Development Tools

Muchang Bahng

Fall 2024

Contents

11 Hardware   **89**

This serves as a quick wiki on the different functionalities of the tools I use. They are for the most part mutually exclusive.

1   Text Editing with Neovim

The first thing you do when coding is typing something, and this requires a text editor. Vim is guaranteed to be on every Linux system, so there is no need to install it. However, you may have to install Neovim (which is just a command away). Vim can be a really big pain in the ass to learn, but I got into it when I was watching some video streams from a senior software engineer at Netflix called The Primeagen. He moved around the code like I've never seen, and I was pretty much at the limit of my typing speed, so I decided to give it a try during the 2023 fall semester. My productivity plummetted during the first 2 days (which was quite scary given that I had homework due), but within a few weeks I was faster than before, so if you have the patience, I would recommend learning it. Here is a summary of reasons why I would recommend learning Vim:

1. It pushes you to know the ins and outs of your editor. As a mechanic with his tools, a programmer should know exactly how to configure their editor.

2. The plugin ecosystem is much more diverse than other editors such as VSCode. You can find plugins/extensions for everything. Here is a summmary of them here.

3. You're faster. If you're going to be coding for say the next 10 years, then why not spend a month to master something that will make you faster by 10%? That way, you'll have coded 1 years worth more with a 1 month commitment. I'd take a free 11 months of coding any day.

4. Computing clusters and servers will be much easier to navigate since they all run Linux with Vim.

5. Vim is lightweight, and you don't have to open up VSCode every time you want to edit a configuration file.

Example 1.1 (Vim vs Neovim)

Experience wise, Vim and Neovim are very similar, and if you configure things rihght, you may not even be able to tell the difference. But there are 3 differences that I want to mention:
1. Neovim can be configured in Lua, which is much cleaner than Vimscript.
2. Neovim provides mouse control right out of the box, which is convenient for me at times and can be easier to transition into, while Vim does not provide any mouse support.
3. There are some plugins that are provided in Neovim that are not in Vim.
Either way, the configuration is essentially the same. At startup, the text editor will parse some predetermined configuration file and load those settings.

It may be the case that a remote server does not have neovim installed, or you may not have the permissions to install it. In this case, you can use **sshfs**, which is a file system client based on the SSH File Transfer Protocol. It allows you to mount a remote directory over SSH.

1.1   Configuration Files

In Vim, your configuration files are located in `/.vimrc` and plugins are located in `/.vim/`. In here, you can put in whatever options, keymaps, and plugins you want. All the configuration is written in VimScript.

```
1   # options
2   filetype plugin indent on
3   syntax on
4   set background=dark
5   set expandtab ts=2 sw=2 ai
6   set nu
7   set linebreak
8   set relativenumber
9
10  # keymaps
```

```
11   inoremap <C-j> <esc>dvbi
12   inoremap jk <esc>
13   nnoremap <C-h> ge
14   nnoremap <C-l> w
```

In Neovim, I organize it using Lua. It essentially looks for the `/.config/nvim/init.lua` file and loads the options from there. We also have the option to import other Lua modules for better file structure with the `require` keyword. The tree structure of this configuration file should be the following below. The extra `user` director layer is necessary for isolating configuration files on multiple user environments.

The init file is the "main file" which is parsed first. I generally don't put any explicit options in this file and reserve it only for require statements. It points to the following (group of) files:

1. **options.lua**: This is where I store all my options.

2. **keymaps.lua**: All keymaps.

3. **plugins.lua**: First contains a script to automatically install packer if it is not there, and then contains a list of plugins to download.

4. **Plugin Files**: Individual configuration files for each plugin (e.g. if I install a colorscheme plugin, I should choose which specific colorscheme I want from that plugin).

5. **Filetype Configuration Files**: Options/keymaps/plugins to load for a specific filetype. This helps increase convenience and speed since I won't need plugins like VimTex if I am working in JavaScript.

Once you have your basic options and keymaps done, you'll be spending most of your time experimenting with plugins. It is worth to mention some good ones that I use.

1. **Packer** as the essential package manager.

2. **Plenary**

3. **Telescope** for quick search and retrieval of files.

4. **Indent-blankline** for folding.

5. **Neoformat** for automatic indent format.

6. **Autopairs** and **autotag** to automatically close quotation marks and parantheses.

7. **Undotree** to generate and navigate undo history.

8. **Vimtex** for compilation of LaTeX documents.

9. **Onedark** and **Oceanic Next** for color schemes.

10. **Vim-Startify** for nice looking neovim startup.

11. **Comment** for commenting visual blocks of code.

It is also worthwhile to see how they are actually loaded in the backend. Each plugin is simply a github repo that has been cloned into `/.local/share/nvim/site/pack/packer/`, which contains two directories. The packages in `start/` are loaded up every time Neovim starts, and those in `opt/` are packages that are loaded up when a command is called in a certain file (known as lazy loading). Therefore, if you have any problems with Neovim, you should probably look into these folders (and possibly delete them and reinstall them using Packer if needed).

1.2   Troubleshooting

A good test to run is `:checkhealth`, which checks for any errors or warnings in your Neovim configuration. You should aim to have every (non-optional) warning cleared, which usually involves having to install some package, making it executable and/or adding to `$PATH`.

If you are getting plugin errors, you can also manually delete the plugin directory in 'pack/packer' and run 'PackerInstall' to re-pull the repos. This may help.

1.3 Language Service Providers

If you were to create a text editor from scratch, you would first want to make a buffer and some external program to analyze this buffer (plus some other text files) concurrently. Things like autocompletion, type checking, and syntax checking may all be taken for granted, but it's not, and these are all provided by the **language service provider**, also known as **LSP**. LSPs are specific to each language, such as `pyright` being the mainstream LSP for Python, and `ts_ls` for TypeScript. Some of its services have specific names, and overlap a lot.

1. *Autocompleting* partially typed words with suggestions based on what you typed so far in the current buffer, or from analyzing existing paths of various directories/files.

2. *Linting*, which is a general term for finding issues in your code.

3. *Type checking* the correct types of variables to find bugs or edge cases in your code.

4. *Symbol searching* variables so that you can jump to where they are declared or defined.

The tricky part about LSPs is that they can get quite heavy in computation. For modern laptops this isn't really a problem. For example, on my Macbook Pro M3 I can have a heavy type checker, full autocompletion of every word, symbol searching of every variable, and linting across *all* files in my current directory (of up to 50 files), all with no noticeable delay. This was quite nice, until I started working on a remote server offering 4 crappy CPUs to work off of, and this just made coding impossible since all of these processes caused a 1 second delay in my writing. Therefore, depending on where you work, LSPs should be lightweight. The balance between functionality and performance is what I think VSCode does very well compared to Neovim.

1.4 Snippets

2   Documentation with LaTeX

Latex is a great way to take notes, and while it may be tedious to write it at first, it's a skill that you build up just like when you first wrote the alphabet in kindergarten. If you think you'll be slow at latex forever, don't worry. In a few months I was faster at taking notes with latex than by hand or google docs.

I still handwrite notes though during class. Some people handwrite because it helps with retention, but I do it because I often need to reorganize the structure and content of my notes multiple times before I have a clear picture of the entire course. My general system is to handwrite notes for about a month's worth of classes, and then spend a few days thinking about them and adding them to my website.

Now let's get back to latex. Most users write on Overleaf, which is a platform that makes latex easier by having all the packages you'll ever need on the cloud, along with a user-friendly GUI. Everything is preconfigured, but that means that your work environment can't really be tailored to the way you like it. I've used Overleaf for about 3 years before I started using Vim, and the lack of Vim keybindings on Overleaf just made me write latex on my local desktop. This allows me to have all my files locally, which I can then store in some remote Github repository.

I use the Neovim plugin **VimTex**, which is installed in my `plugins.lua` with `use lervag/vimtex`. Then, you want to install TexLive, which is needed to compile tex files and to manage packages. The directions for TexLive installation is available [here](https://tug.org/texlive/quickinstall.html). Once I downloaded the install files, I like to run `sudo perl ./install-tl -scheme=small`. Be careful with the server location (which can be set with the `-location` parameter), as I have gotten some errors. I set `-scheme=small`, which installs about 350 packages compared to the default scheme, which installs about 5000 packages (7GB). I also did not set `-no-interaction` since I want to slightly modify the `-texuserdir` to some other path rather than just my home directory.

2.1   TLMGR

Once you installed everything, make sure to add the binaries to PATH, which will allow you to access the **tlmgr** package manager, which pulls from the CTAN (Comprehensive TeX Archive Network) and gives VimTex access to these executables. Unfortunately, the small scheme installation does not also install the **latexmk** compiler, which is recommended by VimTex. We can simply install this by running

```
1   sudo tlmgr install latexmk
```

Now run `:checkhealth` in Neovim and make sure that everything is OK, and install whatever else is needed.

To install other Latex packages (and even document classes), we can use tlmgr. All the binaries and packages are located in `/usr/loca/texlive/202*/` and since we're modifying this, we should run it with root privileges. The binaries can also be found here. Let's go through some basic commands:

1. List all available packages: `tlmgr list`

2. List installed packages: `tlmgr list -only-installed` (the packages with the 'i' next to them are installed)

3. Install a package and dependencies: `sudo tlmgr install amsmath tikz`

4. Reinstall a package: `sudo tlmgr install amsmath -reinstall`

5. Remove a package: `sudo tlmgr remove amsmath` More commands can be found here for future reference.

2.2   PDF Viewers

I will already assume you have a PDF viewer installed. Many operating systems come with their own default PDF viewers, such as Preview for MacOS, Adobe/Edge for Windows, or some other for Ubuntu. So what should you look for in a PDF viewer? That depends on your needs, but as of May 2025 here are mine.

1. *Customization of keyboard shortcuts.* For example, when I want to scroll down 10 pages, I want to be able to use some keymaps to go there rather than scrolling with my mouse wheel.

2. Support for dark mode. Text is generally black on a white background, but this can hurt the eyes especially at night when my room is darker.

3. Modifying and viewing comments are nice when people are annotating or editing my work. This is especially useful in collaborative research or when doing literature reviews.

4. Ability to fill/sign PDF forms. Not as important but is nice for e-signing.

On Arch Linux I use **zathura**, which is lightweight and also comes with vim motions for navigation. On Mac the most similar is **Skim**, which also has keybindings and supports a dark mode[1].

2.3   Compilation and Debugging

From the moment you compile a latex file, there are several files that are generated before the final PDF renders. Let's start with the most basic ones. T

Definition 2.1 (`paper.log`)

Log file of the compilation. You should first check this when debugging.

Definition 2.2 (`paper.aux`)

This is used for reference information. Mainly, if you use
`ref{}` but the corresponding
`label{}` is only later in the document, LaTeX has not seen it and won't look ahead. Instead, each time there is a
`label{}`, a command gets written to the `.aux` file. Next time you compile your `.tex`, that `.aux` is inputted right before the document gets started, and the command you had written to it then registers the value for
`ref{}`.

Definition 2.3 (`paper.synctex.gz`)

This specifies the synchronization between your PDF and source files. In other words, it allows you to click in the PDF to go to the corresponding line in your source code and vice versa. This can be safely deleted.

Definition 2.4 (`paper.out`)

This is used by `hyperref` to store a list of PDF bookmarks.

Now there may or may not exist, depending on the packages you use.

Definition 2.5 (`paper.toc`)

If you use
`tableofcontents`, this will create a `.toc` file.

---

[1]Under Skim, Settings, PDF Display, Invert Colors for Dark Mode

> **Definition 2.6 (`paper.bbl/blg`)**
>
> The bbl doc is outputted by biber (or bibtex) and contains the prepared data to be used by biblatex (or natbib). Then the blg is the log files of biber/bibtex.

> **Definition 2.7 (`paper.fls`)**
>

2.4   Macros

2.5   Figures and Tikz

After this, you can install Inkscape, which is free vector-based graphics editor (like Adobe Illustrator). It is great for drawing diagrams, and you can generate custom keymaps that automatically open Inkscape for drawing diagrams within LaTeX, allowing for an seamless note-taking experience.

3   Version Control with Git

Git is a pretty complex version control tool. It allows you to perform different actions. We'll go over them, starting with the most basic to the most complex. In order to learn this, we should know the structure of the git history.

3.1   Local Git Repository

When you do `git init` in a repository, you are essentially saying that you want to keep track of the history of this repository. This can obviously be done with an undo tree, which comes out-of-box in almost all text editors, but it is much more powerful.

Definition 3.1 (Local Git Tree)

The history of our repository is essentially a tree, with each node representing some edits composed of
1. adding a new file
2. modifying a file
3. deleting a file

Each node is represented by a hash generated from its previous node and the corresponding edits. You can see your history using

```
1    git log
```

`HEAD` is a pointer to the node that reflects the state of your current repository (minus your uncommitted edits), which is usually the most recent node.

Unlike most undo trees, these nodes are not added automatically. You must add them manually through a 2-step process.

Definition 3.2 (Stage)

You want to take a set of edits and **stage** them. This essentially tells git that these staged files/lines are going to be a part of the next node.

Definition 3.3 (Commit)

Then you commit your changes, which does the following.
1. This takes all of your staged changes and packages them in a node A.
2. It looks at `HEAD`, uses `HEAD`'s hash to generate the hash of A, and appends A to `HEAD` by having A point to `HEAD`.[a]
3. It moves `HEAD` to A.

---
[a]So nodes actually point to *previous nodes*.

Therefore, when you make your first commit, you are creating a genesis node from which every other edit will be based off of. Your `HEAD` then points to this commit. This is great start, and let's add more functionality.

Definition 3.4 (Checkout a Commit)

You can move `HEAD` to point to a specific commit by using

```
1    git checkout <commit-hash>    # point to this commit
2    git checkout HEAD~N           # point to the commit $N$ nodes before HEAD
```

This leaves you in a `detached head state`, which means that your head is not pointing to the end node. This is useful if you want to
1. *explore the codebase at a commit's snapshot in time.*

Note that so far, we have described git as a linked list plus some extra head pointer. Adding to this linked list is easy since we are simply adding new edits, but deleting can be very tricky. We will first introduce how to delete the most recent K commits, which is the easiest way to delete.

Definition 3.5 (Reset)

Say that your history is
$$(A) \leftarrow (B) \leftarrow (C) \leftarrow (H \mapsto D) \tag{1}$$
If we want to throw away commits C and D, we can **reset** to B, which deletes C, D and has H point to B, giving us
$$(A) \leftarrow (H \mapsto B) \tag{2}$$
1. A **soft reset** means that the edits introduced in C and D will still be kept as unstaged changes, and so you may use them as a starting point to make your next commit.
2. A **hard reset** means that the edits are also completely deleted.

Most beginners in git really know these commands when working with their history, but this is really just a glorified stack. The additional operations can be daunting because they have the risk of introducing *conflicts*.

3.2 Conflicts

Definition 3.6 (Conflicts)

A **conflict** arises when two commits contain edits that change some location independently at the same time. They occur most frequently when working with multiple branches, but they can happen even when working on a single branch. Git will tell you when there is conflict between commits C and C' at a certain location. At this point, you will have to manually go to that location and compare the changes introduced in C and C', called **hunks**. The conflict looks generally like this.

```
1   ... some code above
2   <<<<< (C)    # hunk 1
3   ========
4   >>>>> (C')  # hunk 2
5   ... some code below
```

In order to fix this conflict, you can
1. select hunk 1 (and ignore hunk 2)
2. select hunk 2
3. select both hunks (i.e. incorporate both edits)
4. manually delete the », ===, « and directly edit the file to make a custom change that overrides both hunks.

Choosing the option to fix a conflict may sometimes be complicated, since you may not always want to select the hunk reflected in your most recent changes, because doing that might introduce another conflict in a later commit that actually modified the old code into the new code.

Definition 3.7 (Revert Commit)

Say that you have history
$$(C_1) \leftarrow (C_2) \leftarrow (C_3) \leftarrow (H \mapsto C_4) \tag{3}$$

You can choose to **revert** and of the 4 commits above. Given any commit C, reverting a commit means that you simply add a new commit C' with the changes that are the exact opposite of C. If we want to revert commit C_2, our history looks like
$$(C_1) \leftarrow (C_2) \leftarrow (C_3) \leftarrow (C_4) \leftarrow (C_2') \tag{4}$$

So really, we are "deleting" our history by adding.

Example 3.1 (Conflicts in Reverting)

Say that you have history
$$(C_1) \leftarrow (C_2) \leftarrow (C_3) \leftarrow (H \mapsto C_4) \tag{5}$$

If you try to revert H, this is fine and will never have conflicts. Say that you made an edit in (C_3) where you added $x = 4$ to some python script, and then you removed this line in (C_4). Then if you add (C_3') to undo it, it tries to delete a line that isn't even there! Therefore you will get a conflict that looks something like

```
1  <<<<< (C4)    # hunk 1
2  - x = 4
3  ========
4  - x = 4
5  >>>>> (C3')  # hunk 2
```

Obviously you can just select either one of the hunks to get what you want.

Conflicts are unavoidable and you will have to get comfortable with them.

Definition 3.8 (Amending a Commit)

If you have some staged edits and you decide that these edits should go into some previous commit rather than a new one, you can **amend** the old commits. In lazygit, you can stage which edits you want to amend with, then go to the commit in your working branch and press `<shift-a>` to amend it.

3.3　Interactive Rebasing

Even though we can revert commits, we haven't actually found out how to truly *delete* a commit from your history which modifies
$$(A) \leftarrow (B) \leftarrow (C) \leftarrow (D) \tag{6}$$

to something like
$$(A) \leftarrow (B) \leftarrow (D) \tag{7}$$

Definition 3.9 (Rebasing)

Essentially, we want to *directly* (unlike a revert) modify our history that goes *beyond* (unlike a reset) the last K commits. Any actions that modifies the history is known as **rebasing**, which can be done automatically by git (regular rebasing just picks all commits) but must often be done **interactively**, which allows for more operations listed below. When you want to start an interactive rebase, you want

---

to tell git from which commit C_s you want to start the interactive rebase on.

```
1  git rebase -i <start commit hash>
```

You are saying that from commit C_s and beyond until the end C_n, I may arbitrarily modify them, but commits previous to C_s will be untouched. When you do this, all commits C_i where $i \geq s$ will be shown as below.

Figure 1: Interactive rebase shown in LazyGit.

