# Potpourri

## Table of Contents



## 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.