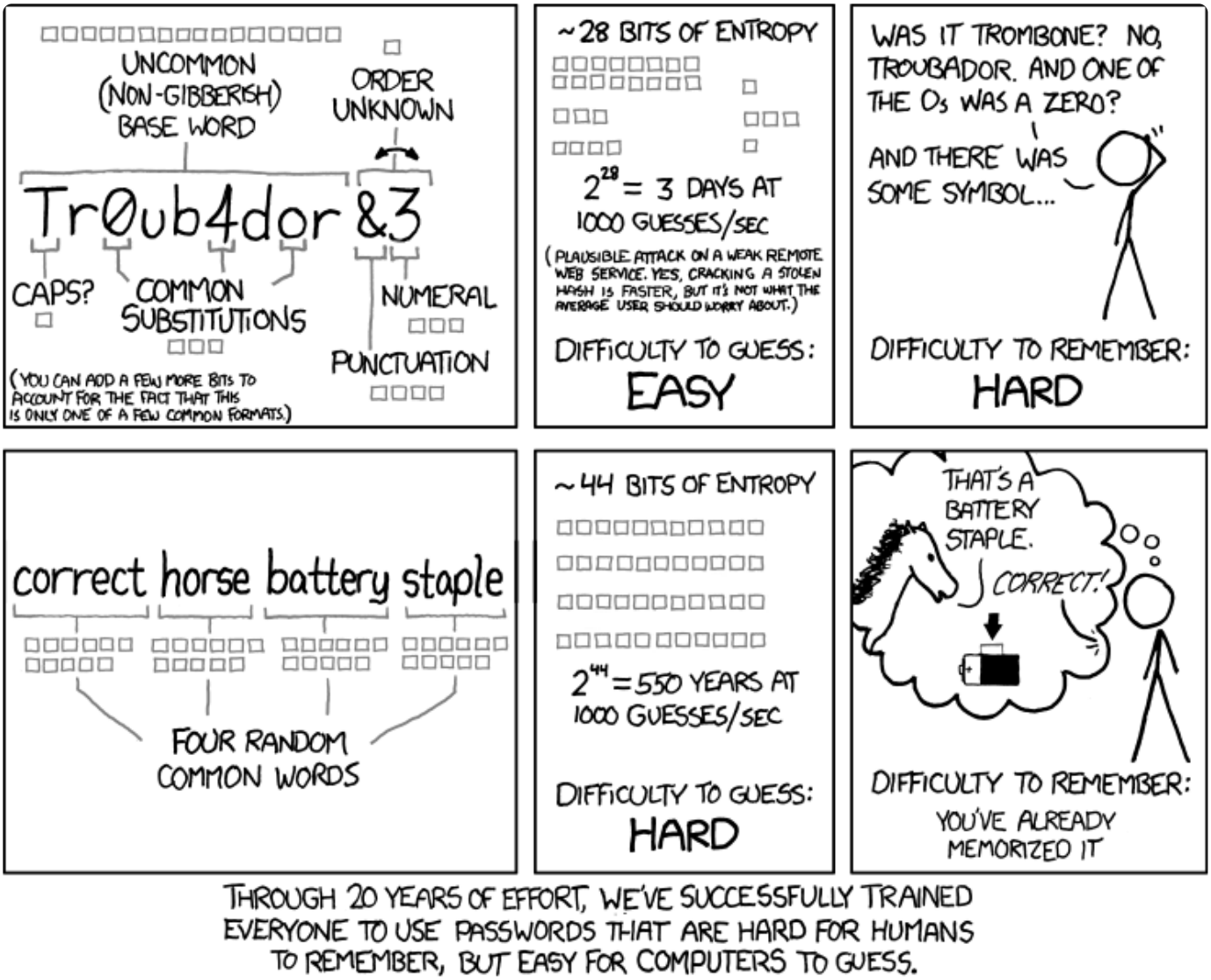
Last year's security and privacy lecture focused on how you can be more secure as a computer *user*. This year, we will focus on security and cryptography concepts that are relevant in understanding tools covered earlier in this class, such as the use of hash functions in Git or key derivation functions and symmetric/asymmetric cryptosystems in SSH.

This lecture is not a substitute for a more rigorous and complete course on computer systems security (6.858) or cryptography (6.857 and 6.875). Don't do security work without formal training in security. Unless you're an expert, don't roll your own crypto. The same principle applies to systems security.

This lecture has a very informal (but we think practical) treatment of basic cryptography concepts. This lecture won't be enough to teach you how to *design* secure systems or cryptographic protocols, but we hope it will be enough to give you a general understanding of the programs and protocols you already use.

Entropy

Entropy is a measure of randomness. This is useful, for example, when determining the strength of a password.



As the above XKCD comic illustrates, a password like "correcthorsebatterystaple" is more secure than one like "Tr0ub4dor&3". But how do you quantify something like this?

Entropy is measured in *bits*, and when selecting uniformly at random from a set of possible outcomes, the entropy is equal to `log_2(# of possibilities)`. A fair coin flip gives 1 bit of entropy. A dice roll (of a 6-sided die) has ~2.58 bits of entropy.

You should consider that the attacker knows the *model* of the password, but not the randomness (e.g. from dice rolls) used to select a particular password.

How many bits of entropy is enough? It depends on your threat model. For online guessing, as the XKCD comic points out, ~40 bits of entropy is pretty good. To be resistant to offline guessing, a stronger password would be necessary (e.g. 80 bits, or more).

Hash functions

A cryptographic hash function maps data of arbitrary size to a fixed size, and has some special properties. A rough specification of a hash function is as follows:

```
hash(value: array<byte>) -> vector<byte, N>  (for some fixed N)
```

An example of a hash function is SHA1, which is used in Git. It maps arbitrary-sized inputs to 160-bit outputs (which can be represented as 40 hexadecimal characters). We can try out the SHA1 hash on an input using the `sha1sum` command:

```
$ printf 'hello' | sha1sum
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
$ printf 'hello' | sha1sum
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
$ printf 'Hello' | sha1sum
f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0
```

At a high level, a hash function can be thought of as a hard-to-invert random-looking (but deterministic) function (and this is the ideal model of a hash function). A hash function has the following properties:

- Deterministic: the same input always generates the same output.
- Non-invertible: it is hard to find an input `m` such that `hash(m) = h` for some desired output `h`.
- Target collision resistant: given an input `m_1`, it's hard to find a different input `m_2` such that `hash(m_1) = hash(m_2)`.
- Collision resistant: it's hard to find two inputs `m_1` and `m_2` such that `hash(m_1) = hash(m_2)` (note that this is a strictly stronger property than target collision resistance).

Note: while it may work for certain purposes, SHA-1 is no longer considered a strong cryptographic hash function. You might find this table of lifetimes of cryptographic hash functions interesting. However, note that recommending specific hash functions is beyond the scope of this lecture. If you are doing work where this matters, you need formal training in security/cryptography.