There are a fixed set of supported operations allows in an interactive rebase.[a]
1. **Pick**. This just means that you are leaving the commit alone, i.e. picking it to be in the rebase.
2. **Reword**. Just edits the commit message.
3. **Squash**. Given commit $C_i \leftarrow C_{i+1}$, you can label C_{i+1} with `squash` to merge it into C_i, turning 2 nodes into one. This almost never causes conflicts. The new commit message is just those of C_i, C_{i+1} concatenated.
4. **Fixup**. Like squash, but discard the commit's message.
5. **Drop**. This deletes a commit and removes it entirely.
6. **Break**. Stop at this commit to edit it. I think you can change which edits you have committed, choose which edits to keep, and choose which edits to remove (back into your unstaged changes).
7. **Edit**. Stop at this commit to amend it.
8. You can also swap commits by editing the text file so that the commits are in a different order.

```
1  # Original order in rebase editor:
2  pick abc123 First commit
3  pick def456 Second commit
4
5  # After swapping lines in editor:
6  pick def456 Second commit
7  pick abc123 First commit
```

After you edit the rebase text file and continue the rebase, git will do the following sequentially:
1. `HEAD`, which pointed at C_n, will point towards C_s.
2. While `HEAD` is pointing at $C_i \neq C_n$ (i.e. not at the end), we do the following.
    (a) It will attempt to perform all the operations you have specified for the next commit C_{i+1}.
    (b) If the operations are finished, we increment `HEAD` to point to C_{i+1} and continue.
    (c) If there is a conflict, it will pause, state that there are conflicts between `HEAD` $= C_i$ and C_{i+1}, and ask you to resolve them. Once resolved it will continue.
3. Then we are done with the rebase since we have went through all commits, modified them, and resolved all conflicts.

Interactive rebasing is an extremely powerful way to modify your commit history, and it's probably the operation where you'll spend the most time on git.

---

[a]Note that pick and reword will never cause conflicts. Squash and fixup will most likely not cause conflicts. Drop, break, edit, and swapping may cause conflicts.

Again, note that if you have changed anything in commit C_i, then the hash of C_i every C_j after will get changed. This causes git to interpret these changed commits as completely new ones, even if we only picked

a given commit without any modifications. For single-branch rebases, this is fine, but this causes some nasty problems when rebasing over multiple branches, as we will talk about later.

Definition 3.10 (Patching)

An easier way to modify your edits in old commits is through **patching**.[a] Within a commit, a **patch** is simply a diff file that you can add and remove to. It's like having a mini-staging area in a commit. When you have selected the different files/lines you have added to your patch, you can either choose to:
1. Remove them from the commit.
2. Add them to another commit.
3. Move them from the original commit to a new commit.

To do this, go into your lazygit, select a commit, and start staging files or lines. Then, you can press `T` to bring up patch options, where you can take these staged changes and put them into a new commit.

---
[a]Though patching really just does an interactive rebase in the backend.

Theorem 3.1 (Splitting Commits Into Two Different Commits)

If you want to split commits,
1. create a dummy commit
2. go to the commit you want to split, get its patches, and add them to the dummy commit.
3. Do an interactive rebase to swap the dummy commits down until you have it at the desired location.

3.4 Branches

Okay, so we now have much better control over our git history, but we've only been treating our history as a linked list. In order to introduce the tree structure, we need to introduce the *branch*. This is especially important if we have a particular previous commit $C_{k<n}$ where we would like to make some different changes to, giving us **diverging histories** with next nodes $C_k \leftarrow C_{k+1}$ and $C_k \leftarrow C_{k'+1}$.

Definition 3.11 (Branch)

A **branch** is a path from the root commit to any leaf node. It represents a unique history from genesis to `HEAD`. To list all branches, use

```
1   >> git branch
2     feature/threading
3   * main
4     test/tensor
```

The asterisk represents which branch you are currently on. The first branch you start off with is a special branch called **main**, or **master** branch.

Therefore, really our linked-list history is a git tree with a single branch.

Definition 3.12 (Creating/Switching Branches)

From any (main or non-main) branch you can create new branches by choosing the commit to split from.
1. Create a new branch from HEAD of current branch.

```
1   git branch <new-branch-name>
```

2. Create a new branch from certain commit of current branch.

```
1   git branch <new-branch-name> <commit-hash>
```

3. To switch to another branch

```
1   git checkout <branch>
```

3.4.1 Working Between Branches

If you are simultaneously working on multiple branches, you may have to checkout/switch between branches frequently. Often, you may have uncommitted changes before you checkout, and git does not allow you to do this. Therefore, we can *stash* them.

Definition 3.13 (Stash)

Stashing changes mean that you can take uncommitted changes and store them in a temporary node but not have it point to any existing commit in a branch. This allows you to save your changes without having to commit incomplete work to a branch, and you can pop them back whenever you need.

Sometimes, you may want to just copy a commit from one branch to another. You can do this using an interactive rebase, but this may be overkill since it is mainly used to work with a sequence of commits.

Definition 3.14 (Cherry-Picking and Pasting)

You can do copy a commit by **cherry picking** it and **pasting** it somewhere else.

3.4.2 Integrating Branches

The reason you want to have different branches is so that you have independent workflows that may hopefully be integrated into the master branch. So how does one actually perform this integration? There are two general ways to do this: a 3-way merge or a rebase. For both methods, we will use this example.

```
1   main      : A1 --- A2 --- A3
2                               \
3   feature1 :                  B1 --- B2 --- B3
4                                               \
5   feature2 :                  C1 --- C2 --- C3
```

Definition 3.15 (Fast-Forward Merge)

If you want to merge `main` and `feature1`, notice that `feature1` is really just ahead of `main` by some number of commits. The easiest way to merge is to add the additional commits in `feature1` to `main`. This is called a **fast-forward merge**, which we can call using

```
1   git checkout main
2   git merge --ff-only feature1
```

Doing so will result in

```
1   main     : A1 --- A2 --- A3 --- B1 --- B2 --- B3
2                              \
3   feature1 :                  --- B1 --- B2 --- B3
4                                    \
5   feature2 :                        --- C1 --- C2 --- C3
```

and we can delete `feature1` since it's not needed.

In fact, if a fast-forward merge is possible, then calling `git merge feature1` will automatically do a fast-forward. We can explicitly set it to only attempt or never attempt fast-forward by adding the `-ff-only` or `-no-ff` flags.

Definition 3.16 (3-Way Merge)

If you want to merge two divergent branches, e.g. `feature1` and `feature2`, then a fast-forward is not possible. Rather, you want to choose to merge `feature2` *into* `feature1`. Git will rather do a **three-way merge** between the divergent node `A3` and the heads of the respective branches `B3` and `C3`. After you resolve conflicts, the tree should look something like

```
1   main     : A1 --- A2 --- A3
2                              \
3   feature1 :                  --- B1 --- B2 --- B3 --- M1
4                                    \                  /
5   feature2 :                        --- C1 --- C2 --- C3 ---
```

Note that we could choose to merge `feature1` into `main` subsequently, resulting in both feature branches merged.

```
1   main     : A1 --- A2 --- A3 ------------------------------ M2
2                              \                               /
3   feature1 :                  --- B1 --- B2 --- B3 --- M1 ---
4                                    \                  /
5   feature2 :                        --- C1 --- C2 --- C3 ---
```

Note that a three-way merge may result in a pretty ugly tree, especially if we are working with dozens of branches. What we would like to do is a three-way merge in a fashion that *looks like* a fast-forward merge. That is, we want the main branch to have a linear structure rather than a series of diverging and converging nodes. In fact, we already have the tools to do this. Let's revisit the interactive rebase again. We have seen that we can do an interactive rebase from a start commit by doing

```
1   git rebase -i <start-commit-hash>
```

What we would like to do is to rebase from a commit in a different branch.

Definition 3.17 (Rebase)

If we want to linearly merge `feature2` into `feature1`, this is called "**rebasing `feature2` *onto* `feature1`.**" We run

```
1   git checkout feature2
2   git rebase feature1
```

---

which means "take my current branch's unique commits and replay them on top of whatever branch I am rebasing on (in here, `feature1`)." This will result in

```
1  main     : A1 --- A2 --- A3
2                              \
3  feature1 :                   B1 --- B2 --- B3
4                                                \
5  feature2 :                                     C1' --- C2' --- C3'
```

where the `C'` are the same commits but with different hashes since they start from a different parent.

Example 3.2 (Updating Feature Branch with Changes from Main)

A common workflow you would do in a large project with multiple developers is as follows. Consider that you are working on `feature1` and another developer is working on `feature2`.

```
1  main     : A1 --- A2 --- A3
2                              \
3  feature1 :                   --- B1 --- B2 --- B3
4                                \
5  feature2 :                     --- C1 --- C2
```

Your friend pushes their changes to `main`, which leads to this structure.

```
1  main     : A1 --- A2 --- A3 --- C1 --- C2
2                              \
3  feature1 :                   --- B1 --- B2 --- B3
4                                \
5  feature2 :                     --- C1 --- C2
```

Your branch has diverged from main, so you will need to rebase your own branch onto main. You checkout to `feature1` and run `git rebase main`. After settling conflicts, your branch will look like the following, updated with the most recent commits from `main`.

```
1  main     : A1 --- A2 --- A3 --- C1 --- C2
2                                          \
3  feature1 :                               --- B1' --- B2' --- B3'
4                                \
5  feature2 :                     --- C1 --- C2
```

Example 3.3 (Converting a Merge Into Rebase)

Say that you already merged `feature1` and `main`.

```
1  main     : A1 --- A2 --- A3 ----------------------- M1
2                              \                       /
3  feature1 :                   --- B1 --- B2 --- B3 ---
```

You realized that you actually wanted to do a fast-forward so that it looks linear! How do you do this?
   1. You first undo the merge.

```
1  git checkout main
2  git reset --hard A3
```

2. Then do the rebase of `feature1` onto `main`.

```
1   git checkout feature1
2   git rebase main
```

3. Then fast-forward to main to include `feature1`'s commits.

```
1   git checkout main
2   git merge --ff-only feature1
```

We will have this in the end.

```
1   main     : A1 --- A2 --- A3 --- B1' --- B2' --- B3'
2   feature1 : A1 --- A2 --- A3 --- B1' --- B2' --- B3'
```

3.5 Remote Trees

So far, we've talked about how you can use git to keep track of your edit history locally, but another benefit is to store these changes in the cloud. This is done through a third-party provider, and they are completely separate entities from git. The three most dominant ones are

1. *Github.* Owned by Microsoft and is the default for most open-source projects, with 100 million users.

2. *Gitlab.* Owned by Gitlab and is slowly gaining popularity due to better control of repositories and after Microsoft acquired github.

3. *Bitbucket.* Owned by Atlassian and used for private repositories in enterprise settings.

Again, all three platforms still use *git*, but the cloud storage is managed separately. All of these platforms provide a remote server that stores all of these git histories of millions of repositories around the world. The motivation behind the need of a remote workspace is that it is a common ground in which many developers can communicate and track the progress of their entire repository.

Definition 3.18 (Remote Repository)

The first step to setting up a cloud-based git tree is to place it on some server (IP address) in some directory with the proper permissions. This remote location containing the git tree and the corresponding code is called the **remote repository**, and it is encoded in either a URL or a SSH host link of the form

```
1   https://github.com/user/repo.git
```

For now, we will consider a local repository having at most 1 remote repository that it can communicate to.[a] Conventionally, the primary[b] remote repository goes under the alias **origin**.

---
[a]When we get to forking, we will talk about multiple remote repositories.
[b]Again, for now it's really the only remote.

None of the commands that we have introduced so far does anything to the remote repository. The first command we should know is how to set up one from scratch.

Definition 3.19 (Add Remote)

Given that git is initialized, we can initialize a corresponding remote repository by running

```
1   git remote add <remote-name> <remote-url>
```

Again, conventionally the `<remote-name>` is put as `origin`.

Great, we have set a remote repository up, but there is an extra step to do. Most synchronizations happens at a branch level rather than the whole tree itself, so what we would need to do for each local branch is create a corresponding remote branch and then *connect* those two so that git knows which branches to sync together. The local branch is called the **downstream** branch and its corresponding remote branch is called **upstream**.

In order to understand how the process of setting upstream branches is done, we introduce another variable in our local repository. In our local git tree, we have stated that for each branch `B`, there is a `HEAD` pointer living at the most recent commit, denoted as `B/HEAD` or just `B`.

Definition 3.20 (Remote References)

There is actually a second pointer called the **remote reference (ref)** in the local branch called `origin/B` (more generally `<remote>/<branch>`), which is a symbolic link that points to the head of the remote branch. They are git's way of keeping track of the state of branches in your remote repositories.

For some local branch, the existence of a remote ref tells you where the corresponding upstream branch is located (i.e. at `<remote>/<branch>`). If you do not see the remote ref, this means that you have not yet connected your local branch to its remote upstream, which can happen if the remote counterpart doesn't exist (i.e. you created a completely new branch) or you have never connected it. You can find all local branches and the remote references by calling

```
1   git branch -vv
2   # Shows all branches with their tracking info
3   # Example output:
4   # * main    abc123 [origin/main] Latest commit message
5   #   feature def456 [origin/feature: ahead 2] Some work
```

Definition 3.21 (Push with Set Upstream)

We consider two cases, where we do not see the remote refs at all.
   1. Say that you have a local branch `main` with some committed changes, and the remote branch does not exist at all. Your tree will look something like this

   ```
   1   Remote:
   2   Local : A <- B <- C <- (main) D
   ```

   2. Say that you have a local branch `main` with some committed changes, and the remote branch does exist but the upstream is not set.

   ```
   1   Remote: A <- (main) B
   2   Local : A <- B <- C <- (main) D
   ```

We can synchronize our local commits with the remote repo by **pushing** them. As we push, we also

set the upstream with the `-u` or `-set-upstream` flag.

```
1  git push -u <remote> <branch>
2
3  # for example, if we want to push our current checked out branch to origin/main
4  git push -u origin main
```

This will set your remote refs, and in both cases we will have

```
1  Remote: A <- B <- C <- (main) D
2  Local : A <- B <- C <- (main, origin/main) D
```

Definition 3.22 (Push)

If your remote ref is already set and you have some further commits,

```
1  Remote: A <- B <- C <- (main) D
2  Local : A <- B <- C <- (origin/main) D <- E <- (main) F
```

then you can just push your changes with `git push`, which will push the commits up to HEAD and update the remote ref to HEAD.

```
1  Remote: A <- B <- C <- D <- E <- (main) F
2  Local : A <- B <- C <- D <- E <- (main, origin/main) F
```

In all of these scenarios, we have only seen cases when `origin/main` and `main` point to the same commit. However, remember that the remote ref is a *local* pointer to the remote repo's head, and so it is only updated every time your local repo interacts with the remote repo. Therefore, they can be different.

Example 3.4 (Remote Ref is Different From True Remote Head)

Say that you are `Local1` and a friend is `Local2`. We first start off with this.

```
1  True Remote: A <- B <- C <- (main) D
2  Remote1    : A <- B <- C <- (main) D
3  Local1     : A <- B <- C <- (main, origin/main) D
4  Remote2    : A <- B <- C <- (main) D
5  Local2     : A <- B <- C <- (main, origin/main) D
```

Your friend makes commits `E` and `F` and pushes them while you are working on commit `G`.

```
1  True Remote: A <- B <- C <- D <- E <- (main) F
2  Remote1    : A <- B <- C <- (main) D
3  Local1     : A <- B <- C <- (origin/main) D <- (main) G
4  Remote2    : A <- B <- C <- D <- E <- (main, origin/main) F
5  Local2     : A <- B <- C <- D <- E <- (main, origin/main) F
```

In this case, your friend's push will update the head of the remote main branch to `F` and update their remote ref to `F` as well. But you have worked only locally and have no interacted with the remote repo for a while, so your remote ref is still `D`. When you therefore try to push your changes, git will complain to you that your future ref and the remote head does not match. Since this creates divergent histories which splits from `D`, it will not let you push.

The scenario above is clearly a problem, and it stems from the fact that we have a remote ref. Why need remote refs at all if they seem redundant and at the same time they might introduce this problem? The answer is convenience. Imagine that there were no remote refs. This means that every time someone pushed to the remote main I would need to be notified that there were changes to main. However, I may not have access to the remote repo while I am working on my code, so I may still have the same problem when I am able to connect. The extra remote ref pointer allows for quick checking to see if my local branch matches its upstream. It turns out that if the upstream changed, git does indeed warn you about it as soon as possible.

Definition 3.23 (Fetch)

Say that for a branch, it looks like this, and your local remote is not updated with the true remote's latest commits.

```
1  True Remote: A <- B <- C <- D <- E <- (main) F
2  Remote1    : A <- B <- C <- (main) D
3  Local1     : A <- B <- C <- (origin/main, main) D
```

To update your remote ref to match the current upstream head, we can **fetch** it to get the following.

```
1  True Remote: A <- B <- C <- D <- E <- (main) F
2  Remote1    : A <- B <- C <- D <- E <- (main) F
3  Local1     : A <- B <- C <- (main) D <- E <- (origin/main) F
```

It comes in three commands.

```
1  git fetch <remote-url> <branch> # fetches from specific remote branch
2  git fetch <remote-url>          # fetches from all branches in a remote repo
3  git fetch                       # fetches from all branches of all remote repos
```

Definition 3.24 (Fast-Forward)

In the case above, note that even though the commits are synchronized to your local remote, the changes aren't actually reflected in your current code since HEAD has not moved. If you want head to also move to the most recent remote ref, then you can do a **fast-forward merge**.

```
1  True Remote: A <- B <- C <- D <- E <- (main) F
2  Remote1    : A <- B <- C <- D <- E <- (main) F
3  Local1     : A <- B <- C <- D <- E <- (origin/main, main) F
```

You can do this in git by checking out to the local branch you are on. After fetching, you should have the remote ref that you want to update your HEAD to, which you put in as an argument.

```
1  git merge --ff-only <remote-url/branch>
2  # e.g.
3  git merge -ff-only origin/main
```

Definition 3.25 (Pull)

The combination of doing a fetch to get the latest commits and update the remote ref, and then doing a (fast-forward) merge to update your HEAD pointer to the remote ref, is so common that we refer to doing these two things sequentially as a **pull**. Therefore, the two commands are the same.

```
1   git pull <remote-url> <branch>
2
3   git fetch <remote-url> <branch>
4   git merge <remote-url/branch>
```

```
1   git pull origin main
2
3   git fetch origin main
4   git merge origin/main
```

Definition 3.26 (Clone)

Finally, you can take an existing remote repository and **clone** it, which creates a complete copy of a
remote repository. It does the following.
   1. Does a `git init` to set up git.
   2. Does `git remote add origin <remote-url>` to connect to the remote repo and set the `origin`
      alias to it.
   3. Does `git fetch origin` to fetch all branches of the remote repo, setting up the local remote
      branches and their corresponding remote refs.
   4. Does `git checkout -b main origin/main` which creates the local branch **main**, checks out to
      it, and sets `origin/main` as its upstream.

3.6   Reflog

Definition 3.27 (Reflog)

Git's **reflog** is a super-history that records every change to your branch tips. It's essentially an undo
tree for all git actions (sort of like a meta undo tree). Therefore, if you ever screwed something up, git
allows you to undo your actions by undoing the entries in the reflog. It contains the following types of
actions.
   1. Branch Operations
        • checkout: switching branches
        • branch: creating/deleting branches
        • merge: merging branches
        • rebase: rebasing branches
        • pull: pulling from remote
   2. Commit Modifications
        • commit: new commits
        • reset: moving HEAD
        • revert: reverting commits
        • amend: amending commits
        • cherry-pick: copying commits
   3. Stash Operations
        • stash: stashing changes
        • stash apply: applying stash
        • stash pop: popping stash
   4. History Rewrites
        • rebase -i: interactive rebase
        • filter-branch: rewriting history
   5. Reference Updates
        • HEAD@n: moving HEAD pointer
        • refs/heads/*: branch pointer updates
        • refs/remotes/*: remote branch updates
        • refs/stash: stash reference updates
   6. Remote Operations

- clone: cloning repository
- fetch: fetching from remote
- push: pushing to remote
- remote add: adding remote

You can access it using the following command.

```
$ git reflog
ab12345 HEAD@{0}: reset: moving to HEAD~1    # Most recent action
bc23456 HEAD@{1}: commit: Add feature X
cd34567 HEAD@{2}: rebase: onto main          # Rebase happened here
ef45678 HEAD@{3}: commit: Initial commit
```

3.7   Pull Requests and Forking

4   Package Management with Pip and Conda

Package management is quite a broad term, but for applications I will talk about them in the context of using Python, JavaScript, and C++.

4.1   Pip

Let's start off with a bit of history of Python, which was launched in 1991. 9 years later, the *Python Distribution Utilities (distutils)* module was first added to the Python 1.6.1 standard library (and a month later, in Python 2.0), with the goal of simplifying the process of installing third-party Python packages. However, distutils only provided the tools for packaging Python code, but Python still lacked a centralized catalogue for packages on the internet. As a result, PEP 241 proposed to standardize metadata for packages, and in 2003, the *Python Package Index (PyPI)* was finally launched. As of May 2024, PyPI has over 500,000 packages.[2] Each package is in the form of source archives, called *wheels*, that contain binary modules from a compiled language.

Naturally, there is a need for a package manager, and *easy install* was one of the first ones. After its deprecation in 2004, a software engineer named Ian Bicking introduced *pyinstall*, which was quickly renamed to *pip*.[3] He also created a virtual environment manager, called *virtualenv*, or *venv*.

Example 4.1 (Managing VirtualEnvs)

Here are some useful commands.

```
# Unix Commands
> python -m venv myVenvName/        # create the venv
> source myVenvName/bin/activate    # activate it
> pip freeze > requirements.txt     # export venv into txt file
```

Example 4.2 (Manage Packages Inside Venvs)

Here are some useful commands.

```
> pip list                  # list all packages
> pip install xxx           # install package
> pip install xxx==0.0.1    # install version of package
> pip uninstall xxx         # uninstall package
```

Some notes:
1. Running `pip install packagename` will look at PyPI, find the relevant package, and install one of the precompiled wheels for the operating system/python version that you are using.
2. `pip uninstall` does not uninstall depedencies! There is no built-in support for this, which is a pity. The best way to do this is to pip freeze and look at the differences in the packages.

4.2   Conda

While pip was great for managing Python packages, the main problem was that they were all focused around python, neglecting non-Python library depedencies, such as HDF5, MKL, LLVM, etc. Therefore, they do not install files into Python's site-packages directory. Therefore, *conda* was released to do more than what pip does: handle dependencies *outside* of Python packages as well as Python packages themselves. To reiterate, pip is for Python packages only, while conda is language-agnostic and can install packages in R or C (though it is mainly focused on Python).

---

[2]Really anybody can upload their own, so many packages may contain malware.
[3]From several suggestions the creator received on his blog post, and it is a recursive acryonym for "pip installs packages."

Now that we've gotten this clear, let's talk about *Anaconda*. In 2012, the company Anaconda Inc. was founded and created the *Anaconda* and *Miniconda* distributions mainly focused on data science and AI project for Python and R. You can think of them having the two main components.

1. Access to the *Anaconda Public Repository*, which consists of about 8000 packages (similar to PyPI).

2. A package manager called *conda*, used to install/uninstall/modify these packages in virtual environments.

So the APR/conda is analogous to PyPI/pip. Furthermore, when you install Anaconda, a collection of about 300 essential packages (e.g. numpy, scipy, pandas) come pre-installed. This allows beginners to set up environments quickly with these essential packages but can come with a lot of bloat. There is also some GUI tools that are installed but are not really essential. Miniconda does not pre-install anything, so every new environment is completely empty.

Example 4.3 (Manage Conda Envs)

```
1  > conda env list                      # list all envionments
2  > conda env create -n envname         # create new conda env with name
3  > conda env create -n envname python=3.9   # create new conda env with specific python
       version
4  > conda env remove -n envname         # remove conda env
```

Example 4.4 (Exporting and Creating Environments)

The commands are pretty simple.

```
1  > conda env export > environment.yaml     # export conda environment to yaml
2  > conda env create -f environment.yaml         # make conda env from yaml file
```

However, these files are not portable across operating systems. Therefore, I would generally recommend to run:

```
1  > conda env export --no-builds > environment.yaml
```

This removes the build hashes (which are OS specific), while still keeping the versions.

```
1  - blosc=1.21.3=h6a678d5_0
2  - brotli=1.0.9=h5eee18b_9
```
```
1  - blosc=1.21.3
2  - brotli=1.0.9
```

The problem with this is that it still keeps OS and architecture specific packages. You could add the `-from-history` flag, which only exports the packages that you've explicitly added, but this has two disadvantages:
1. The package versions are lost.
2. The channels are lost.
This defeats the purpose of conda, and it hasn't been resolved. So the best way is to just use the `-no-builds` flag and manually remove whatever packages are not compatible.

Unlike PyPI, the Anaconda repository is divided into *channels*, which are specific links that contain some subfamily of packages. The two most popular ones to know are:

1. `default`: The default channel that is always there for the essentials.

2. `conda-forge`: A free open-source channel containing about 30,000 packages (as of May 2025). Anybody can contribute to this channel.

Example 4.5 (Manage Conda Channels)

There are global commands that affect all conda environments. This can also be changed in your
`.condarc` file, where the channels are listed from highest priority (top) to lowest (bottom).

```
1  > conda config --add channels some-channel        # add a channel permanently to ALL envs
2  > conda config --remove channels some-channel      # remove channel only to current env
```

The following commands are for env-specific settings.

```
1  > conda config --show channels                     # show channels for current env
2  > conda config --env --add channels some-channel    # add channel only to current env
3  > conda config --env --remove channels some-channel  # remove channel only to current
       env
```

Example 4.6 (Manage Packages Inside Conda Envs)

Now that we know about channels, we can talk about installing packages.

```
1  > conda install packagename                    # install package
2  > conda install packagename=0.0.1              # install specific version of package
3  > conda install -c channel packagename         # install package from channel
4  > conda uninstall packagename                  # uninstall package with dependencies
5  > conda uninstall packagename --force          # uninstall package only without
       dependencies
6  > conda uninstall --all --keep-env             # uninstall all packages in env
```

Note that:
1. Conda uses one = sign rather than pip, which uses ==.
2. Conda actually supports both `uninstall` and `remove` keywords, unlike pip which only supports
   `uninstall`.
3. `conda remove` will remove all dependencies that are not used by other packages, which is nice.

4.3 Using Pip with Conda

Now we go to the question that I have asked myself countless times, but have never took the time to study
it until now. What is the difference between `pip install` and `conda install`? How should I use them
together? To determine this, let's compare their behavior.

Example 4.7 (Fresh Environment)

Note that there is always `pip` installed in a venv while nothing is installed in a conda env. Since the
`conda list` is quite verbose, I will exclude the `Build` and `Channel` colummns from now on.

```
1  > pip list
2  Package Version
3  ------- -------
4  pip     25.0.1
```

```
1  > conda list
2  # packages in environment at /opt/miniconda3/envs/test:
3  #
4  # Name                    Version          Build  Channel
```

Example 4.8 (Dependency Installation when Installing a Package)

Now let's install a single package numpy==2.1.0. We can see that the pip list is very minimal and only lists Python-related dependencies, while conda list contains a bunch of non-Python dependencies (a total of 24). Note that pip was automatically installed as a dependency, so we can also run `pip list` in the conda env and get the same output as the one in the venv.

```
1   > pip install numpy==2.1.0
2   > pip list
3   Package Version
4   ------- -------
5   numpy   2.1.0
6   pip     25.0.1
7   .
8   .
9   .
10  .
11  .
12  .
```

```
1   > conda install numpy=2.1.0
2   > conda list
3   # packages in environment at /opt/miniconda3/envs/test:
4   #
5   # Name                     Version
6   ...
7   ncurses                    6.5
8   numpy                      2.1.0
9   openssl                    3.4.1
10  pip                        25.0.1
11  python                     3.13.2
12  ...
```

Example 4.9 (Warning)

I have found in a few cases that if you have pip installed in a conda environment, running `pip install` may not run the `pip` binary in the environment. For example, `which pip` may output the global pip binary, which will install to your global environment rather than your virtual one—even if activated. In this case, you want to force python to execute the pip binary.

1. One way to do this is to simply write out the full path of the pip binary in your conda env.
2. A better way is to activate your conda environment, make sure `which python` outputs the correct binary, and run

```
1   python -m pip install ...
```

Example 4.10 (Uninstalling Package)

Say that we have `pandas` installed and take a look at the list.

```
1   > pip install pandas
2   > pip list
3   Package          Version
4   ---------------  ----------
5   numpy            2.2.4
6   pandas           2.2.3
7   pip              25.0.1
8   python-dateutil  2.9.0.post0
9   pytz             2025.2
10  six              1.17.0
11  tzdata           2025.2
```

```
1   > conda install pandas
2   > conda list
3   # packages in environment at /opt/miniconda3/envs/test:
4   #
5   # Name                     Version
6   ...
7   numpy                      2.2.4
8   openssl                    3.4.1
9   pandas                     2.2.3
10  pip                        25.0.1
11  ...
```

Now if we uninstall it, we can see that conda removes dependencies while pip doesn't.

```
1  > pip uninstall pandas
2  > pip list
3  Package           Version
4  --------------- -----------
5  numpy             2.2.4
6  pip               25.0.1
7  python-dateutil 2.9.0.post0
8  pytz              2025.2
9  six               1.17.0
10 tzdata            2025.2
```

```
1  > conda uninstall pandas
2  > conda list
3  # packages in environment at /opt/miniconda3/envs/test:
4  #
5  # Name                    Version
6  ca-certificates           2025.1.31
7  openssl                   3.4.1
8  .
9  .
10 .
```

Example 4.11 (Dependency Updating)

Now say that we have `numpy==1.26.4` and `scipy==1.12.0` installed in our venv and conda env.

```
1  Package Version
2  ------- -------
3  numpy   1.26.4
4  pip     23.2.1
5  scipy   1.12.0
6  .
7  .
8  .
9  .
10 .
11 .
12 .
13 .
14 .
```

```
1  # packages in environment at /opt/miniconda3/envs/test:
2  #
3  # Name                    Version
4  numpy                     1.26.4
5  openssl                   3.4.1
6  pip                       25.0.1
7  python                    3.12.9
8  python_abi                3.12
9  readline                  8.2
10 scipy                     1.12.0
11 setuptools                78.1.0
12 tk                        8.6.13
13 tzdata                    2025b
14 wheel                     0.45.1
```

We would like to upgrade numpy to 2.2.0, but this will break the dependency for scipy. Both package managers report this, and pip gives a more readable message. However, note that conda does not install `numpy=2.2.0`, while pip *does* and reports that this can break things. So even though it checks for dependencies, it does *not* automatically update them!

```
1  Package Version
2  ------- -------
3  numpy   2.2.0
4  pip     23.2.1
5  scipy   1.12.0
6  .
```

```
1  # packages in environment at /opt/miniconda3/envs/test:
2  # Name                    Version
3  numpy                     1.26.4
4  pip                       25.0.1
5  python                    3.12.9
6  scipy                     1.12.0
```

As we have seen there are two deal-breakers for pip, which is that it does not clean up dependencies upon installation and that it updates packages that may break dependencies. This is really because pip is a package manager, but it is *not* a dependency manager. So personally, I only do pip install when it is absolutely necessary, i.e. I need a package that is only available on PyPI and not on any Anaconda channels.

Theorem 4.1 (Best Practices for using Conda and Pip)

Here are my personal best practices.
1. Always use conda environments.
    (a) Conda environments completely replace virtualenvs. There is nothing you can do in vir-

tualenvs that you cannot do in conda envs.
  (b) Venvs only work with pip, while conda envs allow you to have access to conda, which can be used to download pip.
  (c) You cannot easily switch Python versions in an environment in venv, since you must have the binary installed on your computer. However for conda, it is as easy as `conda install python=3.x`.
2. Always use `conda install` if possible, and only use `pip install` if you need a package only on PyPI. This is for the following reasons.
  (a) Due to dependency breaking as mentioned above (and elaborated below), pip can be a huge headache to work with.[a]
  (b) Pip usually breaks more often when downloading more outdated packages. [b]
3. Whether you export your environment one way or another will depend on how flexible/rigid you want your working environment to be when you share.
  (a) `conda env export` will keep track of every (including non-Python related) modules, and those imported with pip will be under the `pip` header.
  (b) `pip freeze` will only keep track of Python packages installed and can be cleaner.

---

[a]Though most widely used packages are pretty good at making sure that there are no incompatibilities.
[b]For example, installing `pandas=1.1.4` works on conda but not with pip.

Others note that if you using conda environments, you should always just use `conda install`, and if you ever need to `pip install`, then just use `venv` and `pip install` everything. With venvs, if you ever see a dependency issue, don't try to resolve it: burn the whole environment down and recreate a new one from scratch.

4.4 Mamba

I've been using pip and conda for about 6 years before I found out about *mamba* in a summer internship. The Mamba project began in 2019 as a thin wrapper around conda and has grown considerably by progressively rewriting conda with equivalent new efficient C++ code. It is estimated to be about 10 times faster in creating a large environment from scratch compared to conda. There are essentially no strict disadvantages to using mamba, but due to its newness it is still relatively unpopular. So besides the fact that mamba support is lower, it might be good to try it as a replacement. It also seems that recently (as of May 2025), conda caught up to the speed of mamba, so it also may not be worth switching. I'll have to run some tests for this.

The bigger consideration is that for many users (such as companies with 200+ employees), Anaconda Inc. starting from 2020 required paid licensing for commercial use, including any use of the `defaults` channels (though `conda-forge` channel remains free).

5   Linux Desktop

The following set of notes describes the everyday use of a Linux operating system. I refer to it for mainly my personal desktop, but it is also useful for working in computing clusters. Some of the commands are specific to the Arch Linux distribution (since that is what I work with), but I occasionally include those from Ubuntu and Red Hat, since I run into these distributions often in servers.

I try to organize this in a way so that one who wishes to get started in Linux can go through these notes chronologically. For now, we will assume that you have a Linux distribution installed. There are many resources beyond this book that helps you do that.

You can always try out Ubuntu (or any other distribution) through a **virtual machine**, which is a software emulation of a physical computer system. It allows you to run multiple operating systems or instances of an operating system on a single physical machine. Each virtual machine operates independently and has its own virtual hardware, including virtual CPU, memory, storage, and network interfaces. Virtual machines are created and managed by virtualization software called **hypervisors**. The hypervisor abstracts the underlying physical hardware and allows multiple virtual machines to share the same resources while isolating them from one another. This enables efficient utilization of hardware resources and provides flexibility in deploying and managing various operating systems and software applications. VMs generally have the advantage of being completely isolated from the main computer, so if anything wrong happens in the VM, it's fine. They can be used in research environments that are beta-testing unstable packages or for white-hacking practices. One example of a hypervisor is Oracle's **VirtualBox**, which is free to download. It should look like this when you open it for the first time.
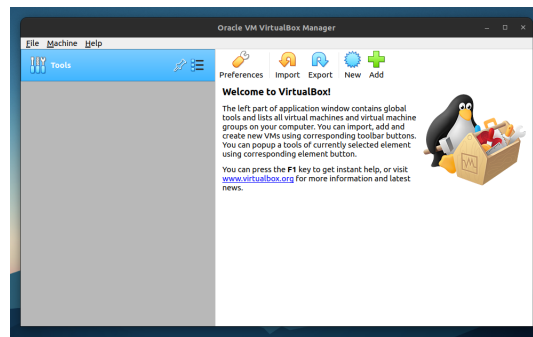

Now in order to create a VM with its own OS, you need to have the appropriate **ISO file**, which is an exact copy of an entire optical disk such as a CD, DVD, or Blu-ray archived into a single file. The essentially stores the entire software needed to operate the OS. Therefore, you should download the proper ISO file from the internet (usually a couple GBs).

1. Ubunutu ISO files

2. Windows 10 ISO files

3. Apple does not allow distribution of its ISO files, so you will need to download from unofficial sources, which may be unsafe.

Once you have this ISO file, you can reuse it to create as many VMs as you want of that OS. Now follow these instructions: Click the new button and select where the virtual machine data will be stored, along with its OS. You can set the RAM, but don't make it more than half of your host computer since it will hog up too much RAM. Choose "Create a virtual hard disk now". Choose "VDI (VirtualBox Disk Image)". Dynamically allocated just means that the virtual disk size will adaptively grow as your storage gets full. Set the disk size to be at least 20GB.