Applications

- Git, for content-addressed storage. The idea of a hash function is a more general concept (there are non-cryptographic hash functions). Why does Git use a cryptographic hash function?
- A short summary of the contents of a file. Software can often be downloaded from (potentially less trustworthy) mirrors, e.g. Linux ISOs, and it would be nice to not have to trust them. The official sites usually post hashes alongside the download links (that point to third-party mirrors), so that the hash can be checked after downloading a file.
- Commitment schemes. Suppose you want to commit to a particular value, but reveal the value itself later. For example, I want to do a fair coin toss "in my head", without a trusted shared coin that two parties can see. I could choose a value `r = random()`,

and then share `h = sha256(r)`. Then, you could call heads or tails (we'll agree that even `r` means heads, and odd `r` means tails). After you call, I can reveal my value `r`, and you can confirm that I haven't cheated by checking `sha256(r)` matches the hash I shared earlier.

Key derivation functions

A related concept to cryptographic hashes, key derivation functions (KDFs) are used for a number of applications, including producing fixed-length output for use as keys in other cryptographic algorithms. Usually, KDFs are deliberately slow, in order to slow down offline brute-force attacks.

Applications

- Producing keys from passphrases for use in other cryptographic algorithms (e.g. symmetric cryptography, see below).
- Storing login credentials. Storing plaintext passwords is bad; the right approach is to generate and store a random salt `salt = random()` for each user, store `KDF(password + salt)`, and verify login attempts by re-computing the KDF given the entered password and the stored salt.

Symmetric cryptography

Hiding message contents is probably the first concept you think about when you think about cryptography. Symmetric cryptography accomplishes this with the following set of functionality:

```
keygen() -> key  (this function is randomized)

encrypt(plaintext: array<byte>, key) -> array<byte>  (the ciphertext)
decrypt(ciphertext: array<byte>, key) -> array<byte>  (the plaintext)
```

The encrypt function has the property that given the output (ciphertext), it's hard to determine the input (plaintext) without the key. The decrypt function has the obvious correctness property, that `decrypt(encrypt(m, k), k) = m`.

An example of a symmetric cryptosystem in wide use today is AES.

Applications

- Encrypting files for storage in an untrusted cloud service. This can be combined with KDFs, so you can encrypt a file with a passphrase. Generate `key = KDF(passphrase)`, and then store `encrypt(file, key)`.

Asymmetric cryptography

The term "asymmetric" refers to there being two keys, with two different roles. A private key, as its name implies, is meant to be kept private, while the public key can be publicly shared and it won't affect security (unlike sharing the key in a symmetric cryptosystem). Asymmetric cryptosystems provide the following set of functionality, to encrypt/decrypt and to sign/verify:

```
keygen() -> (public key, private key)  (this function is randomized)

encrypt(plaintext: array<byte>, public key) -> array<byte>  (the cipherte
decrypt(ciphertext: array<byte>, private key) -> array<byte>  (the plaint

sign(message: array<byte>, private key) -> array<byte>  (the signature)
verify(message: array<byte>, signature: array<byte>, public key) -> bool
```

The encrypt/decrypt functions have properties similar to their analogs from symmetric cryptosystems. A message can be encrypted using the *public* key. Given the output (ciphertext), it's hard to determine the input (plaintext) without the *private* key. The decrypt function has the obvious correctness property, that `decrypt(encrypt(m, public key), private key) = m`.

Symmetric and asymmetric encryption can be compared to physical locks. A symmetric cryptosystem is like a door lock: anyone with the key can lock and unlock it. Asymmetric encryption is like a padlock with a key. You could give the unlocked lock to someone (the public key), they could put a message in a box and then put the lock on, and after that, only you could open the lock because you kept the key (the private key).

The sign/verify functions have the same properties that you would hope physical signatures would have, in that it's hard to forge a signature. No matter the message, without the *private* key, it's hard to produce a signature such that `verify(message, signature, public key)` returns true. And of course, the verify function has the obvious correctness property that `verify(message, sign(message, private key), public key) = true`.

Applications

- PGP email encryption. People can have their public keys posted online (e.g. in a PGP keyserver, or on Keybase). Anyone can send them encrypted email.
- Private messaging. Apps like Signal and Keybase use asymmetric keys to establish private communication channels.
- Signing software. Git can have GPG-signed commits and tags. With a posted public key, anyone can verify the authenticity of downloaded software.

Key distribution

Asymmetric-key cryptography is wonderful, but it has a big challenge of distributing public keys / mapping public keys to real-world identities. There are many solutions to this problem. Signal has one simple solution: trust on first use, and support out-of-band public key exchange (you verify your friends' "safety numbers" in person). PGP has a different solution, which is web of trust. Keybase has yet another solution of social proof (along with other neat ideas). Each model has its merits; we (the instructors) like Keybase's model.

Case studies

Password managers

This is an essential tool that everyone should try to use (e.g. KeePassXC, pass, and 1Password). Password managers make it convenient to use unique, randomly generated high-entropy passwords for all your logins, and they save all your passwords in one place, encrypted with a symmetric cipher with a key produced from a passphrase using a KDF.

Using a password manager lets you avoid password reuse (so you're less impacted when websites get compromised), use high-entropy passwords (so you're less likely to get compromised), and only need to remember a single high-entropy password.

Two-factor authentication

Two-factor authentication (2FA) requires you to use a passphrase ("something you know") along with a 2FA authenticator (like a YubiKey, "something you have") in order to protect against stolen passwords and phishing attacks.

Full disk encryption

Keeping your laptop's entire disk encrypted is an easy way to protect your data in the case that your laptop is stolen. You can use cryptsetup + LUKS on Linux, BitLocker on Windows, or FileVault on macOS. This encrypts the entire disk with a symmetric cipher, with a key protected by a passphrase.

Private messaging

Use Signal or Keybase. End-to-end security is bootstrapped from asymmetric-key encryption. Obtaining your contacts' public keys is the critical step here. If you want good security, you need to authenticate public keys out-of-band (with Signal or Keybase), or trust social proofs (with Keybase).

SSH

We've covered the use of SSH and SSH keys in an earlier lecture. Let's look at the cryptography aspects of this.

When you run `ssh-keygen`, it generates an asymmetric keypair, `public_key`, `private_key`. This is generated randomly, using entropy provided by the operating system (collected from hardware events, etc.). The public key is stored as-is (it's public, so keeping it a secret is not important), but at rest, the private key should be encrypted on disk. The `ssh-keygen` program prompts the user for a passphrase, and this is fed through a key derivation function to produce a key, which is then used to encrypt the private key with a symmetric cipher.

In use, once the server knows the client's public key (stored in the `.ssh/authorized_keys` file), a connecting client can prove its identity using asymmetric signatures. This is done through challenge-response. At a high level, the server picks a random number and sends it to the client. The client then signs this message and sends the signature back to the server, which checks the signature against the public key on record. This effectively proves that the client is in possession of the private key corresponding to the public key that's in the server's `.ssh/authorized_keys` file, so the server can allow the client to log in.

Resources

- Last year's notes: from when this lecture was more focused on security and privacy as a computer user
- Cryptographic Right Answers: answers "what crypto should I use for X?" for many common X.