After you created this, go to the VM settings (this is where you can edit your CPU cores, RAM cap, etc.). To add the ISO file, click on the "Empty" tab right under the "Controller:IDE", then the CD icon to the right, and "choose a disk file". You should now choose the ISO file. Then go tweak other settings, and set the

display:video memory to the max (128MB). Now you should be able to go through the installation wizard when you turn the VM on. Refer to the instructions for each OS.
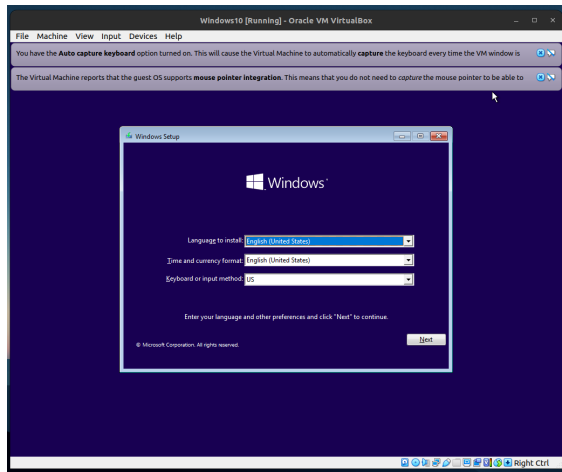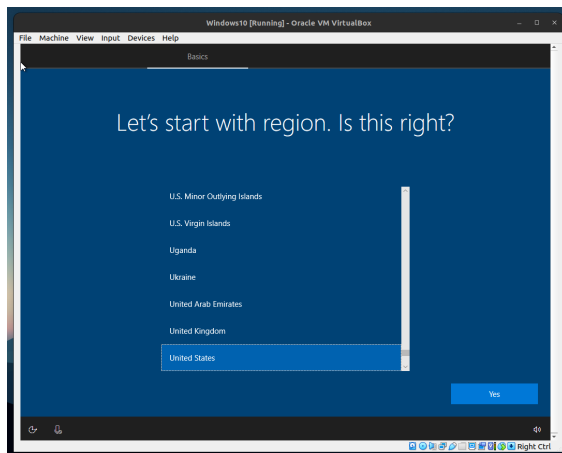

(a) Windows 10 Set Up
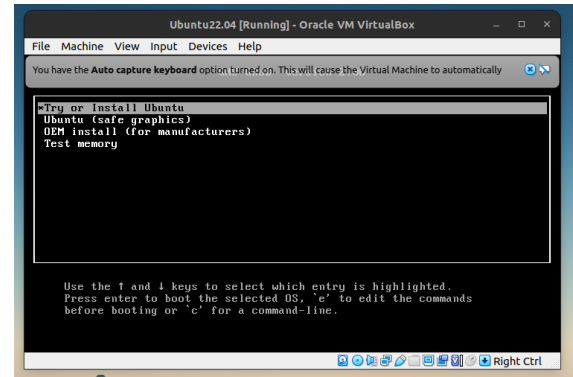
(b) Ubuntu 22.04 Set Up

Figure 2: What you should get once you open up the VM after adding ISO files.

1. For Windows: Say I don't have a product key. Click Windows 10 Home. Accept terms. Select the custom installation. Click the drive and click new, making the parititon at least 10534MB, and click apply. Next. Wait for the system to load.

2. For Ubuntu, you should get a GRUB view. Select "Try or install Ubunutu".

You should now see one of these two screens.

(a) Windows 10 Set Up
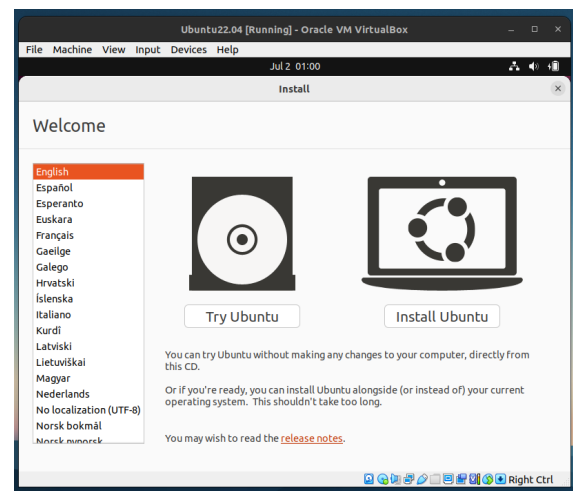
(b) Ubuntu 22.04 Set Up

Figure 3: What you should get once you open up the VM after initial configuration and log in.

1. For Windows, select your region. Select the keyboard layout. Sign in or create a Microsoft account. Choose privacy terms. Skip whatever.

2. For Ubuntu: Select Install Ubuntu with English. Set the keyboard layout. The normal installation may take a while, so I would select minimal depending on what you need. If you are short on time,

you can uncheck the download updates while installing since you can always do that after you install. Click Erase disk and install Ubunutu. Choose region and add information.

Finally, you should see your desktop.
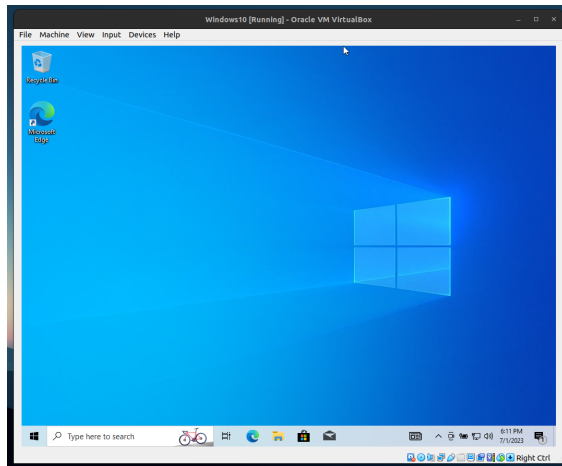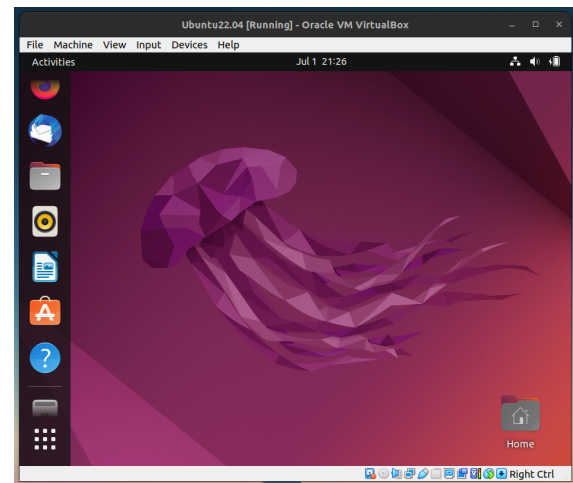
(a) Windows 10 Set Up   (b) Ubuntu 22.04 Set Up

Figure 4: What you should see once everything is set up.

For my personal use, the packages below are ones that I end up installing every time I create a new VM to work in during research.

```
1   sudo apt update
2   sudo apt install snapd
3   sudo snap install --classic code
4   sudo snap install slack
5   sudo apt install git
6   sudo snap install spotify
7   sudo apt install htop
8   wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
9   sudo dpkg -i google-chrome-stable_current_amd64.deb
10  sudo apt install virtualbox
```

Once you are ready to use Linux consistently, it is optimal to **dual boot** it, which means that you have one computer that is divided into two: one for each operating system. Then you need to partition your drive and allocate it to your secondary OS. There are plenty of guides and tutorials online on how to do this.

There may be a point where you may need to resize your drive partitions as you need more or less space in one of your OS. This is when we need to do **partition resizing**. To do this, we need an empty thumb drive with at least 8GB of space in it (everything in here will be deleted). Then in your Ubuntu, install balenaEtcher and an Ubuntu (any version) ISO file. Mount the ISO file into your USB drive using balenaEtcher, following the steps in this video to eventually get into Gparted. Another popular guide uses Rufus in the Windows system, but I have found that this does not work for me.

5.1 Systemd

A **process** is really any program that is running on your computer. A **daemon** is a background process that runs continuously, performing specific tasks even when no user is logged in.

Once the kernel has been loaded and completed its initialization process, it creates a collection of *spontaneous* (as in the kernel starts them automatically) processes in user space. They're really part of the kernel

implementation and don't necessarily correspond to programs in the filesystem. They're not configurable and they don't require administrative attention. These processes can be monitored with the commands `ps`, `top`, or `htop`.

The most important process is the init process, with a system PID of 1 and with special privileges. It is used to get the system running and for starting other processes.

1. Setting the name of the computer

2. Setting the time sone

3. Checking disks with `fsck`

4. Mounting filesystems

5. Removing old files from the `/tmp` directory

6. Configuring network interfaces

7. Configuring packet filter

8. Starting up other daemons and network services, along with killing zombie processes or parenting orphaned processes.

There are three flavors of system management processes in widespread use:

1. Historically, SysVinit was a series of plaintext files that ran as scripts to start processes, but due to some problems, Linux now uses systemd.

2. An init variant that derives from the BSD UNIX, used on most BSD-based systems.

3. A more recent contender called **systemd** which aims to cover the init processes and much more. This significant increase in control causes some controversy.

4. Other flavors include Apple MacOS's **launchd** before it adopted systemd. Ubuntu also used **Upstart** before migrating to systemd.

Systemd is essentially a collection of smaller programs, services, and libraries such as systemctl, journalctl, init, process management, network management, login management, logs, etc. Some processes may depend on other processes, and with hundreds of them, it's very hard to do manually, which is why systemd does it all for you. A post on the systemd blog notes that a full build of the project generates 69 different binaries (subject to change).

Definition 5.1

A **unit** is anything that is managed by systemd. It can be "a service, a socket, a device, a mount point, an automount point, a swap file or partition, a startup carget, a watched filesystem path, a time controlled and supervised by systemd, a resource management slice, or a group of externally created processes." Within systemd, the behavior of each unit is defined and configured by a **unit file**. Within systemd, the behavior of each unit is defined and configured by a **unit file**.

The files are all over the place:

1. `/lib/systemd/system` contains standard systemd unit files
2. `/usr/lib/systemd/system` are from locally installed packages, e.g. if I installed a pacman package that contained unit files, then those would go here.
3. `/etc/systemd/system` is where you put your custom files. etc also has the highest priority, so it overwrites the other files.
4. `/run/systemd/system` is a scratch area for transient units.

By convention, unit files are named with a suffix that varies according to the type of unit being configured. For example, service units have a `.service` suffix and timers user `.timer`. Within the unit file, some sections e.g. (`[Unit]`) apply generically to all kinds of units, but others (e.g. `[Service]`) can appear only in the context of a particular unit type.

Example 5.1 (Service Unit File)

If we go into one of these unit files, which have the prefix `.service`, they are usually formatted as such:

```
1   # comments are just the same as in bash Scripts
2   # the headers are important!
3
4   [Unit]          #
5   Description=Description of the unit file
6   Documentation=man:something
7   After=network.target
8
9   [Service]
10  Type=forking  # tells that the process may exit and is not permanent
11  PIDFile=       #
12  ExecStartPre= # scripts to run before you start
13  ExecStart=    # scripts to run when starting
14  ExecReload=   # script to run when you try to reload the process
15  ExecStop=     # script to run to stop the process
16
17  [Install]   # Tells at what point should this be running
18  WantedBy=multi-user.target
```

5.1.1   systemctl: Managing systemd

systemctl is an all-purpose command for investigating the status of systemd and making changes to its configuration. Running `systemctl` without any arguments invokes the default `list-units` subcommand, which shows all loaded and activive services, sockets, targets, mounts, and devices. To show only services, use `-type=service`.

The two main commands that you will use to interact with systemd is `systemctl` and `journalctl`.

1. `systemctl status unit` checks the status, ouputting the description, whether it's enabled/disabled, and whether it's active/inactive.

2. `systemctl enable unit` enables it, which means that it will start when booting the computer. It does this by creating a symlink to the unit file. This is different from start.

3. `systemctl disable unit` disables it.

4. `systemctl start unit` starts it now and runs it immediately.

5. `systemctl stop unit` makes it inactive.

6. `systemctl reload` will run whatever is in the `ExecReload` in the unit file.

7. `systemctl restart` runs ExecStop and then ExecStart.

8. `systemctl kill unit` kills the process.

Some of the statuses that you may see are inactive (deactivated, exited), active (activating, running), failed, static (not started, frozen by systemd), bad (broken, probably due to bad unit files), masked (ignored by systemd), indirect (disabled, but another unit file references it so it could be activated).

To troubleshoot, you should run `systemctl -failed` to see if there are any failed processes, which can be a problem, and then you can use `journalctl -since=today` to view your systemd logs. This log is important for diagnosing fundamental problems with your system. To view only entries logged at the error level or above, you can set the priorities with `-p err -b`.

5.1.2 Targets

5.1.3 Systemd Logging

The **journald** daemon allows you to capture log messages produced by the kernel and services. These system messages are stored in the `/run` directory, but we can access them directly with the `journalctl` command.

Example 5.2

You can configure **journald** to retain messages from prior boots. To do this, edit the following file and configure the `Storage` attribute:

```
#/etc/systemd/journald.conf
  [Journal]
  Storage=persistent
```

Then, you can obtain a list of prior boots with `journalctl -list-boots` and you can access messages from a prior boot by referring to its index or by naming its long-form ID: `journalctl -b -1`.

5.2 Directory Structure

It should be clear that the   stands for your user home directory, while `/` stands for the root directory.

```
(base) mbahng@xps15:~\$ pwd
/home/mbahng
(base) mbahng@xps15:~\$ cd /
(base) mbahng@xps15:/\$ pwd
/
```

Let us now take a look at the contents of the root directory:

```
(base) mbahng@xps15:~\$ ls /
bin    dev   lib   libx32     mnt   root  snap  timeshift  var
boot   etc   lib32 lost+found opt   run   srv   tmp
cdrom  home  lib64 media      proc  sbin  sys   usr
```

You can see that the root home directory is in here, as opposed to user home directories in the `/home` folder.

1. `root`: This contains all the files for when you need to boot. You shouldn't mess with this.

2. `etc`: This is where you system wide configuration for applications is stored (unlike local configuration files for one user, which is stored in your home directory). It is often a target for backups.

3. `media`, `mnt`: Used for mounting external storage systems and even internal storage systems.

4. `opt`: A place where you can install whatever you want. Quite flexible.

5.2.1 Users and Permission

You should first check which users are on your system. Most people just check their home directory using

```
(base) mbahng@xps15:~\$ ls -l /home
total 8
drwxr-xr-x  3 root    root    4096 Jan 17 23:57 linuxbrew
drwxr-xr-x 44 mbahng  mbahng  4096 Jul  2 13:27 mbahng
```

But this is not accurate. Rather, we should check the contents of the `/etc/passwd` file, which has a list of users in our computer (1 per line). The purpose is to contain a listing of and the options that are associated with your user accounts on your server.

```
1   (base) mbahng@xps15:~\$ cat /etc/passwd
2   root:x:0:0:root:/root:/bin/bash
3   daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
4   bin:x:2:2:bin:/bin:/usr/sbin/nologin
5   sys:x:3:3:sys:/dev:/usr/sbin/nologin
6   sync:x:4:65534:sync:/bin:/bin/sync
7   games:x:5:60:games:/usr/games:/usr/sbin/nologin
8   man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
9   lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
10  mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
11  news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
12  uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
13  proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
14  www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
15  backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
16  list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
17  irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
18  gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
19  nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
20  ...
```

Let us just examine my user.

```
1   (base) mbahng@xps15:~\$ cat /etc/passwd | grep mbahng
2   mbahng:x:1000:1000:mbahng,,,:/home/mbahng:/bin/bash
```

Going from left to right, mbahng is my user, the x stands for a hashed password that cannot be shown, the 1000 is the user id (UID), the 1000 is the group id (GID), mbahng is the user information field (optional), next `/home/mbahng` is the user's home directory, and finally `/bin/bash` is the shell designated for the user. When you create a user id when first installing Ubuntu, this will almost always have uid of 1000. On most linux distributions, the user accounts that will be used by humans are given uids of 1000 and above. Note that in Ubunutu 22.04, a home directory is not created automatically (this differs based on distribution) when we create a new user. So note the following commands. To add a user called batman, we have

```
1   (base) mbahng@xps15:~\$ sudo useradd batman          # just add user
2   (base) mbahng@xps15:~\$ sudo useradd -m batman       # add user with home dir
3   (base) mbahng@xps15:~\$ cat /etc/passwd | grep batman
4   batman:x:1001:1001::/home/batman:/bin/sh
```

and it gives a new uid that is the next available one from 1000, i.e. 1001. To delete the user, just do

```
1   (base) mbahng@xps15:~\$ sudo userdel batman          # delete user
2   (base) mbahng@xps15:~\$ sudo userdel -r batman       # delete user w/ home dir
```

Now let's talk about changing passwords. If you want to change your own password, you can just type `passwd` and go through the steps. To set another user's password, you need to be in root mode and type

```
1   (base) mbahng@xps15:~\$ sudo passwd batman           # set password for batman
```

Note that we have a hashed version of the user's password in the `/etc/passwd` file. We can actually see the full hashed versions by going into `/etc/shadow`.

Running `ls -l` command lists all files and directories in your current working directory, along with their permissions.

```
1   -rw-rw-r--  1 mbahng mbahng 4336730777 Sep 29  2022   cuda_11.8.0_520.61.05_linux.run
2   drwxr-xr-x  9 mbahng mbahng       4096 Jul  1 23:33   Desktop
3   drwxr-xr-x  8 mbahng mbahng       4096 Jul  1 15:08   Documents
4   drwxr-xr-x  6 mbahng mbahng      12288 Jul  1 22:36   Downloads
5   drwxr-xr-x  4 mbahng mbahng       4096 Jun 29 19:43   Games
6   drwxr-xr-x  6 mbahng mbahng       4096 Feb 22 17:27   Jts
7   drwxrwxr-x  5 mbahng mbahng       4096 Jun 28 19:39   KakaoTalk
8   drwxr-xr-x 16 mbahng mbahng       4096 Jun  2 21:13   miniconda3
9   drwxrwxr-x  4 mbahng mbahng       4096 Jun 22 13:12   nltk_data
```

The first columm is a string of 10 characters representing the permissions. They are divided into 4 sections:

```
1   d   rwx   r-x   r-x
```

The first letter can be a d, l, or -, meaning directory, link, or file, respectively. The next three groups, representing the permissions of the user (third columm), group (fourth), and everyone else, have the same format. It is rwx, which stands for read, write, execute.

1. Read: Means to read a file or read a directory.

2. Write: Means to edit a file or modify the contents of a directory.

3. Execute: Means to run the file as an executable or go `cd` into the directory.

A dash in place of any one of them means that whatever entity does not have the permissions. However, we can set the permissions using the `chmod` command. If we have a file named `testfile.txt` in our current directory, we can add or revoke permissions with

```
1   chmod +r testfile.txt    // assign read permissions to all users
2   chmod +w testfile.txt    // assign write permissions to all users
3   chmod +x testfile.txt    // assign execute permissions to all users
4
5   chmod g+rw testfile.txt   // assign read and write to group
6   chmod u-r testfile.txt   // revoke read to user
7   chmod o+x testfile.txt    // assign execute to other users
```

Writing all these can be tedious, so what we can do is take advantage of the numerical encodings of the permissions. Note that $r = 4, w = 2, x = 1$, and so any number between 0 and 7 can encode the three bits (through the coefficients of the binary expansion). Therefore, if we wanted every permission for all users, we can write

```
1   chmod 770 testfile.txt
```

where the first 7 stands for `rwx`, the next 7 stands for `rwx`, and the final 0 stands for `--`. To change the permissions for everything inside a directory (e.g. say you want to make all downloads only readable and writable by you), then you can type

```
1   chmod 600 ~/Downloads/*
```

If you have multiple users in your computer (type `ls /home`), then you may want to give ownership of a directory or folder to another user.

```
1   (base) mbahng@xps15: ls /home
```

```
2  batman mbahng
```

To change permissions of a file/directory to another user and group, we can use the `chown` command (with sudo)

```
1  sudo chown -R batman:batman Downloads/
```

5.3   Display Servers

When you boot up your computer, you are greeted with a graphical user interface (GUI) that allows you to interact with your computer. This is the job of the display server, which is a program that provides graphical display capabilities for the operating system.

Definition 5.2 (Display Server)

A **display server** is a program that manages the communication between your computer's hardware and graphical software applications. It acts as a bridge for input and output devices; for example, it processes the input from your keyboard and mouse and outputs graphics to the monitor. The display server is responsible for the fundamental task of drawing windows and handling the low-level aspects of input and output, but it doesn't dictate how these windows look or are arranged. For almost every purpose, there are two types of display servers:
1. **X**: The X Window System, which is the older and more established display server.
2. **Wayland**: The newer and more modern display server.

Definition 5.3 (X Window System)

The **X Window System** is a windowing protocol for Unix/Linux OSes, similar to the way that Microsoft Windows or Apple Mac OS X can run different apps in separate windows. **X** defines the protocol for a display server what can render windows on a *display client* (your computer), inside which are running apps.[a]
1. **X11** refers to version 11 of the X protocol, while
2. **Xorg** is an open-source implementation of X.

---
[a]Explanation here: https://www.reddit.com/r/linuxquestions/comments/3uh9n9/what_exactly_is_xxorgx11/

Definition 5.4 (Wayland)

X, made in 1984, has developed a lot of cruft over the years, and Wayland is a modern replacement for X. It is a protocol for a compositor to talk to its clients, as well as a C library implementation of that protocol. The compositor can be a standalone display server running on Linux kernel modesetting and evdev input devices, an X application, or a wayland client itself.

5.4   Package Management

Linux comes in many flavors of distributions. Most beginners look at screenshots of these distributions on the internet and judge them based on their aesthetics (e.g. I like how Kali Linux looks so I'll go with that one). A common feature of all Linux distributions is that they provide the user the power to customize their system however they want, so you can essentially make every linux distribution look like any other. So what are some things you should consider when choosing a distribution?

1. First is the popularity and how well it is supported. This includes the number of people who use the distribution (e.g. the Ubuntu StackExchange is a very large community) and how good the documen-

tation is overall (e.g. the ArchLinux wiki is very well documented).

2. Each linux distribution essentially consists of a kernel and package manager. The architecture, design, and the update scheme of the kernel may be an interest to many linux users.

3. Every distribution has its own native package manager, and the availability of certain necessary packages, the ease of installation, and the updating schemes is also something to consider.

4. The ideals of the respective communities. The community behind each distribution has a certain set of ideals that they lean more towards. For example, the Ubuntu community likes having programs that are right out of the box, with good GUI support and is more beginner-friendly while Arch has more of a minimal and extremely customizable nature to it with its software being much more CLI dependent.

Let's begin with the package managers. Every application on your system (Firefox, Spotify, pdf readers, VSCode, etc.) is a package, and manually downloading and managing each one is impossible to do. Therefore, each distribution has its own native package manager that automatically takes care of downloading, installing, removing, checking dependency requirements of each package. In order to download a package, a package manager should also know where it is downloading *from*. Essentially, a package manager itself can be downloaded with other package managers, so package managers are packages as well.

1. **apt** : The advanced packaging tool is the native manager for Ubuntu distributions.

2. **pacman** : Native package manager for Arch Linux.

3. **yay** : The package manager for software in the **Arch User Repository**.

4. **snap** :

5. **flatpak** :

6. **dpkg** : Package manager for Debian based distributions.

Chances are if you are using one distribution, you would only have to work with a small subset of these package managers. Each package manager has one or more files in the computer that specify a list of **repositories**.

5.5   Wget

wget is a command-line utility used to download files from the internet. It stands for "web get."

5.6   Pacman

For example, the configuration file for pacman is located at `/etc/pacman.conf`. In the options section, I can configure stuff like text color, enabling/disabling parallel downloads, choosing specific packages to ignore upgrading, etc. Then, we can specify the servers that we should download from. In the text below, the server variable defines which server we should look at first, and then the Include variable stores the location of the file `mirrorlist` that defines a list of other servers that we should download from.

The mirrorlist file stores a list of URLs. Each URL is a **mirror**, which is a server that contains a physical replica of all the packages that are available to you via `pacman` (hence the name mirror). You can literally type in the links provided in Figure 6 (replacing `$repo` with `core` and `$arch` with `x86_64`). It contains a tarball of each package ready to be downloaded. Some repos might contain more packages than others, some might have packages that only they supply that others don't, but if you can install the piece of software via your package manager then one of your configured repos is declaring they have it available and therefore should have the file on hand to give to you if asked for it. A list of all available mirrors are available here (this only uses HTTPS, but HTTP mirrors are also available).

The mirrors that you download from should be trustworthy and fast. The speed is mainly related to how close you are to that mirror geographically, so if you are moving to another country you should probably update this mirrorlist for faster download speeds. There is a default mirrorlist file that is generated, but you can download and use the **reflector** package to update it.

```
1   # The following paths are commented out with their default values listed.
2   # If you wish to use different paths, uncomment and update the paths.
3   #RootDir      = /
4   #DBPath       = /var/lib/pacman/
5   #CacheDir     = /var/cache/pacman/pkg/
6   #LogFile      = /var/log/pacman.log
7   #GPGDir       = /etc/pacman.d/gnupg/
8   #HookDir      = /etc/pacman.d/hooks/
9   HoldPkg       = pacman glibc
10  #XferCommand = /usr/bin/curl -L -C - -f -o %o %u
11  #XferCommand = /usr/bin/wget --passive-ftp -c -O %o %u
12  #CleanMethod = KeepInstalled
13  Architecture = auto
14
15  # Pacman won't upgrade packages listed in IgnorePkg and members of IgnoreGroup
16  #IgnorePkg    =
17  #IgnoreGroup =
18
19  #NoUpgrade    =
20  #NoExtract    =
21
22  # Misc options
23  #UseSyslog
24  #Color
25  #NoProgressBar
26  CheckSpace
27  #VerbosePkgLists
28  ParallelDownloads = 5
29  ILoveCandy
```

Figure 5: Subset of contents of the `/etc/pacman.conf` file

```
1   Server = https://archlinux.mailtunnel.eu/$repo/os/$arch
2   Server = https://mirror.cyberbits.eu/archlinux/$repo/os/$arch
3   Server = https://mirror.theo546.fr/archlinux/$repo/os/$arch
4   Server = https://mirror.sunred.org/archlinux/$repo/os/$arch
5   Server = https://mirror.f4st.host/archlinux/$repo/os/$arch
6   Server = https://md.mirrors.hacktegic.com/archlinux/$repo/os/$arch
7   Server = https://mirrors.neusoft.edu.cn/archlinux/$repo/os/$arch
8   Server = https://mirror.moson.org/arch/$repo/os/$arch
9   Server = https://archlinux.thaller.ws/$repo/os/$arch
```

Figure 6: Contents of the `/etc/pacman.d/mirrorlist` file

Here are some common commands:

1. Install a package: `sudo pacman -S pkg1` (`-s` stands for synchronize)

2. Remove a package: `sudo pacman -R pkg`

    - remove dependencies also: `-s` (recursive)

    - also remove configuration files: `-n` (no save)

    - also removes children packages: `-c` (cascade)

3. Update all packages: `sudo pacman -Syu`

    - synchronize: `-S`

    - refresh package databases: `-y` (completely refresh: `-yy`)

    - system upgrade: `-u`

4. List installed packages: `pacman -Q`

    - List detailed info about a package: `pacman -Qi pkg`

    - List all files provided by a package: `pacman -Ql pkg`

    - List all orphaned packages: `pacman -Qdt`

    - List all packages that have updates available: `pacman -Qu`

    - List all explicitly installed packages: `pacman -Qet`

    - Display the dependency tree of a package: `pactree pkg` (from the `pacman-contrib` package)

    - List last 20 installed packages:

      ```
      1   expac --timefmt='%Y-%m-%d %T' '%l\t%n' | sort | tail -n 20
      ```

5. To check size of current packages and dependencies, download `expac` and run `expac -H M '%m t%n' | sort -h`

6. The package cache stored in `/var/cache/pacman/pkg/` keeps old or uninstalled versions of packages automatically. This is helpful since it also keeps older versions of packages in the cache, and you can manually downgrade in case some packages break.

    - We can delete all cached versions of installed and uninstalled packages, except for the most recent 3, by running `paccache -r` (provided by the `pacman-contrib` package).

    - To remove all cached packages not currently installed, run `pacman -Sc`

    - To remove all cached aggressively, run `pacman -Scc`

    - To downgrade, you go into the package cache directory and say you want to see which versions of neovim you have installed. You can `ls` the directory to see the following.

      ```
      1   neovim-0.9.5-1-x86_64.pkg.tar.zst
      2   neovim-0.9.5-1-x86_64.pkg.tar.zst.sig
      3   neovim-0.9.5-2-x86_64.pkg.tar.zst
      4   neovim-0.9.5-2-x86_64.pkg.tar.zst.sig
      ```

      We have an older version of neovim installed, and to roll it back we can use

      ```
      1   pacman -U neovim-0.9.5-1-x86_64.pkg.tar.zst
      ```

The pacman log (`/var/log/pacman.log`) is also useful since it logs all pacman outputs when you do anything with pacman. So if you are looking for the packages that have been installed in the latest `pacman -Syu`, then you can use this to individually see each package that has been upgraded.

5.7 Yay

Yay is used to install from the Arch User repository and must be updated separately. To run this, you can either run `yay -Syu` or you can just run `yay`. Since this is not officially maintained, these packages are more likely to break something. The yay logs are not stored separately can can be accessed in the pacman logs.

5.8 Dpkg and Deb files

Ubuntu is a Linux distribution within the family of Debian-based systems (with Debian, Linux Mint, etc.). File of the `.deb` format is used to distribute and install software packages on these systems. A deb package contains the files for a particular software application or library, along with metadata that describes the package and instructions on how to install or remove it. The package format follows a specific structure and includes files such as control files, data files, and scripts. Therefore, many downloaded packages may come in this format, similar to how a file is zipped before we have to extract it.

Dpkg is the primary package manager for Debian based systems. It installs, builds, removes, configures, and retrieves information for Debian packages of the `.deb` format. Given that we have some file `package.deb` downloaded, the command

```
1  dpkg -i package.deb
```

installs the specified package from the `package.deb` file. Removing it is just (note without the suffix)

```
1  dpkg -r package
```

5.9 Apt

read more here

While dpkg is the native package manager for Debian based systems, apt is just a built-in Ubuntu tool to help install these Debian packages and manage dependencies. To run apt commands, we must have root privilege, so we should always use sudo. When these command are run, you should get a confirmation question asking whether you want to continue, with [Y/n]. The capital letter is the default, so you can either enter in 'y' or just press ENTER.

1. The update command connects to various URLs to download a list of available packages. Periodically, new packages are introduced to Debian and Ubunutu repositories all the time, so this command refreshes the index so that it knows what packages are available and at what versions. It is a good idea to run this before you use apt commands for the day.

   ```
   1  sudo apt update
   ```

2. The upgrade command just updates all packages and their dependencies to their latest versions. However, this does not update packages which require the installation of *additional* packages.

   ```
   1  sudo apt upgrade
   ```

3. The dist upgrade updates packages including those that need installation of new dependencies. So it is a good idea to run upgrade first and then dist-upgrade after.

   ```
   1  sudo apt dist-upgrade
   ```

Installing and removing packages is easy.

1. We can install from the apt repository with

```
1   sudo apt install htop
```

2. We can remove it with

```
1   sudo apt remove htop
```

If you don't know the name of the application or package you want to install, then you can search for a keyword with apt search. Say that you want to install vim but you don't know what the actual package name is called. You can just type

```
1   apt search vim
```

The central location where apt gets its updates from is contained in the `/etc/apt/sources.list` file. Here is a snippet of it in my system.

```
1   # deb cdrom:[Ubuntu 22.04.1 LTS _Jammy Jellyfish_ - Release amd64 (20220809.1)]/
2   jammy main restricted
3
4   # See http://help.ubuntu.com/community/UpgradeNotes for how to upgrade to
5   # newer versions of the distribution.
6   deb http://us.archive.ubuntu.com/ubuntu/ jammy main restricted
7   # deb-src http://kr.archive.ubuntu.com/ubuntu/ jammy main restricted
8
9   ## Major bug fix updates produced after the final release of the
10  ## distribution.
11  deb http://us.archive.ubuntu.com/ubuntu/ jammy-updates main restricted
12  # deb-src http://kr.archive.ubuntu.com/ubuntu/ jammy-updates main restricted
13
14  ## N.B. software from this repository is ENTIRELY UNSUPPORTED by the Ubuntu
15  ## team. Also, please note that software in universe WILL NOT receive any
16  ## review or updates from the Ubuntu security team.
17  deb http://us.archive.ubuntu.com/ubuntu/ jammy universe
18  # deb-src http://kr.archive.ubuntu.com/ubuntu/ jammy universe
19  deb http://us.archive.ubuntu.com/ubuntu/ jammy-updates universe
20  # deb-src http://kr.archive.ubuntu.com/ubuntu/ jammy-updates universe
21  ...
```

5.10   Snap and Flatpak

Other package managers that you may need to use often are snap and flatpak, which can both be installed with

```
1   sudo apt install snap flatpak
```

5.11   Windows Managers and Desktop Environments

These days, the terms window managers (WMs) and Desktop Environments (DEs) are used interchangeably, but they mean slightly different things. A window manager is the display software that determines how the pixels for each window overlaps with other and their movement. This is generally divided into two paradigms with the most familiar being **floating WMs** and the other being **tiling WMs**. Even before I knew about

tiling WMs, I found myself manually tiling windows on floating WMs, so the move to tiling WMs was a no-brainer.

Some DEs and WMs are:

1. GNOME

2. KDE Plasma

3. Qtile

5.12   Shells and Terminals

Beginners may think of the shell and the terminal to be the same thing, but they are different. The **shell** is a command line interpreter, a layer that sits on top of the kernel in which the user can interact with. It is essentially the only API to the kernel where the user can input commands and processes them. The **terminal emulator** is a wrapper program that runs a shell and allows us to access the API. It may be useful to think of the shell as like a programming language and the terminal as a text editor like VSCode.

The three most common shells are the following:

1. **Bash**:

2. **Zsh**:

3. **Fish**:

Some common terminal emulators (most of which comes as a part of the desktop environment) are the following:

1. **Kitty**:

2. **Alacritty**:

3. **Gnome-Terminal**:

5.12.1   Crontab

To schedule jobs, you run `crontab -e`, which will give you a text file for which you can list jobs. It looks like

```
1    # Edit this file to introduce tasks to be run by cron.
2    #
3    # Each task to run has to be defined through a single line
4    # indicating with different fields when the task will be run
5    # and what command to run for the task
6    #
7    # To define the time you can provide concrete values for
8    # minute (m), hour (h), day of month (dom), month (mon),
9    # and day of week (dow) or use '*' in these fields (for 'any').
10   #
11   # Notice that tasks will be started based on the cron's system
12   # daemon's notion of time and timezones.
13   #
14   # Output of the crontab jobs (including errors) is sent through
15   # email to the user the crontab file belongs to (unless redirected).
16   #
17   # For example, you can run a backup of all your user accounts
18   # at 5 a.m every week with:
19   # 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
20   #
21   # For more information see the manual pages of crontab(5) and cron(8)
```

```
22  #
23  # m h  dom mon dow    command
```

In the bottom line, we can add the following to run `sudo apt update` every minute. The 5 columns refer to minute (0-59), hour (0-24), date of month (1-31), month (1-12), and date of week (0-7, where 0 and 7 is Sunday). The asterick means every instance of.

```
1  # Run every minute
2  * * * * * sudo apt update
3
4  # Run at 9:15am every first day of the month
5  15 9 1 * * sudo apt update
6
7  # Run for every minute of every hour for the 13th day of every month if it is Friday
8  * * 13 * 5 sudo apt update
```

You get the idea.

5.13   Graphics Drivers

Note that one type of data we must store on memory is the individual pixels in a computer screen. Say that in a 1920×1080 resolution computer, there are about $1920 \times 1080 \times 3 \approx 2$ million bytes of data that we have to store. This isn't that much data (only 2MB), but we must update it quite fast since our screens are always updating. This is why all computer which have a GUI comes with a built-in graphics driver. To see the GPU hardware specifications, install **lshw**.

Definition 5.5 (Graphics Processing Unit)

The **GPU** is a specialized processing unit that is designed to handle the rendering of images and videos. It is designed to handle the rendering of images and videos, and is optimized for parallel processing. Like the CPU, it has some common metrics:

1. **Clock Speed**: The speed at which the GPU can execute instructions. This is usually measured in MHz or GHz.
2. **Memory**: The amount of memory that the GPU has. This is usually measured in GB.
3. **Memory Bandwidth**: The speed at which the GPU can read and write to its memory. This is usually measured in GB/s.
4. **Cores**: The number of cores that the GPU has. This is usually measured in thousands, which allows for parallel processing.

You can check which GPUs you have by running `lspci | grep VGA` or `neofetch`. There are generally two types of GPUs:

1. **Integrated GPU**: This type of GPU is built into the same chip as the CPU (Central Processing Unit). It shares resources with the CPU, including memory, which can lead to reduced performance for graphics-intensive tasks. However, its integrated nature makes it more power-efficient and cost-effective.
2. **Discrete GPU**: This is a separate component from the CPU and comes with its own RAM (usually called VRAM or Video RAM). It is typically installed in a dedicated slot on the motherboard. Because it operates independently of the CPU, a discrete GPU can offer significantly better performance for graphics processing, gaming, or deep learning.

Definition 5.6 (Monitor)

Furthermore, your computer monitor, which actually displays these pixels to you, must also have metrics that match the GPU. Some properties:
1. The **resolution** is the number of pixels that the monitor can display, and is usually measured in pixels.
2. The **refresh rate** is the number of times the monitor can refresh the image on the screen per second, and is usually measured in Hz.

To see these metrics for all monitors connected to your computer, run **xrandr**, which lists all the resolutions and possible refresh rates for each resolution.

Definition 5.7 (Graphics Driver)

In order for your operating system to communicate with your GPU, you need a **graphics driver**. This is a piece of software that allows the operating system to communicate with the GPU. There are two main types of graphics drivers:
1. **Open Source Drivers**: These are drivers that are developed and maintained by the open source community. They are usually included with the Linux kernel, and are generally stable and reliable.
2. **Proprietary Drivers**: These are drivers that are developed and maintained by the GPU manufacturer. They are usually not included with the Linux kernel, and are generally more feature-rich and performant than open source drivers.