Exercises

1. **Entropy.**
   1. Suppose a password is chosen as a concatenation of four lower-case dictionary words, where each word is selected uniformly at random from a dictionary of size 100,000. An example of such a password is `correcthorsebatterystaple`. How many bits of entropy does this have?
   2. Consider an alternative scheme where a password is chosen as a sequence of 8 random alphanumeric characters (including both lower-case and upper-case letters). An example is `rg8Ql34g`. How many bits of entropy does this have?
   3. Which is the stronger password?
   4. Suppose an attacker can try guessing 10,000 passwords per second. On average, how long will it take to break each of the passwords?
2. **Cryptographic hash functions.** Download a Debian image from a mirror (e.g. from this Argentinean mirror). Cross-check the hash (e.g. using the `sha256sum` command) with the hash retrieved from the official Debian site (e.g. this file hosted at `debian.org`, if you've downloaded the linked file from the Argentinean mirror).
3. **Symmetric cryptography.** Encrypt a file with AES encryption, using OpenSSL: `openssl aes-256-cbc -salt -in {input filename} -out {output filename}`. Look at the contents using `cat` or `hexdump`. Decrypt it with `openssl`

`aes-256-cbc -d -in {input filename} -out {output filename}` and confirm that the contents match the original using `cmp`.

4. **Asymmetric cryptography.**
   1. Set up SSH keys on a computer you have access to (not Athena, because Kerberos interacts weirdly with SSH keys). Make sure your private key is encrypted with a passphrase, so it is protected at rest.
   2. Set up GPG
   3. Send Anish an encrypted email (public key).
   4. Sign a Git commit with `git commit -S` or create a signed Git tag with `git tag -s`. Verify the signature on the commit with `git show --show-signature` or on the tag with `git tag -v`.

Security and Privacy

The world is a scary place, and everyone's out to get you.

Okay, maybe not, but that doesn't mean you want to flaunt all your secrets. Security (and privacy) is generally all about raising the bar for attackers. Find out what your threat model is, and then design your security mechanisms around that! If the threat model is the NSA or Mossad, you're *probably* going to have a bad time.

There are *many* ways to make your technical persona more secure. We'll touch on a lot of high-level things here, but this is a process, and educating yourself is one of the best things you can do. So:

Follow the Right People

One of the best ways to improve your security know-how is to follow other people who are vocal about security. Some suggestions:

- @TroyHunt
- @SwiftOnSecurity
- @taviso
- @thegrugq
- @tqbf
- @mattblaze
- @moxie

See also this list for more suggestions.

General Security Advice

Tech Solidarity has a pretty great list of do's and don'ts for journalists that has a lot of sane advice, and is decently up-to-date. @thegrugq also has a good blog post on travel security advice that's worth reading. We'll repeat much of the advice from those sources here, plus some more. Also, get a USB data blocker, because USB is scary.

Authentication

The very first thing you should do, if you haven't already, is download a password manager. Some good ones are:

- 1password
- KeePass
- BitWarden
- `pass`

If you're particularly paranoid, use one that encrypts the passwords locally on your computer, as opposed to storing them in plain-text at the server. Use it to generate passwords for all the web sites you care about right now. Then, switch on two-factor authentication, ideally with a FIDO/U2F dongle (a YubiKey for example, which has 20% off for students). TOTP (like Google Authenticator or Duo) will also work in a pinch, but doesn't protect against phishing. SMS is pretty much useless unless your threat model only includes random strangers picking up your password in transit.

Also, a note about paper keys. Often, services will give you a "backup key" that you can use as a second factor if you lose your real second factor (btw, always keep a backup dongle somewhere safe!). While you *can* stick those in your password managers, that means that should someone get access to your password manager, you're totally hosed (but maybe you're okay with that thread model). If you are truly paranoid, print out these paper keys, never store them digitally, and place them in a safe in the real world.

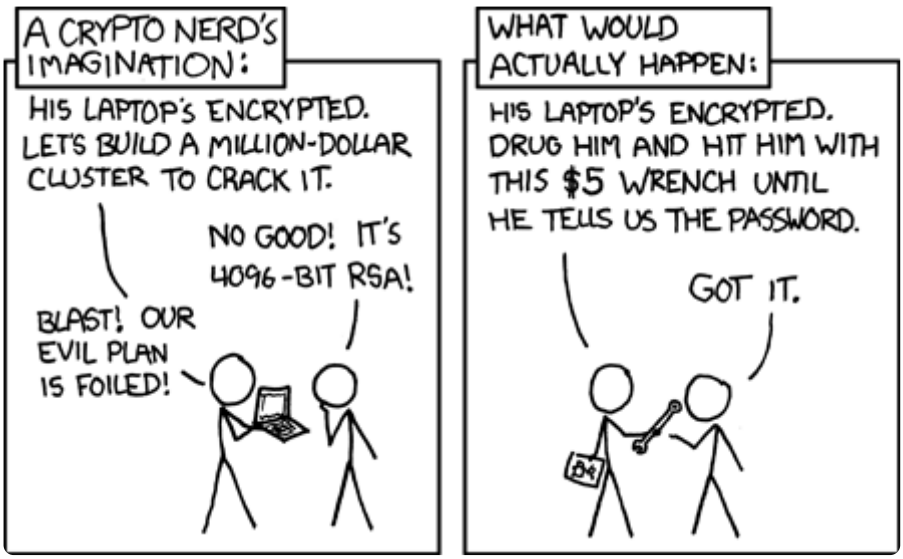
Private Communication

Use Signal (setup instructions. Wire is fine too; WhatsApp is okay; don't use Telegram). Desktop messengers are pretty broken (partially due to usually relying on Electron, which is a huge trust stack).

E-mail is particularly problematic, even if PGP signed. It's not generally forward-secure, and the key-distribution problem is pretty severe. keybase.io helps, and is useful for a number of other reasons. Also, PGP keys are generally handled on desktop computers, which is one of the least secure computing environments. Relatedly, consider getting a Chromebook, or just work on a tablet with a keyboard.

File Security

File security is hard, and operates on many level. What is it you're trying to secure against?



- Offline attacks (someone steals your laptop while it's off): turn on full disk encryption. (cryptsetup + LUKS on Linux, BitLocker on Windows, FileVault on macOS. Note that this won't help if the attacker *also* has you and really wants your secrets.
- Online attacks (someone has your laptop and it's on): use file encryption. There are two primary mechanisms for doing so
  - Encrypted filesystems: stacked filesystem encryption software encrypts files individually rather than having encrypted block devices. You can "mount" these filesystems by providing the decryption key, and then browse the files inside it freely. When you unmount it, those files are all unavailable. Modern solutions include gocryptfs and eCryptFS. More detailed comparisons can be found here and here
  - Encrypted files: encrypt individual files with symmetric encryption (see `gpg -c`) and a secret key. Or, like `pass`, also encrypt the key with your public key so only you can read it back later with your private key. Exact encryption settings matter a lot!
- Plausible deniability (what seems to be the problem officer?): usually lower performance, and easier to lose data. Hard to actually prove that it provides deniable encryption! See the discussion here, and then consider whether you may want to try VeraCrypt (the maintained fork of good ol' TrueCrypt).
- Encrypted backups: use Tarsnap or Borgbase
  - Think about whether an attacker can delete your backups if they get a hold of your laptop!