Intel drivers are open source, but Nvidia drivers are proprietary (which is why Linus Torvalds has beef with Nvidia).[a]

---

[a]A video of Linus Torvalds saying "fuck you" to Nvidia: https://www.youtube.com/watch?v=iYWzMvlj2RQ

Some popular graphics drivers include **mesa** for Intel and **nvidia** drivers for NVIDIA.

5.13.1   Multiple GPUs

Everything is pretty straightforward when you have one graphics card, but when you have multiple graphics cards, you have to specify which one you want to use. If you want to only use one GPU, you can just disable the other one in the BIOS. However, if you have an Intel/Nvidia dual driver and want to use both, install **Nvidia Optimius** (for Ubuntu, it is supported through **nvidia-prime**).[4],[5]

Now make sure that the systemd daemon is running, and you can call `optimus-manager -switch hybrid` to enable hybrid graphics. This will log you out.

5.14   Peripheral Devices

Peripheral devices refer to other devices outside of the motherboard, including mice, keyboards for input, monitors, printers, network managers, and usb ports. Even the GPU is considered a peripheral device. These must be connected to the motherboard in some way to be managed by the operating system, and similar to the databus connecting the CPU and memory, there are buses that connect the motherboard and these peripheral devices.

Definition 5.8 (PCI Bus)

The **PCI (Peripheral Component Interconnect)** bus is a high-speed bus that connects the motherboard to peripheral devices. It is used to connect devices like network cards, sound cards, and graphics cards to the motherboard. PCI buses operated based on the PCI standard, which is a set of

---

[4]This wiki article (https://github.com/Askannz/optimus-manager/wiki) provides a good overview of this matter.
[5]Installation instructions here: https://github.com/Askannz/optimus-manager?tab=readme-ov-file

specifications that define the physical and electrical characteristics of the bus.

The command to use to enumerate all PCI devices is `sudo lspci` (with `-v` for verbose).

```
1   00:00.0 Host bridge: Intel Corporation 10th Gen Core Processor
2   00:01.0 PCI bridge: Intel Corporation 6th-10th Gen Core Processor
3   00:02.0 VGA compatible controller: Intel Corporation CometLake-H
4   00:04.0 Signal processing controller: Intel Corporation Xeon
5   00:08.0 System peripheral: Intel Corporation Xeon E3-1200 v5/v6
6   00:12.0 Signal processing controller: Intel Corporation Comet
7   00:13.0 Serial controller: Intel Corporation Device 06fc
8   00:14.0 USB controller: Intel Corporation Comet Lake USB 3.1
9   00:14.2 RAM memory: Intel Corporation Comet Lake PCH Shared
10  00:14.3 Network controller: Intel Corporation Comet Lake PCH
11  00:15.0 Serial bus controller: Intel Corporation Comet Lake
12  00:15.1 Serial bus controller: Intel Corporation Comet Lake
13  00:16.0 Communication controller: Intel Corporation Comet
14  00:1c.0 PCI bridge: Intel Corporation Device 06b8 (rev f0)
15  00:1c.6 PCI bridge: Intel Corporation Device 06be (rev f0)
16  00:1d.0 PCI bridge: Intel Corporation Comet Lake PCI Express
17  00:1e.0 Communication controller: Intel Corporation Comet Lake
18  00:1f.0 ISA bridge: Intel Corporation Device 068e
19  00:1f.3 Audio device: Intel Corporation Comet Lake PCH cAVS
20  00:1f.4 SMBus: Intel Corporation Comet Lake PCH SMBus Controller
21  00:1f.5 Serial bus controller: Intel Corporation Comet Lake
22  01:00.0 3D controller: NVIDIA Corporation TU117M [GeForce GTX 1650
23  02:00.0 PCI bridge: Intel Corporation JHL7540 Thunderbolt 3 Bridge
24  03:00.0 PCI bridge: Intel Corporation JHL7540 Thunderbolt 3 Bridge
25  03:01.0 PCI bridge: Intel Corporation JHL7540 Thunderbolt 3 Bridge
26  03:02.0 PCI bridge: Intel Corporation JHL7540 Thunderbolt 3 Bridge
27  03:04.0 PCI bridge: Intel Corporation JHL7540 Thunderbolt 3 Bridge
28  04:00.0 System peripheral: Intel Corporation JHL7540 Thunderbolt
29  38:00.0 USB controller: Intel Corporation JHL7540 Thunderbolt 3
30  6c:00.0 Unassigned class [ff00]: Realtek Semiconductor Co., Ltd.
31  6d:00.0 Non-Volatile memory controller: Samsung Electronics Co
```

Figure 7: This is the following output of lspci on my personal computer.

5.15   Architecture

Arch Linux states on their website that they have *official packages optimized for the x86-64 architecture.*[6]

By running **cat /proc/cpuinfo**, you can see the specs of each CPU core you have. This includes the **model name** (clock cycle), **cache size**, **flags**, and **microcode**. The flags are the most important, since they tell you what features your CPU has.[7]

1. **lm**: 64 bit architecture.

2. **vmx** (Intel) or **svm** (AMD): Hardware virtualization .

3. **aes**: Accelerate AES encryption.

4. **fpu**: Floating Point Unit, which is used for floating point operations.

---

[6]https://archlinux.org/
[7]The entire list of flags and what they can do is mentioned in the Arch kernel source code, which is a good reference: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/include/asm/cpufeatures.h

5. **vme**: Virtual 8086 mode enhancements, which is used for virtualization.

6. **de**: Debugging extensions, which is used for debugging.

7. **pse**: Page Size Extensions, which is used for larger page sizes.

8. **tsc**: Time Stamp Counter, which is used for timing.

9. **msr**: Model Specific Registers, which is used for model specific operations.

10. **mce**: Machine Check Exception, which is used for error checking.

11. **pae**: Physical Address Extensions, which is used for larger memory.

12. **mce**: Machine Check Exception, which is used for error checking.

5.16   System Hardware

6　Remote Access and Cluster Computing with SSH

This is about remote accessing, mainly with SSH.

Definition 6.1 (SSH Configuration File)

In `$HOME/.ssh/`, you will see some public-private key files. You may be logging into remote servers as such.

```
1  > ssh username@123.123.123.123
2  > ssh username@login.duke.edu
```

This can be quite tedious, so you can create a `$HOME/.ssh/config` file to store shortcuts.

```
1  Host server1
2    HostName 123.123.123.123
3    User username
```

and then you can connect as such.

```
1  > ssh server1
```

Definition 6.2 (Installing Packages without Sudo)

Most likely when you are SSHing into a remote server, it is a computing cluster that you do not have admin access to, and you cannot thus include global packages. Therefore, the default method of installing binaries is by building them from source. You should therefore have a designated directory where you put all of your binaries in (I use `$HOME/.local/bin`) and include it in `$PATH`.

6.1　Clipboard

Clipboards have been a pain in the ass for me. Say that you connect to a remote server from your local machine, and then you open neovim in your remote machine. You can use `y` or `d` to copy/cut things and `p` to paste them in other buffers within the server, but you cannot easily copy/paste between your local machine and remote server. This becomes a pain when you need to paste some code on the remote server to GPT or when you take a code snippet from github and try to paste it on the buffer in the remote session. To fix this, you need a clipboard provider on the remote server. Do the following. I did this when my local machine has MacOS with ARM64 architecture, connecting to a remote machine running Ubuntu 22.04 with x86. Given the differences, the following steps should work.

1. *You may not have to do this step, but I did this while troubleshooting.* Install `xclip` on the remote server. If you don't have sudo access, then just build it from source. Then add the binary to somewhere in `$PATH`.

2. Install `lemonade` on both your local machine and the remote server. Make sure they are both in your `$PATH`.

3. Make a script on your local machine for sanity checking, call it `remote` or something, and add it to path. It should have the following. Note that the `-R`flag is important.[8]

```
1  #!/bin/bash
2  ps cax | grep lemonade> /dev/null
3  if [ $? -eq 0 ]; then
4    echo "lemonade is running."
```

---

[8]https://gist.github.com/bketelsen/27c2cd5b1376e72e240321baa0fbc81a

```
5  else
6    echo "lemonade is not running."
7    nohup lemonade server &
8  fi
9  ssh -R 2489:127.0.0.1:2489 mb625@123.123.123.123
```

4. Make sure to put the following in your `config` file. I put it in both your local and remote.[9]

```
1  ForwardX11 yes
2  ForwardX11Trusted yes
```

5. You may also have to set `clipboard=unnamedplus`. I had this on by default.

6. Make sure that you do not have the keymaps `y` to `+y` enabled.

We are done. Run `:checkhealth` on neovim in your remote server and confirm that lemonade is detected.

6.2   High Performance Computing with Slurm

You will most likely be remotely connecting to a server to access a compute cluster. This is a giant server consisting of hundreds or thousands of processing units, called a **cluster**. A cluster has different **partitions**, and within each partition contains **nodes**. In each node, there is a specific number of CPUs and GPUs available. There is a natural queuing system that connects users to nodes, through an open-source workload manager called **slurm**.

As soon as you ssh into the cluster, you will most likely be in a **login node**.

```
1  >>> hostname
2  compsci-login-03
```

While it does have CPUs, you generally don't want to use this for computation. It usually contains a few CPUs.

```
1  >>> lscpu
2  Architecture:          x86_64
3    CPU op-mode(s):      32-bit, 64-bit
4    Address sizes:       42 bits physical, 48 bits virtual
5    Byte Order:          Little Endian
6  CPU(s):                6
7    On-line CPU(s) list: 0-5
8  Vendor ID:             GenuineIntel
9    Model name:          Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz
10   ...
```

Figure 8: There are only 6 CPUs on this list.

You want to switch to a **compute node** for heavy computational tasks.

6.2.1   Exploring Specs and Status with sinfo

Before you even connect to a compute node, you want to see which nodes are available. In here, the **sinfo** command is your friend.[10] Just running it in your login node allows you to list the partitions on the cluster and see their availability. Note that given some partition, e.g. `compsci`, it consists of the nodes `linux[1-40]`.

---

[9]https://github.com/neovim/neovim/issues/8028
[10]The man pages is a 5 min read and explains everything in this subsection.

```
1   >>> sinfo
2   PARTITION     AVAIL  TIMELIMIT  NODES  STATE NODELIST
3   compsci          up 90-00:00:0      1 drain* linux23
4   compsci          up 90-00:00:0      4    mix linux[31,35-37]
5   compsci          up 90-00:00:0     21  alloc linux[1-7,9-10,21-22,24-30,32-34]
6   compsci          up 90-00:00:0     13   idle linux[11-20,38-40]
7   compsci-gpu*     up 90-00:00:0      3   mix- compsci-cluster-fitz-[06,08,20]
8   compsci-gpu*     up 90-00:00:0      1  down* gpu-compute7
9   compsci-gpu*     up 90-00:00:0      7   drng compsci-cluster-fitz-[04,19,29],...
10  compsci-gpu*     up 90-00:00:0     40  drain compsci-cluster-fitz-[01-03,09-18,...],...
11  compsci-gpu*     up 90-00:00:0      4    mix compsci-cluster-fitz-[05,07],...
12  compsci-gpu*     up 90-00:00:0      3  alloc linux[41-43]
13  grisman          up 60-00:00:0      1   plnd grisman-37
14  grisman          up 60-00:00:0      2    mix fennario-[02,04]
15  grisman          up 60-00:00:0      5  alloc grisman-40,jerry[1,3,6-7]
16  grisman          up 60-00:00:0      7   idle fennario-[01,03,05-06],jerry[2,4-5]
17  wiseman          up 60-00:00:0      1    mix wiseman-01
18  nlplab           up 60-00:00:0      1    mix nlplab-01
19  nlplab-core      up 90-00:00:0      1    mix nlplab-core-01
20  bhuwan           up 90-00:00:0      1    mix bhuwan-01
21  rudin            up 90-00:00:0      1   mix- rudin-01
22  skynet           up 90-00:00:0      1   idle skynet-02
23  wills            up 90-00:00:0      1   idle olympus
```

One of them is in drain state, 4 are in mix, 21 in alloc, and 13 idle. To clarify what each of these statuses mean, lets properly define them.

Definition 6.3 (Compute Node Status Codes)

1. **idle** - Node is completely free and available for new jobs.
2. **alloc** - Node is fully allocated/occupied by running jobs.
3. **mix** - Node is partially allocated (some CPUs/GPUs free, some busy). You can potentially get resources on these nodes.
4. **mix-** - Mixed state with some issue (the minus indicates a problem). Node has available resources but may have hardware issues.
5. **drain** - Node is being drained (no new jobs accepted, finishing current jobs). Jobs can finish but no new jobs will start.
6. **drng** - Node is currently draining (actively finishing jobs before maintenance).
7. **down*** - Node is offline/broken (asterisk indicates it's not responding).
8. **drain*** - Node is drained with issues (asterisk indicates problems).
9. **plnd** - Node is in planned maintenance.

Sometimes, you may want to focus on the nodes rather than the partitions, since the nodes are what you will be connecting to. Therefore, you can list the nodes first and detail their specs. The `-N` flag makes it node-centric.

```
1   >>>sinfo -N
2   NODELIST                 NODES    PARTITION STATE
3   bhuwan-01                    1       bhuwan mix
4   compsci-cluster-fitz-01      1 compsci-gpu* mix
5   compsci-cluster-fitz-02      1 compsci-gpu* idle
6   compsci-cluster-fitz-03      1 compsci-gpu* idle
7   compsci-cluster-fitz-04      1 compsci-gpu* mix
8   compsci-cluster-fitz-05      1 compsci-gpu* mix
9   compsci-cluster-fitz-06      1 compsci-gpu* mix
```

However, it only shows the node name, partition, and state. If we want to look at more detailed specs, we can put the `long` tag, which shows more detail on CPUs and the memory.

```
>>> sinfo -N --long
NODELIST                  PARTITION      STATE CPUS    S:C:T MEMORY    WEIGHT AVAIL_FE
bhuwan-01                    bhuwan     mixed 64      2:32:1  101095        1   (null)
compsci-cluster-fitz-01  compsci-gpu*   mixed 48      2:12:2  768414       49    a5000
compsci-cluster-fitz-02  compsci-gpu*    idle 48      2:12:2  768414       49    a5000
compsci-cluster-fitz-03  compsci-gpu*    idle 48      2:12:2  768414       49    a5000
compsci-cluster-fitz-04  compsci-gpu*   mixed 48      2:12:2  768414       49    a5000
compsci-cluster-fitz-05  compsci-gpu*   mixed 48      2:12:2  768414       99    a6000
compsci-cluster-fitz-06  compsci-gpu*   mixed 48      2:12:2  768414       99    a6000
```

Let's clarify what the additional columns mean.

1. The `CPUS` refer to the number of CPUs on the node. This number should be equal to the product of the `S:C:T`.

2. The `S:C:T` refers to the Socket:Core:Thread configuration. The socket is the number of physical CPU packages (chips) installed on the motherboard. Each socket contains one complete processor. The core is the number of physical processing cores per socket. Each core can execute instructions independently. The thread is number of hardware threads per core, enabled by technologies like Intel's Hyperthreading or AMD's SMT (Simultaneous Multithreading)

3. The `MEMORY` is the available RAM in MB.

4. The `WEIGHT` means priority weight for job scheduling.

5. `AVAIL_FEATURES` provides simple text labels like `a6000` that can be used as job constraints to target specific node types.

We can see CPU information, but for most cases we would like to focus on the GPU specs. This is usually stored in the `GRES` (generic resources) column, which specifies the actual GPU hardware installed in each node. It is in the format `gpu:type:count` (e.g. `gpu:a6000:4` for four A6000 GPUs). Unfortunately, it is not contained in the long tag. To inspect this, we will need to specify the format using the `-o` tag.

```
sinfo -N -o "%.25N %.6t %.4c %.4X %.8m %.20G %.15f"
            NODELIST  STATE CPUS SOCK   MEMORY                 GRES  AVAIL_FEATURES
            bhuwan-01    mix   64    2  1010955  gpu:a6000:8(S:0-1)          (null)
compsci-cluster-fitz-01    mix   48    2   768414  gpu:a5000:4(S:0-1)           a5000
compsci-cluster-fitz-02   idle   48    2   768414  gpu:a5000:4(S:0-1)           a5000
compsci-cluster-fitz-03   idle   48    2   768414  gpu:a5000:4(S:0-1)           a5000
```

Note that there is a lot of overlap between the `GRES` and `AVAIL_FEATURES` columns, but they are fundamentally different. While `GRES` defines what hardware exists, while `AVAIL_FEATURES` offers convenient tags for job submission, which will be elaborated later.

6.2.2 Interactive Jobs with srun

There are two ways to submit or run jobs. In the beginning, you might prototype a lot and want to have continuous access to your code on a compute node as you debug. At this point, it is best to use an **interactive** shell.

In interactive mode, you are changing your host in your current terminal session from your login node to the compute node, allowing you to iterate and debug easily. You can connect to the first available compute node by running `srun` with the `-i` flag (for interactive) and `pty` (for pseudo-terminal) set to whatever shell you want.

```
>>> hostname
```

```
2  compsci-login-03
3  >>> srun --pty zsh -i
4  srun: job 9601353 queued and waiting for resources
5  srun: job 9601353 has been allocated resources
6  >>> bin hostname
7  linux54
```

This is the minimalistic way, but you would like more control over the session you want to connect.

Definition 6.4 (Node Specification Options)

Here is more detail on tags related to specifying which node you want.
1. -partition=compsci-gpu. Look for nodes in the compsci-gpu partition.
2. -gres=gpu:a5000:1. Look for nodes that have gpu:a5000 in the GRES column. This is a direct resource request.
3. -constraint=a6000. Look for nodes that have a6000 in the AVAL_FEATURES column. This is a feature-based node selection.
4. -vvv for verbosity.

Definition 6.5 (Constraint Options)

Another natural property to have is the ability to limit resources beyond the physical capabilities of each node.
1. -time=1:00:00. Run it for 1 hour.
2. -mem=16G
3. -cpus-per-task=4

Here is an example. Even better, you can use tmux so that your progress is not lost when you lose connection.

```
1  srun --nodes=1 --ntasks-per-node=1 --partition=compsci-gpu --time=1:00:00 --gres=gpu:a5000
       --pty zsh -i -vvv
```

6.2.3 Batch Jobs with sbatch

Once you are done, you probably want to run large-scale tests. This requires you to submit some script with different arguments on not just one node, but multiple runs (perhaps over multiple combinations of hyperparameters) over different nodes. This is best done through **batch runs**, called through the sbatch command.

We will first perform the simplest sbatch script, and I heavily recommend to run this as a preliminary troubleshooting step in order to make sure everything is working properly before you even try to run your script in a batch job.

Theorem 6.1 (Batch Job to Print Hostname)

We will connect to a compute node and simply print out the hostname to verify that we have successfully connected. This command that you run when you connect to a node can be inputted into the -wrap argument. The standard out and error is by default logged in slurm-jobid.out in the directory where you submitted the sbatch script in.

```
1  >>> ls
2  >>> hostname
3  compsci-login-03
4  >>> sbatch --wrap="hostname"
5  Submitted batch job 9601364
6  >>> ls
7  slurm-9601364.out
8  >>> cat slurm-9601364.out
9  linux54
```

Figure 9: I start in the login node `compsci-login-03`. Then I call `hostname` which returns `linux54`.

Batch jobs give us more flexibility in some ways and less in others. For example, we don't need to be connected to the session if we submit a batch script.[11] Running dozens of batch jobs may be hard to keep track of, so it is recommended to make a custom name for each one. Additionally, in interactive shells, we can see standard output and error in real time. We do not have such an interface in batch runs, so we must direct it to some other file. Usually, it is a file called `jobname-jobid.out` and `jobname-jobid.err`. These are additional arguments that are a good idea to set when running batch jobs.

Definition 6.6 (Naming and Logging Options)

All of the tags representing
1. `-job-name`. The job name, which can be seen under the `NAME` column when calling `squeue`.
2. `-output`. Where you should log standard output.
3. `-error`. Where you should log standard error.

Great, so we have introduced our first way to run a batch job: through the `-wrap` argument.

Theorem 6.2 (Run Batch Jobs with Wrap)

Here is a complete script. The `%x` refers to the name and `%j` refers to the job id.

```
1   sbatch \
2       --job-name="whatever" \
3       --output="logs/%x-%j.out" \
4       --error="logs/%x-%j.err" \
5       --mem=16G \
6       --cpus-per-task=4 \
7       --gres=gpu:1 \
8       --time=4:00:00 \
9       --wrap="
10          cd /usr/project/xtmp/mb625/runs
11          python run.py
12      "
```

While you can put all of your commands to run on the compute node into the `-wrap` argument, this might be messy and inconvenient. A more efficient thing to do is to run it as a bash script.

---

[11]Though we can do this with interactive runs on tmux.

Theorem 6.3 (Run Batch Jobs as a Bash Script)

You create a normal bash script, but you input the sbatch arguments at the top as such. After that, you can write your script.

```
>>> cat run.sh
#!/usr/bin/env bash
#SBATCH --mem=40gb
#SBATCH --time=48:00:00
#SBATCH --partition=compsci-gpu
eval "\$(conda shell.bash hook)"
conda activate intnn
python main.py
>>> sbatch run.sh
```

Theorem 6.4 (Tips and Workflow for Batch Scripts)

I want to end with some convenient tips for debugging on batch runs. Unlike interactive runs, where you can just see where you are and debug code in real time, you have less visibility for batch runs. You should not assume that the conda environment will automatically be set up, nor that you will start in a specific directory. So before you have your scripts to run, you should probably just run a dummy script that ensures that everything up to the pre-script echo statement runs.

1. Ensure that you are using the proper shell, with environment variables properly set (especially `PATH`).
2. Source and activate your conda environment.
3. cd to the desired directory.
4. Echo that you are about to start, along with relevant logs.
5. Run your script.
6. Echo that you have just finished, along with relevant logs.

6.2.4   Checking Queue with squeue

To check the status of submitted, running, or canceled jobs, `squeue` is your friend. Just running squeue will list out all jobs from all users (?).

```
>>> squeue
     JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
   9597293    bhuwan gpu-bash     rx55  R 1-00:12:41      1 bhuwan-01
   9597071    compsci fmri_pre    zy151  R   13:33:36      1 linux25
   9601348    compsci    alpha     tqt3  R      29:53      1 linux24
   9597328    compsci sys/dash    yw628  R   22:20:41      1 linux31
   9597434 compsci-g cheshire    xz424 PD       0:00      1 (Resources)
   9597436 compsci-g   iSB619    xz424 PD       0:00      1 (Priority)
   9597673 compsci-g cheshire    xz424 PD       0:00      1 (Priority)
   9591596 compsci-g      zsh    rt195  R 3-00:10:50      1 linux54
   9600314 compsci-g run_all_    sf349  R    3:03:14      1 linux52
 9590035_0 compsci-g sw-1q57j   cjb131  R 3-16:55:33      1 compsci-cluster-fitz-19
9597248_47 compsci-g sn_label    dt161  R   12:45:35      1 compsci-cluster-fitz-20
9597248_48 compsci-g sn_label    dt161  R   12:45:35      1 compsci-cluster-fitz-20
9597248_49 compsci-g sn_label    dt161  R   12:45:35      1 compsci-cluster-fitz-20
```

Rather than looking at all queued jobs, you probably want to focus on a specific user. You can either filter the current runs from yourself,

---

```
1  >>> squeue --me
2    JOBID PARTITION      NAME    USER ST      TIME  NODES NODELIST(REASON)
3  9601352 compsci-g      zsh    mb625  R      7:18      1 linux54
```

or from a specific user.

```
1  >>> squeue -u cjb131
2      JOBID PARTITION      NAME     USER ST       TIME  NODES NODELIST(REASON)
3   9590035_0 compsci-g sw-1q57j   cjb131  R 3-16:59:45      1 compsci-cluster-fitz-19
```

Less frequently, you might want to focus on a specific job ID. You can input this through the `-j` (job) tag.

```
1  >>> squeue -j 9597376
2    JOBID PARTITION      NAME    USER ST      TIME  NODES NODELIST(REASON)
3   9597376     rudin vscode-s   dt161  R   9:02:39      1 rudin-01
```

6.2.5   Canceling Jobs with scancel

To cancel a current job, `scancel` is your friend. To cancel a specific jobs, you just write it down.

```
1  >>> scancel 9601352
2  srun: Job step aborted: Waiting up to 32 seconds for job step to finish.
3  slurmstepd: error: *** STEP 9601352.0 ON linux54 CANCELLED AT 2025-07-12T15:19:14 ***
```

To cancel all of your own jobs, run

```
1  >>> scancel --me        # one way
2  >>> scancel -u $whoami    # another way
```

6.2.6   Multi-GPU or Multi-Node Runs

7   Unit and Integration Tests

In general, if you can reduce the number of lines of code to accomplish a task, it is seen as a good thing. You are (most of the time) simplifying the logic and therefore reducing the probability of there being bugs. This leads to our first rule.

Theorem 7.1

Code is not an asset. It's a liability.

However, codebases must grow, and therefore they must be maintained. As a student, I never worked with unit tests since I never developed something large or complex enough to warrant them. This is the case for most college students, and unless you start working—either during an internship or full-time—you may not even know unit testing even exists. When I wrote my first unit tests during my internship in junior year, the existing testing suite was massive and a lot to take in. I was supposed to write separate tests for the new features I was working on, but I felt like I was winging it since I never properly learned how to write tests. I've read up on a few books and used them to write this guide.

1. Korikov's *Unit Testing: Principles, Practices, Patterns*

There are two general schools of unit testing (classical/Detroit and London), and I present the classical school's definition below.

Definition 7.1 (Unit Test)

In classical unit testing, a **unit test** is a test that satisfies the 3 properties.
  1. *Atomicity.* It verifies a single unit of behavior.[a]
  2. *Efficiency.* It does it quickly, and
  3. *Isolation.* It does it in isolation from other tests.[b]

  ---
  [a]This may or may not encompass units of *code*, which are generally classes in OOP.
  [b]TBD

The London school still asserts efficiency, but the atomicity and isolation requirements are stricter. The basic idea is that say you have two classes A, B with B a child of A. You want to test the behavior of each class and therefore have test suites T_A, T_B for each class. Now say that there is a bug in T_A. Then T_A and T_B will fail, even though the singular problem was in A. This doesn't provide the isolation that we need, since T_C for any class C should fail if and only if there is a bug in C. This is why in the London school, you make *double/mock classes*, which are minimal copies of the class which are created for testing only.

Definition 7.2 (Integration Test)

An **integration test** is a test doesn't satisfy precisely one of the three properties.
  1. *Atomicity.* You may need to see how two units of code act together.
  2. *Efficiency.* This is subjective depending on your time constraints.
  3. *Isolation.* Multiple tests may use a shared dependency.
An **end-to-end test** is usually defined as not satisfying both atomicity and isolation, which often means that it doesn't run quickly either.

What should unit tests achieve?

1. *Protection against regressions.*[12] When you add a new feature that introduces a new bug, the tests should find that bug and report it before the features is pushed into production. Having good protection reduces the probability of tests passing that should be actually failing, i.e. false negatives.

---
[12]A regression is a software bug.

2. *Resistance to Refactoring.* A test should still be runnable after refactoring your code, i.e. should not be about implementation details. This is a binary attribute: a test either has resistance or it does not. If you don't do this, then you will generate a lot of tests that should pass but fail due to refactoring, i.e. false positives. This brittleness dilutes your ability and willingness to react to problems in the code.

3. *Fast Feedback.* Should be fast.

4. *Maintainability.* Should be maintainable.

With this in mind, we will talk about the ways in which we can write tests.

7.1   Structure of Unit Tests

To write a proper unit test, use the following, also called the Given-When-Then pattern.

1. *Arrange.* Bring the system under test (SUT) and dependencies to a desired state. This is usually the largest of the three, but if it's significantly larger than the act and assert sections combined, then it's a good idea to extract the arrangements either into private methods or a separate function.

2. *Act.* Call methods on SUT, pass the prepared dependencies, and capture the output (if any).

3. *Assert.* Verify the outcome. This should be a single line of a method call, and be careful if it is not because then this indicates that an atomic behavior is really 2 methods, and there is an incorrect design choice somewhere. Then it can be followed by one or more assertions.

4. *Teardown.* Depending on the language, this may be necessary if there is no automatic garbage disposal.

If you have an arange, act, assert, act, assert, etc., then this is a sign that it's an integration test. Alternatively, you should refactor it so that it is a sequence of unit tests.

Theorem 7.2 (Avoid If Statements in Tests)

An if statement is a conditional, and this is branching behavior that we don't want in a linear sequence of code in a unit test.

Assert statements should be outputted by verbose messages.

7.1.1   Output Based Testing

Definition 7.3 (Output-Based (Functional) Testing)

7.1.2   State Based Testing

Definition 7.4 (State-Based Testing)

7.1.3   Communication Based Testing

Definition 7.5 (Communication-Based Testing)

7.2   Continuous Integration (CI) with Git Actions and Docker

Continuous integration (CI), or **continuous development (CD)**, refers to any automated process that runs whenever you perform some action on a repository. These can include:

1. Compiling your package upon pushing to a git branch. This saves the time of manually compiling it yourself.

2. Compiling and/or running unit tests on your package, over possibly different compiler/interpreter versions on different operating systems and different architectures, whenever someone opens a pull request. This is usually done by automatically creating docker images and running a script that sets up the environment for your system.

3. Automatically publishing a new package version to PyPI upon a push to the master branch of a repository.

Github actions provide **workflow scripts** that you can include in your repository's `github/workflows/` directory that automates this. They are essentially yaml files that activate upon some command, whether that'd be a push to a branch, a pull request, or even the completion/failure of another workflow. This gives great convenience in deploying code.

8 Profiling

8.1 Python

The main tool I use to profile Python code is the `cProfile` module. It is a *determinstic* profiler and is very good at handling large scripts over large codebases. To print the profiling outputs, run

```
1  python -m cProfile -s time script.py --args ...
```

where `-s` is the parameter in which the function calls are sorted. This gives you something like

```
1  26457919 function calls (25109295 primitive calls) in 37.789 seconds
2
3  Ordered by: internal time
4  List reduced from 17047 to 100 due to restriction <100>
5
6     ncalls  tottime  percall  cumtime  percall filename:lineno(function)
7   61380/60    2.684    0.000    7.607    0.127
         /usr/project/xtmp/mb625/protopnext-image-genetic/protopnet/prediction_heads.py:157(forward)
8        18    1.578    0.088    5.114    0.284
         /usr/project/xtmp/mb625/protopnext-image-genetic/protopnet/prototype_layers.py:453(u...
9     84718    1.370    0.000    1.370    0.000 {built-in method torch.tensor}
10    10993    1.339    0.000    1.360    0.000 {method 'to' of 'torch._C._TensorBase' objects}
11      549    0.987    0.002    0.987    0.002 {built-in method torch.conv2d}
12   122760    0.903    0.000    0.903    0.000 {built-in method torch.rsub}
13    73728    0.898    0.000    2.021    0.000
         /usr/project/xtmp/mb625/protopnext-image-genetic/protopnet/prediction_heads.py:349
14    33600    0.878    0.000    0.878    0.000
         /usr/project/xtmp/mb625/protopnext-image-genetic/protopnet/datasets/bioscan_genetic.py:56
```

Some info:

1. Note that these times refer to the time it took to run the function *in its current stack frame* (so not profiled for any subfunction calls).

2. The `ncalls` give you one number which give you the number `t` times this function was called. If it is form `r/t`, then `t` refers to the unique calls, and `r` refers to the total number of calls, including recursive calls.

3. The `filename:lineno(function)` precisely tells you which function we are profiling.

For more control over the analysis, you should write the profiling information to some text file. You can specify this with the `-o` arg (the `-s` doesn't do anything).

```
1  python -m cProfile -o profile.txt script.py --args ...
```

The saved file is not directly readable, and you need the `pstats` module to parse it.

```
1  import pstats
2
3  p = pstats.Stats('profile.txt')          # load the data
4  p.strip_dirs() \                         # can strip away all leading path information from
       file names
5    .sort_stats('time') \                  # sort it by time
6    .print_stats(100)                      # print the top 100 function calls
```

Say that we saw that the function call of `'torch.cat'` was causing the biggest bottleneck. To see where

each **'cat'** was called, we can write

```
1  p.print_callers(100, 'cat')
```

8.2   C++

9 Debugging

10   Design Patterns

At some point, you would have mastered the syntax and your main bottleneck will be how well you can design your repository. Chances are, you would either have specific needs or would see the same problems over and over again. The following *design patterns* are general ways that you can structure your code to meet these needs or avoid these problems.

Definition 10.1 (UML Diagram)

A **UML (Unified Modeling Language) diagram** is a directed graph of nodes.
   1. Each node is a class C, labeled in a box, with attributes and methods.
   2. A node can *extend* a class, i.e. is a child class.
   3. A node can *implement* a class, i.e. is an implementation of an interface.

In the Gang of 4's *Design Patterns*, they go over 23 design patterns. I'll list them briefly here for a quick reference, but each subsection will go through in detail the following:

1. The motivation behind this pattern. Mainly, when to use it, and what natural problems motivate this pattern.

2. The solution, which describes the elements that make up this design, their relationships, responsibilities, and collaborations.

3. The consequences, which are the results and trade-offs of applying the pattern.

4. A real-life example of when I had to use this pattern, with code and a UML diagram.

10.1   Creational Patterns

How objects are instantiated.

10.1.1   Singleton

A class is a blueprint, so it should be the case that we would instantiate multiple objects from this class, right? Not always. There are times when we want just *one* instance o of a class C to exist, and it is usually in the case when a bunch of other objects $o_1, \ldots, o_n$ need to interact with o uniformly.

Definition 10.2 (Singleton)

In a **singleton pattern**, we want only one instance of a specific class to exist. It is characterized by the following.
   1.

Results.

Example 10.1 (Logger)

Say that you want to implement a simple logger. You want there to be only one logger object that writes to a central location.

```
1   class Logger:
2     _instance = None
3
4     def __new__(cls):
5       if cls._instance is None:
6         cls._instance = super().__new__(cls)
```

```
 7          cls._instance.logs = []   # Initialize logs only once
 8      return cls._instance
 9
10  def log(self, message):
11      self.logs.append(message)
12      print(f"[LOG] {message}")
13
14  def show_all_logs(self):
15      print("\n=== All Logs ===")
16      for i, log in enumerate(self.logs, 1):
17          print(f"{i}. {log}")
```

```
 1  # Usage example
 2  def main():
 3      # First logger instance
 4      logger1 = Logger()
 5      logger1.log("Application started")
 6      logger1.log("User logged in")
 7
 8      # Second logger instance (same as first!)
 9      logger2 = Logger()
10      logger2.log("Processing data")
11
12      # Third instance from different part of code
13      logger3 = Logger()
14      logger3.log("Task completed")
15
16      # All loggers are the same instance
17      print(f"\nlogger1 is logger2: {logger1 is logger2}")
18      print(f"logger2 is logger3: {logger2 is logger3}")
19
20      # All share the same log history
21      logger1.show_all_logs()
22
23  if __name__ == "__main__":
24      main()
```

```
 1  [LOG] Application started
 2  [LOG] User logged in
 3  [LOG] Processing data
 4  [LOG] Task completed
 5
 6  logger1 is logger2: True
 7  logger2 is logger3: True
 8
 9  === All Logs ===
10  1. Application started
11  2. User logged in
12  3. Processing data
13  4. Task completed
14  .
15  .
16  .
17  .
18  .
19  .
20  .
21  .
22  .
23  .
24  .
```

10.1.2   Factory

When a method returns one of several possible classes that share a common super class. You would want some kind of "factory" that can seamlessly generate new objects for you. We want to emphasize that this behavior is generally wanted at runtime. You should use it

1. when you don't know ahead of time what class object you need.

2. when all of the potential classes are in the same subclass hierarchy.

3. to centralize class selection code.

4. when you don't want to use to have to know every subclass.

5. to encapsulate object creation.

Definition 10.3 (Factory Pattern)

In a **factory pattern**, you can create objects without specifying their exact class. It is characterized by the following.
1. It is especially useful for creating objects at runtime.
2. It specifically avoids using a constructor and rather uses a factory method for object construction.
3. Factory methods can be specified in an interface and implemented by subclasses or implemented in a base class and optionally overridden by subclasses.

Results.

Example 10.2 (Animal Factory)

```python
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()

# Usage
animal = AnimalFactory.create_animal("dog")
print(animal.speak())  # Woof!
```

Example 10.3 (Case Study)

`torch.randn()` is a factory method that generates torch tensors sampled element-wise from a standard Gaussian!

10.1.3 Builder

Motivation.

Definition 10.4 (Builder Pattern)

In a **builder pattern**, we construct complex objects step by step. It is characterized by the following.
1. It is used to create objects made from a bunch of other objects.
2. The creation of the parts are hidden from the client.
3. Unlike factory—where we just instantiate generic instances of a class—the builder gives us precise control of instantiation.

Results. However, it has gotten a bit out of style recently.