Internet Security & Privacy

The internet is a *very* scary place. Open WiFi networks are scary. Make sure you delete them afterwards, otherwise your phone will happily announce and re-connect to something with the same name later!

If you're ever on a network you don't trust, a VPN *may* be worthwhile, but keep in mind that you're trusting the VPN provider *a lot*. Do you really trust them more than your ISP? If you truly want a VPN, use a provider you're sure you trust, and you should probably pay for it. Or set up WireGuard for yourself – it's excellent!

There are also secure configuration settings for a lot of internet-enabled applications at cipherlist.eu. If you're particularly privacy-oriented, privacytools.io is also a good resource.

Some of you may wonder about Tor. Keep in mind that Tor is *not* particularly resistant to powerful global attackers, and is weak against traffic analysis attacks. It may be useful for hiding traffic on a small scale, but won't really buy you all that much in terms of privacy. You're better off using more secure services in the first place (Signal, TLS + certificate pinning, etc.).

Web Security

So, you want to go on the Web too? Jeez, you're really pushing your luck here.

Install HTTPS Everywhere. SSL/TLS is critical, and it's *not* just about encryption, but also about being able to verify that you're talking to the right service in the first place! If you run your own web server, test it and test it again. TLS configuration can get hairy. HTTPS Everywhere will do its very best to never navigate you to HTTP sites when there's an alternative. That doesn't save you, but it helps. If you're truly paranoid, blacklist any SSL/TLS CAs that you don't absolutely need.

Install uBlock Origin. It is a wide-spectrum blocker that doesn't just stop ads, but all sorts of third-party communication a page may try to do. And inline scripts and such. If you're willing to spend some time on configuration to make things work, go to medium mode or even hard mode. Those *will* make some sites not work until you've fiddled with the settings enough, but will also significantly improve your online security.

If you're using Firefox, enable Multi-Account Containers. Create separate containers for social networks, banking, shopping, etc. Firefox will keep the cookies and other state for each of the containers totally separate, so sites you visit in one container can't snoop on sensitive data from the others. In Google Chrome, you can use Chrome Profiles to achieve similar results.

Exercises

TODO

1. Encrypt a file using PGP
2. Use veracrypt to create a simple encrypted volume
3. Enable 2FA for your most data sensitive accounts i.e. GMail, Dropbox, Github, &c

Potpourri

Table of Contents

- Keyboard remapping
- Daemons
- FUSE
- Backups
- APIs
- Common command-line flags/patterns
- Window managers
- VPNs
- Markdown
- Hammerspoon (desktop automation on macOS)
- Booting + Live USBs
- Docker, Vagrant, VMs, Cloud, OpenStack
- Notebook programming
- GitHub

Keyboard remapping

As a programmer, your keyboard is your main input method. As with pretty much anything in your computer, it is configurable (and worth configuring).

The most basic change is to remap keys. This usually involves some software that is listening and, whenever a certain key is pressed, it intercepts that event and replaces it with another event corresponding to a different key. Some examples:

- Remap Caps Lock to Ctrl or Escape. We (the instructors) highly encourage this setting since Caps Lock has a very convenient location but is rarely used.
- Remapping PrtSc to Play/Pause music. Most OSes have a play/pause key.
- Swapping Ctrl and the Meta (Windows or Command) key.

You can also map keys to arbitrary commands of your choosing. This is useful for common tasks that you perform. Here, some software listens for a specific key combination and executes some script whenever that event is detected.

- Open a new terminal or browser window.
- Inserting some specific text, e.g. your long email address or your MIT ID number.
- Sleeping the computer or the displays.

There are even more complex modifications you can configure:

- Remapping sequences of keys, e.g. pressing shift five times toggles Caps Lock.

- Remapping on tap vs on hold, e.g. Caps Lock key is remapped to Esc if you quickly tap it, but is remapped to Ctrl if you hold it and use it as a modifier.
- Having remaps being keyboard or software specific.

Some software resources to get started on the topic:

- macOS - karabiner-elements, skhd or BetterTouchTool
- Linux - xmodmap or Autokey
- Windows - Builtin in Control Panel, AutoHotkey or SharpKeys
- QMK - If your keyboard supports custom firmware you can use QMK to configure the hardware device itself so the remaps works for any machine you use the keyboard with.

Daemons

You are probably already familiar with the notion of daemons, even if the word seems new. Most computers have a series of processes that are always running in the background rather than waiting for a user to launch them and interact with them. These processes are called daemons and the programs that run as daemons often end with a `d` to indicate so. For example `sshd`, the SSH daemon, is the program responsible for listening to incoming SSH requests and checking that the remote user has the necessary credentials to log in.

In Linux, `systemd` (the system daemon) is the most common solution for running and setting up daemon processes. You can run `systemctl status` to list the current running daemons. Most of them might sound unfamiliar but are responsible for core parts of the system such as managing the network, solving DNS queries or displaying the graphical interface for the system. Systemd can be interacted with the `systemctl` command in order to `enable`, `disable`, `start`, `stop`, `restart` or check the `status` of services (those are the `systemctl` commands).

More interestingly, `systemd` has a fairly accessible interface for configuring and enabling new daemons (or services). Below is an example of a daemon for running a simple Python app. We won't go in the details but as you can see most of the fields are pretty self explanatory.

```
# /etc/systemd/system/myapp.service
[Unit]
Description=My Custom App
After=network.target

[Service]
User=foo
Group=foo
WorkingDirectory=/home/foo/projects/mydaemon
ExecStart=/usr/bin/local/python3.7 app.py
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

Also, if you just want to run some program with a given frequency there is no need to build a custom daemon, you can use `cron`, a daemon your system already runs to perform scheduled tasks.

FUSE

Modern software systems are usually composed of smaller building blocks that are composed together. Your operating system supports using different filesystem backends because there is a common language of what operations a filesystem supports. For instance, when you run `touch` to create a file, `touch` performs a system call to the kernel to create the file and the kernel performs the appropriate filesystem call to create the given file. A caveat is that UNIX filesystems are traditionally implemented as kernel modules and only the kernel is allowed to perform filesystem calls.

FUSE (Filesystem in User Space) allows filesystems to be implemented by a user program. FUSE lets users run user space code for filesystem calls and then bridges the necessary calls to the kernel interfaces. In practice, this means that users can implement arbitrary functionality for filesystem calls.

For example, FUSE can be used so whenever you perform an operation in a virtual filesystem, that operation is forwarded through SSH to a remote machine, performed there, and the output is returned back to you. This way, local programs can see the file as if it was in your computer while in reality it's in a remote server. This is effectively what `sshfs` does.

Some interesting examples of FUSE filesystems are:

- sshfs - Open locally remote files/folder through an SSH connection.
- rclone - Mount cloud storage services like Dropbox, GDrive, Amazon S3 or Google Cloud Storage and open data locally.

- gocryptfs - Encrypted overlay system. Files are stored encrypted but once the FS is mounted they appear as plaintext in the mountpoint.
- kbfs - Distributed filesystem with end-to-end encryption. You can have private, shared and public folders.
- borgbackup - Mount your deduplicated, compressed and encrypted backups for ease of browsing.

Backups

Any data that you haven't backed up is data that could be gone at any moment, forever. It's easy to copy data around, it's hard to reliably backup data. Here are some good backup basics and the pitfalls of some approaches.

First, a copy of the data in the same disk is not a backup, because the disk is the single point of failure for all the data. Similarly, an external drive in your home is also a weak backup solution since it could be lost in a fire/robbery/&c. Instead, having an off-site backup is a recommended practice.

Synchronization solutions are not backups. For instance, Dropbox/GDrive are convenient solutions, but when data is erased or corrupted they propagate the change. For the same reason, disk mirroring solutions like RAID are not backups. They don't help if data gets deleted, corrupted or encrypted by ransomware.

Some core features of good backups solutions are versioning, deduplication and security. Versioning backups ensure that you can access your history of changes and efficiently recover files. Efficient backup solutions use data deduplication to only store incremental changes and reduce the storage overhead. Regarding security, you should ask yourself what someone would need to know/have in order to read your data and, more importantly, to delete all your data and associated backups. Lastly, blindly trusting backups is a terrible idea and you should verify regularly that you can use them to recover data.

Backups go beyond local files in your computer. Given the significant growth of web applications, large amounts of your data are only stored in the cloud. For instance, your webmail, social media photos, music playlists in streaming services or online docs are gone if you lose access to the corresponding accounts. Having an offline copy of this information is the way to go, and you can find online tools that people have built to fetch the data and save it.

For a more detailed explanation, see 2019's lecture notes on Backups.

APIs

We've talked a lot in this class about using your computer more efficiently to accomplish *local* tasks, but you will find that many of these lessons also extend to the wider internet. Most services online will have "APIs" that let you programmatically

access their data. For example, the US government has an API that lets you get weather forecasts, which you could use to easily get a weather forecast in your shell.

Most of these APIs have a similar format. They are structured URLs, often rooted at `api.service.com`, where the path and query parameters indicate what data you want to read or what action you want to perform. For the US weather data for example, to get the forecast for a particular location, you issue GET request (with `curl` for example) to https://api.weather.gov/points/42.3604,-71.094. The response itself contains a bunch of other URLs that let you get specific forecasts for that region. Usually, the responses are formatted as JSON, which you can then pipe through a tool like `jq` to massage into what you care about.

Some APIs require authentication, and this usually takes the form of some sort of secret *token* that you need to include with the request. You should read the documentation for the API to see what the particular service you are looking for uses, but "OAuth" is a protocol you will often see used. At its heart, OAuth is a way to give you tokens that can "act as you" on a given service, and can only be used for particular purposes. Keep in mind that these tokens are *secret*, and anyone who gains access to your token can do whatever the token allows under *your* account!

IFTTT is a website and service centered around the idea of APIs — it provides integrations with tons of services, and lets you chain events from them in nearly arbitrary ways. Give it a look!

Common command-line flags/patterns

Command-line tools vary a lot, and you will often want to check out their `man` pages before using them. They often share some common features though that can be good to be aware of:

- Most tools support some kind of `--help` flag to display brief usage instructions for the tool.
- Many tools that can cause irrevocable change support the notion of a "dry run" in which they only print what they *would have done*, but do not actually perform the change. Similarly, they often have an "interactive" flag that will prompt you for each destructive action.
- You can usually use `--version` or `-V` to have the program print its own version (handy for reporting bugs!).
- Almost all tools have a `--verbose` or `-v` flag to produce more verbose output. You can usually include the flag multiple times (`-vvv`) to get *more* verbose output, which can be handy for debugging. Similarly, many tools have a `--quiet` flag for making it only print something on error.
- In many tools, `-` in place of a file name means "standard input" or "standard output", depending on the argument.

- Possibly destructive tools are generally not recursive by default, but support a "recursive" flag (often `-r`) to make them recurse.
- Sometimes, you want to pass something that *looks* like a flag as a normal argument. For example, imagine you wanted to remove a file called `-r`. Or you want to run one program "through" another, like `ssh machine foo`, and you want to pass a flag to the "inner" program (`foo`). The special argument `--` makes a program *stop* processing flags and options (things starting with `-`) in what follows, letting you pass things that look like flags without them being interpreted as such: `rm -- -r` or `ssh machine --for-ssh -- foo --for-foo`.

Window managers

Most of you are used to using a "drag and drop" window manager, like what comes with Windows, macOS, and Ubuntu by default. There are windows that just sort of hang there on screen, and you can drag them around, resize them, and have them overlap one another. But these are only one *type* of window manager, often referred to as a "floating" window manager. There are many others, especially on Linux. A particularly common alternative is a "tiling" window manager. In a tiling window manager, windows never overlap, and are instead arranged as tiles on your screen, sort of like panes in tmux. With a tiling window manager, the screen is always filled by whatever windows are open, arranged according to some *layout*. If you have just one window, it takes up the full screen. If you then open another, the original window shrinks to make room for it (often something like 2/3 and 1/3). If you open a third, the other windows will again shrink to accommodate the new window. Just like with tmux panes, you can navigate around these tiled windows with your keyboard, and you can resize them and move them around, all without touching the mouse. They are worth looking into!

VPNs

VPNs are all the rage these days, but it's not clear that's for any good reason. You should be aware of what a VPN does and does not get you. A VPN, in the best case, is *really* just a way for you to change your internet service provider as far as the internet is concerned. All your traffic will look like it's coming from the VPN provider instead of your "real" location, and the network you are connected to will only see encrypted traffic.

While that may seem attractive, keep in mind that when you use a VPN, all you are really doing is shifting your trust from you current ISP to the VPN hosting company. Whatever your ISP *could* see, the VPN provider now sees *instead*. If you trust them *more* than your ISP, that is a win, but otherwise, it is not clear that you have gained much. If you are sitting on some dodgy unencrypted public Wi-Fi at an airport, then maybe you don't trust the connection much, but at home, the trade-off is not quite as clear.

You should also know that these days, much of your traffic, at least of a sensitive nature, is *already* encrypted through HTTPS or TLS more generally. In that case, it usually matters

little whether you are on a "bad" network or not – the network operator will only learn what servers you talk to, but not anything about the data that is exchanged.

Notice that I said "in the best case" above. It is not unheard of for VPN providers to accidentally misconfigure their software such that the encryption is either weak or entirely disabled. Some VPN providers are malicious (or at the very least opportunist), and will log all your traffic, and possibly sell information about it to third parties. Choosing a bad VPN provider is often worse than not using one in the first place.

In a pinch, MIT runs a VPN for its students, so that may be worth taking a look at. Also, if you're going to roll your own, give WireGuard a look.

Markdown

There is a high chance that you will write some text over the course of your career. And often, you will want to mark up that text in simple ways. You want some text to be bold or italic, or you want to add headers, links, and code fragments. Instead of pulling out a heavy tool like Word or LaTeX, you may want to consider using the lightweight markup language Markdown.

You have probably seen Markdown already, or at least some variant of it. Subsets of it are used and supported almost everywhere, even if it's not under the name Markdown. At its core, Markdown is an attempt to codify the way that people already often mark up text when they are writing plain text documents. Emphasis (*italics*) is added by surrounding a word with `*`. Strong emphasis (**bold**) is added using `**`. Lines starting with `#` are headings (and the number of `#`s is the subheading level). Any line starting with `-` is a bullet list item, and any line starting with a number `+ .` is a numbered list item. Backtick is used to show words in `code font`, and a code block can be entered by indenting a line with four spaces or surrounding it with triple-backticks:

````
```
code goes here
```
````

To add a link, place the *text* for the link in square brackets, and the URL immediately following that in parentheses: `[name](url)`. Markdown is easy to get started with, and you can use it nearly everywhere. In fact, the lecture notes for this lecture, and all the others, are written in Markdown, and you can see the raw Markdown here.

Hammerspoon (desktop automation on macOS)

Hammerspoon is a desktop automation framework for macOS. It lets you write Lua scripts that hook into operating system functionality, allowing you to interact with the keyboard/mouse, windows, displays, filesystem, and much more.

Some examples of things you can do with Hammerspoon:

- Bind hotkeys to move windows to specific locations
- Create a menu bar button that automatically lays out windows in a specific layout
- Mute your speaker when you arrive in lab (by detecting the WiFi network)
- Show you a warning if you've accidentally taken your friend's power supply

At a high level, Hammerspoon lets you run arbitrary Lua code, bound to menu buttons, key presses, or events, and Hammerspoon provides an extensive library for interacting with the system, so there's basically no limit to what you can do with it. Many people have made their Hammerspoon configurations public, so you can generally find what you need by searching the internet, but you can always write your own code from scratch.

Resources

- Getting Started with Hammerspoon
- Sample configurations
- Anish's Hammerspoon config

Booting + Live USBs

When your machine boots up, before the operating system is loaded, the BIOS/UEFI initializes the system. During this process, you can press a specific key combination to configure this layer of software. For example, your computer may say something like "Press F9 to configure BIOS. Press F12 to enter boot menu." during the boot process. You can configure all sorts of hardware-related settings in the BIOS menu. You can also enter the boot menu to boot from an alternate device instead of your hard drive.

Live USBs are USB flash drives containing an operating system. You can create one of these by downloading an operating system (e.g. a Linux distribution) and burning it to the flash drive. This process is a little bit more complicated than simply copying a `.iso` file to the disk. There are tools like UNetbootin to help you create live USBs.

Live USBs are useful for all sorts of purposes. Among other things, if you break your existing operating system installation so that it no longer boots, you can use a live USB to recover data or fix the operating system.

Docker, Vagrant, VMs, Cloud, OpenStack

Virtual machines and similar tools like containers let you emulate a whole computer system, including the operating system. This can be useful for creating an isolated environment for testing, development, or exploration (e.g. running potentially malicious code).

Vagrant is a tool that lets you describe machine configurations (operating system, services, packages, etc.) in code, and then instantiate VMs with a simple `vagrant up`. Docker is conceptually similar but it uses containers instead.

You can also rent virtual machines on the cloud, and it's a nice way to get instant access to:

- A cheap always-on machine that has a public IP address, used to host services
- A machine with a lot of CPU, disk, RAM, and/or GPU
- Many more machines than you physically have access to (billing is often by the second, so if you want a lot of computing for a short amount of time, it's feasible to rent 1000 computers for a couple of minutes)

Popular services include Amazon AWS, Google Cloud, Microsoft Azure, DigitalOcean.

If you're a member of MIT CSAIL, you can get free VMs for research purposes through the CSAIL OpenStack instance.

Notebook programming

Notebook programming environments can be really handy for doing certain types of interactive or exploratory development. Perhaps the most popular notebook programming environment today is Jupyter, for Python (and several other languages). Wolfram Mathematica is another notebook programming environment that's great for doing math-oriented programming.

GitHub

GitHub is one of the most popular platforms for open-source software development. Many of the tools we've talked about in this class, from vim to Hammerspoon, are hosted on GitHub. It's easy to get started contributing to open-source to help improve the tools that you use every day.

There are two primary ways in which people contribute to projects on GitHub:

- Creating an issue. This can be used to report bugs or request a new feature. Neither of these involves reading or writing code, so it can be pretty lightweight to do. High-quality bug reports can be extremely valuable to developers. Commenting on existing discussions can be helpful too.
- Contribute code through a pull request. This is generally more involved than creating an issue. You can fork a repository on GitHub, clone your fork, create a new branch, make some changes (e.g. fix a bug or implement a feature), push the branch, and then create a pull request. After this, there will generally be some back-and-forth with the project maintainers, who will give you feedback on your patch. Finally, if all goes well, your patch will be merged into the upstream repository. Often times, larger projects will have a contributing guide, tag beginner-friendly issues, and some even have mentorship programs to help first-time contributors become familiar with the project.

Q&A

For the last lecture, we answered questions that the students submitted:

- Any recommendations on learning Operating Systems related topics like processes, virtual memory, interrupts, memory management, etc
- What are some of the tools you'd prioritize learning first?
- When do I use Python versus a Bash scripts versus some other language?
- What is the difference between `source script.sh` and `./script.sh`
- What are the places where various packages and tools are stored and how does referencing them work? What even is `/bin` or `/lib`?
- Should I `apt-get install` a python-whatever, or `pip install` whatever package?
- What's the easiest and best profiling tools to use to improve performance of my code?
- What browser plugins do you use?
- What are other useful data wrangling tools?
- What is the difference between Docker and a Virtual Machine?
- What are the advantages and disadvantages of each OS and how can we choose between them (e.g. choosing the best Linux distribution for our purposes)?
- Vim vs Emacs?
- Any tips or tricks for Machine Learning applications?
- Any more Vim tips?
- What is 2FA and why should I use it?
- Any comments on differences between web browsers?

Any recommendations on learning Operating Systems related topics like processes, virtual memory, interrupts, memory management, etc

First, it is unclear whether you actually need to be very familiar with all of these topics since they are very low level topics. They will matter as you start writing more low level code like implementing or modifying a kernel. Otherwise, most topics will not be relevant, with the exception of processes and signals that were briefly covered in other lectures.

Some good resources to learn about this topic:

- MIT's 6.828 class - Graduate level class on Operating System Engineering. Class materials are publicly available.
- Modern Operating Systems (4th ed) - by Andrew S. Tanenbaum is a good overview of many of the mentioned concepts.
- The Design and Implementation of the FreeBSD Operating System - A good resource about the FreeBSD OS (note that this is not Linux).

- Other guides like Writing an OS in Rust where people implement a kernel step by step in various languages, mostly for teaching purposes.

What are some of the tools you'd prioritize learning first?

Some topics worth prioritizing:

- Learning how to use your keyboard more and your mouse less. This can be through keyboard shortcuts, changing interfaces, &c.
- Learning your editor well. As a programmer most of your time is spent editing files so it really pays off to learn this skill well.
- Learning how to automate and/or simplify repetitive tasks in your workflow because the time savings will be enormous...
- Learning about version control tools like Git and how to use it in conjunction with GitHub to collaborate in modern software projects.

When do I use Python versus a Bash scripts versus some other language?

In general, bash scripts are useful for short and simple one-off scripts when you just want to run a specific series of commands. bash has a set of oddities that make it hard to work with for larger programs or scripts:

- bash is easy to get right for a simple use case but it can be really hard to get right for all possible inputs. For example, spaces in script arguments have led to countless bugs in bash scripts.
- bash is not amenable to code reuse so it can be hard to reuse components of previous programs you have written. More generally, there is no concept of software libraries in bash.
- bash relies on many magic strings like `$?` or `$@` to refer to specific values, whereas other languages refer to them explicitly, like `exitCode` or `sys.args` respectively.

Therefore, for larger and/or more complex scripts we recommend using more mature scripting languages like Python or Ruby. You can find online countless libraries that people have already written to solve common problems in these languages. If you find a library that implements the specific functionality you care about in some language, usually the best thing to do is to just use that language.

What is the difference between `source script.sh` and `./script.sh`

In both cases the `script.sh` will be read and executed in a bash session, the difference lies in which session is running the commands. For `source` the commands are executed in your current bash session and thus any changes made to the current environment, like changing directories or defining functions will persist in the current session once the `source` command finishes executing. When running the script standalone like `./script.sh`, your current bash session starts a new instance of bash that will run the commands in `script.sh`. Thus, if `script.sh` changes directories, the new bash

instance will change directories but once it exits and returns control to the parent bash session, the parent session will remain in the same place. Similarly, if `script.sh` defines a function that you want to access in your terminal, you need to `source` it for it to be defined in your current bash session. Otherwise, if you run it, the new bash process will be the one to process the function definition instead of your current shell.

What are the places where various packages and tools are stored and how does referencing them work? What even is `/bin` or `/lib`?

Regarding programs that you execute in your terminal, they are all found in the directories listed in your `PATH` environment variable and you can use the `which` command (or the `type` command) to check where your shell is finding a specific program. In general, there are some conventions about where specific types of files live. Here are some of the ones we talked about, check the Filesystem, Hierarchy Standard for a more comprehensive list.

- `/bin` - Essential command binaries
- `/sbin` - Essential system binaries, usually to be run by root
- `/dev` - Device files, special files that often are interfaces to hardware devices
- `/etc` - Host-specific system-wide configuration files
- `/home` - Home directories for users in the system
- `/lib` - Common libraries for system programs
- `/opt` - Optional application software
- `/sys` - Contains information and configuration for the system (covered in the first lecture)
- `/tmp` - Temporary files (also `/var/tmp`). Usually deleted between reboots.
- `/usr/` - Read only user data
  - `/usr/bin` - Non-essential command binaries
  - `/usr/sbin` - Non-essential system binaries, usually to be run by root
  - `/usr/local/bin` - Binaries for user compiled programs
- `/var` - Variable files like logs or caches

Should I `apt-get install` a python-whatever, or `pip install` whatever package?

There's no universal answer to this question. It's related to the more general question of whether you should use your system's package manager or a language-specific package manager to install software. A few things to take into account:

- Common packages will be available through both, but less popular ones or more recent ones might not be available in your system package manager. In this case, using the language-specific tool is the better choice.
- Similarly, language-specific package managers usually have more up to date versions of packages than system package managers.

- When using your system package manager, libraries will be installed system wide. This means that if you need different versions of a library for development purposes, the system package manager might not suffice. For this scenario, most programming languages provide some sort of isolated or virtual environment so you can install different versions of libraries without running into conflicts. For Python, there's virtualenv, and for Ruby, there's RVM.
- Depending on the operating system and the hardware architecture, some of these packages might come with binaries or might need to be compiled. For instance, in ARM computers like the Raspberry Pi, using the system package manager can be better than the language specific one if the former comes in form of binaries and the latter needs to be compiled. This is highly dependent on your specific setup.

You should try to use one solution or the other and not both since that can lead to conflicts that are hard to debug. Our recommendation is to use the language-specific package manager whenever possible, and to use isolated environments (like Python's virtualenv) to avoid polluting the global environment.

What's the easiest and best profiling tools to use to improve performance of my code?

The easiest tool that is quite useful for profiling purposes is print timing. You just manually compute the time taken between different parts of your code. By repeatedly doing this, you can effectively do a binary search over your code and find the segment of code that took the longest.

For more advanced tools, Valgrind's Callgrind lets you run your program and measure how long everything takes and all the call stacks, namely which function called which other function. It then produces an annotated version of your program's source code with the time taken per line. However, it slows down your program by an order of magnitude and does not support threads. For other cases, the `perf` tool and other language specific sampling profilers can output useful data pretty quickly. Flamegraphs are a good visualization tool for the output of said sampling profilers. You should also try to use specific tools for the programming language or task you are working with. For example, for web development, the dev tools built into Chrome and Firefox have fantastic profilers.

Sometimes the slow part of your code will be because your system is waiting for an event like a disk read or a network packet. In those cases, it is worth checking that back-of-the-envelope calculations about the theoretical speed in terms of hardware capabilities do not deviate from the actual readings. There are also specialized tools to analyze the wait times in system calls. These include tools like eBPF that perform kernel tracing of user programs. In particular `bpftrace` is worth checking out if you need to perform this sort of low level profiling.

What browser plugins do you use?

Some of our favorites, mostly related to security and usability:

- uBlock Origin - It is a wide-spectrum blocker that doesn't just stop ads, but all sorts of third-party communication a page may try to do. This also covers inline scripts and other types of resource loading. If you're willing to spend some time on configuration to make things work, go to medium mode or even hard mode. Those will make some sites not work until you've fiddled with the settings enough, but will also significantly improve your online security. Otherwise, the easy mode is already a good default that blocks most ads and tracking. You can also define your own rules about what website objects to block.
- Stylus - a fork of Stylish (don't use Stylish, it was shown to steal users' browsing history), allows you to sideload custom CSS stylesheets to websites. With Stylus you can easily customize and modify the appearance of websites. This can be removing a sidebar, changing the background color or even the text size or font choice. This is fantastic for making websites that you visit frequently more readable. Moreover, Stylus can find styles written by other users and published in userstyles.org. Most common websites have one or several dark theme stylesheets for instance.
- Full Page Screen Capture - Built into Firefox and Chrome extension. Lets you take a screenshot of a full website, often much better than printing for reference purposes.
- Multi Account Containers - lets you separate cookies into "containers", allowing you to browse the web with different identities and/or ensuring that websites are unable to share information between them.
- Password Manager Integration - Most password managers have browser extensions that make inputting your credentials into websites not only more convenient but also more secure. Compared to simply copy-pasting your user and password, these tools will first check that the website domain matches the one listed for the entry, preventing phishing attacks that impersonate popular websites to steal credentials.
- Vimium - A browser extension that provides keyboard-based navigation and control of the web in the spirit of the Vim editor.

What are other useful data wrangling tools?

Some of the data wrangling tools we did not have time to cover during the data wrangling lecture include `jq` or `pup` which are specialized parsers for JSON and HTML data respectively. The Perl programming language is another good tool for more advanced data wrangling pipelines. Another trick is the `column -t` command that can be used to convert whitespace text (not necessarily aligned) into properly column aligned text.

More generally a couple of more unconventional data wrangling tools are vim and Python. For some complex and multi-line transformations, vim macros can be a quite invaluable tool to use. You can just record a series of actions and repeat them as many times as you want, for instance in the editors lecture notes (and last year's video) there is an example of converting an XML-formatted file into JSON just using vim macros.

For tabular data, often presented in CSVs, the pandas Python library is a great tool. Not only because it makes it quite easy to define complex operations like group by, join or filters; but also makes it quite easy to plot different properties of your data. It also supports exporting to many table formats including XLS, HTML or LaTeX. Alternatively the R programming language (an arguably bad programming language) has lots of functionality for computing statistics over data and can be quite useful as the last step of your pipeline. ggplot2 is a great plotting library in R.

What is the difference between Docker and a Virtual Machine?

Docker is based on a more general concept called containers. The main difference between containers and virtual machines is that virtual machines will execute an entire OS stack, including the kernel, even if the kernel is the same as the host machine. Unlike VMs, containers avoid running another instance of the kernel and instead share the kernel with the host. In Linux, this is achieved through a mechanism called LXC, and it makes use of a series of isolation mechanisms to spin up a program that thinks it' s running on its own hardware but it' s actually sharing the hardware and kernel with the host. Thus, containers have a lower overhead than a full VM. On the flip side, containers have a weaker isolation and only work if the host runs the same kernel. For instance if you run Docker on macOS, Docker needs to spin up a Linux virtual machine to get an initial Linux kernel and thus the overhead is still significant. Lastly, Docker is a specific implementation of containers and it is tailored for software deployment. Because of this, it has some quirks: for example, Docker containers will not persist any form of storage between reboots by default.

What are the advantages and disadvantages of each OS and how can we choose between them (e.g. choosing the best Linux distribution for our purposes)?

Regarding Linux distros, even though there are many, many distros, most of them will behave fairly identically for most use cases. Most of Linux and UNIX features and inner workings can be learned in any distro. A fundamental difference between distros is how they deal with package updates. Some distros, like Arch Linux, use a rolling update policy where things are bleeding-edge but things might break every so often. On the other hand, some distros like Debian, CentOS or Ubuntu LTS releases are much more conservative with releasing updates in their repositories so things are usually more stable at the expense of sacrificing newer features. Our recommendation for an easy and stable experience with both desktops and servers is to use Debian or Ubuntu.

Mac OS is a good middle point between Windows and Linux that has a nicely polished interface. However, Mac OS is based on BSD rather than Linux, so some parts of the system and commands are different. An alternative worth checking is FreeBSD. Even though some programs will not run on FreeBSD, the BSD ecosystem is much less fragmented and better documented than Linux. We discourage Windows for anything

but for developing Windows applications or if there is some deal breaker feature that you need, like good driver support for gaming.

For dual boot systems, we think that the most working implementation is macOS' bootcamp and that any other combination can be problematic on the long run, specially if you combine it with other features like disk encryption.

Vim vs Emacs?

The three of us use vim as our primary editor but Emacs is also a good alternative and it' s worth trying both to see which works better for you. Emacs does not follow vim' s modal editing, but this can be enabled through Emacs plugins like Evil or Doom Emacs. An advantage of using Emacs is that extensions can be implemented in Lisp, a better scripting language than vimscript, Vim' s default scripting language.

Any tips or tricks for Machine Learning applications?

Some of the lessons and takeaways from this class can directly be applied to ML applications. As it is the case with many science disciplines, in ML you often perform a series of experiments and want to check what things worked and what didn' t. You can use shell tools to easily and quickly search through these experiments and aggregate the results in a sensible way. This could mean subselecting all experiments in a given time frame or that use a specific dataset. By using a simple JSON file to log all relevant parameters of the experiments, this can be incredibly simple with the tools we covered in this class. Lastly, if you do not work with some sort of cluster where you submit your GPU jobs, you should look into how to automate this process since it can be a quite time consuming task that also eats away your mental energy.

Any more Vim tips?

A few more tips:

- Plugins - Take your time and explore the plugin landscape. There are a lot of great plugins that address some of vim' s shortcomings or add new functionality that composes well with existing vim workflows. For this, good resources are VimAwesome and other programmers' dotfiles.
- Marks - In vim, you can set a mark doing `m<X>` for some letter `X`. You can then go back to that mark doing `'<X>`. This lets you quickly navigate to specific locations within a file or even across files.
- Navigation - `Ctrl+O` and `Ctrl+I` move you backward and forward respectively through your recently visited locations.
- Undo Tree - Vim has a quite fancy mechanism for keeping track of changes. Unlike other editors, vim stores a tree of changes so even if you undo and then make a different change you can still go back to the original state by navigating the undo tree. Some plugins like gundo.vim and undotree expose this tree in a graphical way.

- Undo with time - The `:earlier` and `:later` commands will let you navigate the files using time references instead of one change at a time.
- Persistent undo is an amazing built-in feature of vim that is disabled by default. It persists undo history between vim invocations. By setting `undofile` and `undodir` in your `.vimrc`, vim will store a per-file history of changes.
- Leader Key - The leader key is a special key that is often left to the user to be configured for custom commands. The pattern is usually to press and release this key (often the space key) and then some other key to execute a certain command. Often, plugins will use this key to add their own functionality, for instance the UndoTree plugin uses `<Leader> U` to open the undo tree.
- Advanced Text Objects - Text objects like searches can also be composed with vim commands. E.g. `d/<pattern>` will delete to the next match of said pattern or `cgn` will change the next occurrence of the last searched string.

What is 2FA and why should I use it?

Two Factor Authentication (2FA) adds an extra layer of protection to your accounts on top of passwords. In order to login, you not only have to know some password, but you also have to "prove" in some way you have access to some hardware device. In the most simple case, this can be achieved by receiving an SMS on your phone, although there are known issues with SMS 2FA. A better alternative we endorse is to use a U2F solution like YubiKey.

Any comments on differences between web browsers?

The current landscape of browsers as of 2020 is that most of them are like Chrome because they use the same engine (Blink). This means that Microsoft Edge which is also based on Blink, and Safari, which is based on WebKit, a similar engine to Blink, are just worse versions of Chrome. Chrome is a reasonably good browser both in terms of performance and usability. Should you want an alternative, Firefox is our recommendation. It is comparable to Chrome in pretty much every regard and it excels for privacy reasons. Another browser called Flow is not user ready yet, but it is implementing a new rendering engine that promises to be faster than the current ones.