Example 10.4 (Pizza)

```python
class Pizza:
    def __init__(self):
        self.size = None
        self.cheese = False
        self.pepperoni = False

    def __str__(self):
        return f"{self.size} pizza, cheese: {self.cheese}, pepperoni: {self.pepperoni}"

class PizzaBuilder:
    def __init__(self):
        self.pizza = Pizza()

    def set_size(self, size):
        self.pizza.size = size
        return self

    def add_cheese(self):
        self.pizza.cheese = True
        return self

    def add_pepperoni(self):
        self.pizza.pepperoni = True
        return self

    def build(self):
        return self.pizza

# Usage
pizza = PizzaBuilder().set_size("large").add_cheese().add_pepperoni().build()
print(pizza)  # large pizza, cheese: True, pepperoni: True
```

10.1.4   Prototype

Motivation.

Definition 10.5 (Prototype Pattern)

In a **prototype pattern**, we create new objects by copying existing ones. It is characterized by the following.

Results.

Example 10.5 (Document)

```python
import copy

class Document:
    def __init__(self, title, content):
        self.title = title
        self.content = content

```

```
8      def clone(self):
9          return copy.deepcopy(self)
10
11     def __str__(self):
12         return f"Title: {self.title}, Content: {self.content}"
13
14 # Usage
15 original = Document("Report", "Q1 Sales Data")
16 clone = original.clone()
17 clone.title = "Report Copy"
18 print(original)  # Title: Report, Content: Q1 Sales Data
19 print(clone)     # Title: Report Copy, Content: Q1 Sales Data
```

10.1.5   Abstract Factory

Motivation.

Definition 10.6 (Abstract Factory Pattern)

In an **abstract factory pattern**, it creates families of related objects without specifying their concrete class. It is characterized by the following.

    1.

Results.

Example 10.6 (Case Study)

```
1  from abc import ABC, abstractmethod
2
3  # Abstract Products
4  class Button(ABC):
5      @abstractmethod
6      def render(self):
7          pass
8
9  class Checkbox(ABC):
10     @abstractmethod
11     def render(self):
12         pass
13
14 # Concrete Products - Windows
15 class WindowsButton(Button):
16     def render(self):
17         return "Rendering Windows button"
18
19 class WindowsCheckbox(Checkbox):
20     def render(self):
21         return "Rendering Windows checkbox"
22
23 # Concrete Products - Mac
24 class MacButton(Button):
25     def render(self):
26         return "Rendering Mac button"
27
```

```python
28  class MacCheckbox(Checkbox):
29      def render(self):
30          return "Rendering Mac checkbox"
31
32  # Abstract Factory
33  class GUIFactory(ABC):
34      @abstractmethod
35      def create_button(self):
36          pass
37
38      @abstractmethod
39      def create_checkbox(self):
40          pass
41
42  # Concrete Factories
43  class WindowsFactory(GUIFactory):
44      def create_button(self):
45          return WindowsButton()
46
47      def create_checkbox(self):
48          return WindowsCheckbox()
49
50  class MacFactory(GUIFactory):
51      def create_button(self):
52          return MacButton()
53
54      def create_checkbox(self):
55          return MacCheckbox()
56
57  # Usage
58  def create_ui(factory):
59      button = factory.create_button()
60      checkbox = factory.create_checkbox()
61      print(button.render())
62      print(checkbox.render())
63
64  # Create Windows UI
65  print("Windows UI:")
66  create_ui(WindowsFactory())
67
68  # Create Mac UI
69  print("\nMac UI:")
70  create_ui(MacFactory())
```

10.2 Structural Patterns

How objects relate to each other.

10.2.1 Adapter

Motivation.

Definition 10.7 (Adapter Pattern)

An **adapter pattern** converts one interface to another that clients expect.
1. It allows classes to work together that on their own could not due to incompatible interfaces. It converts the interface of one class into something that may be used by another class.

Results.

Example 10.7 (Sockets)

```python
class EuropeanSocket:
    def plug_in_eu(self):
        return "Providing 230V (EU)"

class USASocket:
    def plug_in_usa(self):
        return "Providing 110V (USA)"

class SocketAdapter:
    def __init__(self, socket):
        self.socket = socket

    def plug_in(self):
        if hasattr(self.socket, 'plug_in_eu'):
            return self.socket.plug_in_eu()
        elif hasattr(self.socket, 'plug_in_usa'):
            return self.socket.plug_in_usa()

# Usage
eu_socket = EuropeanSocket()
usa_socket = USASocket()

adapter1 = SocketAdapter(eu_socket)
adapter2 = SocketAdapter(usa_socket)

print(adapter1.plug_in())  # Providing 230V (EU)
print(adapter2.plug_in())  # Providing 110V (USA)
```

10.2.2   Bridge

Motivation.

Definition 10.8 (Bridge Pattern)

In a **bridge pattern**, it separates abstraction from implementation so they can vary independently.
1.

Results.

Example 10.8 (Case Study)

```python
# Implementation
class Color:
```

```
3        def fill(self):
4            pass
5
6    class Red(Color):
7        def fill(self):
8            return "red"
9
10   class Blue(Color):
11       def fill(self):
12           return "blue"
13
14   # Abstraction
15   class Shape:
16       def __init__(self, color):
17           self.color = color
18
19       def draw(self):
20           pass
21
22   class Circle(Shape):
23       def draw(self):
24           return f"Drawing circle in {self.color.fill()}"
25
26   class Square(Shape):
27       def draw(self):
28           return f"Drawing square in {self.color.fill()}"
29
30   # Usage
31   red = Red()
32   blue = Blue()
33
34   red_circle = Circle(red)
35   blue_square = Square(blue)
36
37   print(red_circle.draw())    # Drawing circle in red
38   print(blue_square.draw())   # Drawing square in blue
```

10.2.3   Composite

Motivation.

Definition 10.9 (Composite Pattern)

In a **composite pattern**, we treat individual objects and compositions uniformly in a tree structure.
  1.

Results.

Example 10.9 (Case Study)

```
1    class Component:
2        def show(self, indent=0):
3            pass
4
```

```
5  class File(Component):
6      def __init__(self, name):
7          self.name = name
8
9      def show(self, indent=0):
10          print("  " * indent + f"- {self.name}")
11
12  class Folder(Component):
13      def __init__(self, name):
14          self.name = name
15          self.children = []
16
17      def add(self, component):
18          self.children.append(component)
19
20      def show(self, indent=0):
21          print("  " * indent + f"+ {self.name}/")
22          for child in self.children:
23              child.show(indent + 1)
24
25  # Usage
26  root = Folder("root")
27  docs = Folder("documents")
28  pics = Folder("pictures")
29
30  docs.add(File("resume.pdf"))
31  docs.add(File("report.docx"))
32  pics.add(File("photo1.jpg"))
33
34  root.add(docs)
35  root.add(pics)
36  root.add(File("readme.txt"))
37
38  root.show()
```

10.2.4   Decorator

Motivation.

Definition 10.10 (Decorator)

In a **decorator pattern**, we add functionality to objects dynamically. It is characterized by the following.

   1.

Results.

Example 10.10 (Coffee)

```
1  class Coffee:
2      def cost(self):
3          return 5
4
5  class MilkDecorator:
```

```
6        def __init__(self, coffee):
7            self.coffee = coffee
8
9        def cost(self):
10           return self.coffee.cost() + 2
11
12   class SugarDecorator:
13       def __init__(self, coffee):
14           self.coffee = coffee
15
16       def cost(self):
17           return self.coffee.cost() + 1
18
19   # Usage
20   coffee = Coffee()
21   coffee_with_milk = MilkDecorator(coffee)
22   coffee_with_milk_and_sugar = SugarDecorator(coffee_with_milk)
23   print(coffee_with_milk_and_sugar.cost())  # 8
```

10.2.5   Facade

Motivation.

Definition 10.11 (Facade Pattern)

In a **facade pattern**, it provides a simplified interface to a complex subsystem. It is characterized by
the following.

Results.

Example 10.11 (Case Study)

```
1    class CPU:
2        def freeze(self):
3            print("CPU: Freezing...")
4
5        def execute(self):
6            print("CPU: Executing...")
7
8    class Memory:
9        def load(self):
10           print("Memory: Loading data...")
11
12   class HardDrive:
13       def read(self):
14           print("HardDrive: Reading...")
15
16   class ComputerFacade:
17       def __init__(self):
18           self.cpu = CPU()
19           self.memory = Memory()
20           self.hard_drive = HardDrive()
21
22       def start(self):
```

```
23          print("Starting computer...")
24          self.cpu.freeze()
25          self.memory.load()
26          self.hard_drive.read()
27          self.cpu.execute()
28          print("Computer started!")
29
30  # Usage
31  computer = ComputerFacade()
32  computer.start()
33  # Simple interface hides complex subsystem interactions
```

10.2.6 Flyweight

Motivation.

Definition 10.12 (Flyweight Pattern)

In a **flyweight pattern**, we share some common state among multiple objects to save memory. It is characterized by the following.
   1.

Results.

Example 10.12 (Case Study)

```python
1   class TreeType:
2       def __init__(self, name, color, texture):
3           self.name = name
4           self.color = color
5           self.texture = texture
6
7       def draw(self, x, y):
8           print(f"Drawing {self.name} tree at ({x}, {y}) - {self.color}, {self.texture}")
9
10  class TreeFactory:
11      _tree_types = {}
12
13      @classmethod
14      def get_tree_type(cls, name, color, texture):
15          key = (name, color, texture)
16          if key not in cls._tree_types:
17              cls._tree_types[key] = TreeType(name, color, texture)
18              print(f"Creating new tree type: {name}")
19          return cls._tree_types[key]
20
21  class Tree:
22      def __init__(self, x, y, tree_type):
23          self.x = x
24          self.y = y
25          self.tree_type = tree_type
26
27      def draw(self):
28          self.tree_type.draw(self.x, self.y)
```

```
29
30  # Usage
31  factory = TreeFactory()
32  trees = []
33
34  # Reuses the same TreeType object for oak trees
35  trees.append(Tree(1, 2, factory.get_tree_type("Oak", "green", "rough")))
36  trees.append(Tree(3, 4, factory.get_tree_type("Oak", "green", "rough")))
37  trees.append(Tree(5, 6, factory.get_tree_type("Pine", "dark green", "smooth")))
38
39  for tree in trees:
40      tree.draw()
```

10.2.7 Proxy

Motivation.

Definition 10.13 (Proxy Pattern)

In a **proxy pattern**, we provide a placeholder or surrogate for another object to control access. It is characterized by the following.

1.

Results.

Example 10.13 (Case Study)

```python
1  class RealImage:
2      def __init__(self, filename):
3          self.filename = filename
4          self.load_from_disk()
5
6      def load_from_disk(self):
7          print(f"Loading image: {self.filename}")
8
9      def display(self):
10         print(f"Displaying image: {self.filename}")
11
12 class ProxyImage:
13     def __init__(self, filename):
14         self.filename = filename
15         self.real_image = None
16
17     def display(self):
18         if self.real_image is None:
19             self.real_image = RealImage(self.filename)
20         self.real_image.display()
21
22 # Usage
23 image1 = ProxyImage("photo1.jpg")
24 image2 = ProxyImage("photo2.jpg")
25
26 image1.display()  # Loads and displays
27 image1.display()  # Just displays (already loaded)
```

```
28    image2.display()   # Loads and displays
```

10.3   Behavioral Patterns

How objects communicate with each other.

10.3.1   Strategy

If you are coding some object that should have the following characteristics

$$a \in \{a_1, a_2, a_3\} = A \tag{8}$$
$$b \in \{b_1, b_2\} = B \tag{9}$$
$$c \in \{c_1, c_2, c_3, c_4\} = C \tag{10}$$

and which characteristics it has determines the implementation of the method, then you should use the strategy pattern.

Definition 10.14 (Strategy Pattern)

In a **strategy pattern**, you can use a family of interchangeable algorithms. It is characterized by the following.
  1. It defines a family of algorithms (different implementations of a set of method), encapsulates each one, and makes them interchangeable.
  2. This enables selecting an algorithm at runtime and let's them vary independently from clients that use it.[a]

  ---
  [a]This is basically object composition.

This pattern is good as it

  1. often reduces long lists of conditionals,

  2. avoids duplicate code,

  3. keeps class changes from forcing other class changes, and

  4. can hide complicated/secret code from the user

However, there may be an increased number of objects/classes.

Example 10.14 (Overwatch)

Say that you are redesigning Overwatch. We can characterize each hero in the following ways:
  1. *Composition*: poking, brawling, diving.[a] If they are a poker, then they will deal a little damage. If they are a brawler, they will deal medium damage. If they are a diver, they will deal high damage.
  2. *Ability to Fly*: Can fly or cannot fly. If they can fly, their z-coordinate will increase by 10 units. If they cannot fly, then nothing will happen.
  3. *Role*: Tank, DPS, Support. Tanks will have the ability to raise a shield for defense. DPS have the ability to dodge attacks. Supports have the ability to heal themselves.
We start off with a class Hero and then want three attributes:

```
1    from abc import ABC, abstractmethod
2
3    class Composition(ABC):
```

```
4    @abstractmethod
5    def attack(self, other: "Hero"): pass
6
7  class Poker(Composition):
8    def attack(self, other): print(f"5 damage dealt to {other}")
9
10 class Brawler(Composition):
11   def attack(self, other): print(f"10 damage dealt to {other}")
12
13 class Diver(Composition):
14   def attack(self, other): print(f"20 damage dealt to {other}")
15
16 class Fly(ABC):
17   @abstractmethod
18   def fly(self): pass
19
20 class CanFly(Fly):
21   def fly(self): print("Flying")
22
23 class CannotFly(Fly):
24   def fly(self): print("Cannot fly.")
25
26 class Role(ABC):
27   @abstractmethod
28   def defend(self): pass
29
30 class DPS(Role):
31   def defend(self): print("Dodging")
32
33 class Tank(Role):
34   def defend(self): print("Raising a shield")
35
36 class Support(Role):
37   def defend(self): print("Healed myself")
```

```
1  class Hero:
2
3    def __init__(self, comp, fly, role):
4      self.comp: Composition = comp
5      self.fly: Fly = fly
6      self.role: Role = role
7
8    def attack(self, other): return self.comp.attack(other)
9    def fly(self): return self.fly.fly()
10   def defend(self): return self.role.defend()
11
12
13 winston = Hero(Diver(), CannotFly(), Tank())
14 widowmaker = Hero(Poker(), CannotFly(), DPS())
15
16 winston.attack(widowmaker)
17 # 20 damage dealt to ...
```

---

[a]https://www.youtube.com/watch?v=93nrrdo-w0M

10.3.2   Observer

Motivation.

Definition 10.15 (Observer Pattern)

In an **observer pattern**, objects subscribe to events and get notified when they occur. Its is characterized by the following.

   1.

Results.

Example 10.15 (News Publisher)

```python
class NewsPublisher:
    def __init__(self):
        self.subscribers = []

    def subscribe(self, subscriber):
        self.subscribers.append(subscriber)

    def notify(self, news):
        for subscriber in self.subscribers:
            subscriber.update(news)

class EmailSubscriber:
    def update(self, news):
        print(f"Email sent: {news}")

class SMSSubscriber:
    def update(self, news):
        print(f"SMS sent: {news}")

# Usage
publisher = NewsPublisher()
publisher.subscribe(EmailSubscriber())
publisher.subscribe(SMSSubscriber())
publisher.notify("Breaking news!")
```

which outputs

```
Email sent: Breaking news!
SMS sent: Breaking news!
```

10.3.3   Command

Motivation.

Definition 10.16 (Command Pattern)

In a **command pattern**, we encapsulate requests as objects, allowing parameterization and queuing.

Results.

Example 10.16 (Remote Control)

```python
class Light:
    def turn_on(self):
        print("Light is ON")

    def turn_off(self):
        print("Light is OFF")

class TurnOnCommand:
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_on()

class TurnOffCommand:
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_off()

class RemoteControl:
    def __init__(self):
        self.command = None

    def set_command(self, command):
        self.command = command

    def press_button(self):
        self.command.execute()

# Usage
light = Light()
remote = RemoteControl()

remote.set_command(TurnOnCommand(light))
remote.press_button()  # Light is ON

remote.set_command(TurnOffCommand(light))
remote.press_button()  # Light is OFF
```

10.3.4   Template Method

Motivation.

Definition 10.17 (Template Pattern)

In a **template pattern**, we define the skeleton of an algorithm and let subclasses override specific steps.

    1.

Results.

Example 10.17 (Data Parsing)

```python
from abc import ABC, abstractmethod

class DataParser(ABC):
    def parse(self):
        self.open_file()
        self.extract_data()
        self.parse_data()
        self.close_file()

    def open_file(self):
        print("Opening file...")

    @abstractmethod
    def extract_data(self):
        pass

    @abstractmethod
    def parse_data(self):
        pass

    def close_file(self):
        print("Closing file...")

class CSVParser(DataParser):
    def extract_data(self):
        print("Extracting CSV data...")

    def parse_data(self):
        print("Parsing CSV format...")

class JSONParser(DataParser):
    def extract_data(self):
        print("Extracting JSON data...")

    def parse_data(self):
        print("Parsing JSON format...")

# Usage
csv_parser = CSVParser()
csv_parser.parse()
```

10.3.5 Iterator

Motivation.

Definition 10.18 (Iterator Pattern)

In an **iterator pattern**, we provide a way to access elements sequentially without exposing underlying representations. It is characterized by the following.
   1.

Results.

Example 10.18 (Case Study)

```python
class BookCollection:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def __iter__(self):
        return BookIterator(self.books)

class BookIterator:
    def __init__(self, books):
        self.books = books
        self.index = 0

    def __next__(self):
        if self.index < len(self.books):
            book = self.books[self.index]
            self.index += 1
            return book
        raise StopIteration

# Usage
collection = BookCollection()
collection.add_book("Python 101")
collection.add_book("Design Patterns")
collection.add_book("Clean Code")

for book in collection:
    print(book)
```

10.3.6 Chain of Responsibility

Motivation.

Definition 10.19 (Chain of Responsibility)

In a **chain of responsibility pattern**, we pass requests along a chain of handlers until one handles it.

Results.

Example 10.19 (Case Study)

```python
class SupportHandler:
    def __init__(self):
        self.next_handler = None

    def set_next(self, handler):
        self.next_handler = handler
        return handler

```

```
9      def handle(self, request):
10          if self.next_handler:
11              return self.next_handler.handle(request)
12          return None
13
14  class Level1Support(SupportHandler):
15      def handle(self, request):
16          if request == "basic":
17              return "Level 1: Handling basic issue"
18          return super().handle(request)
19
20  class Level2Support(SupportHandler):
21      def handle(self, request):
22          if request == "intermediate":
23              return "Level 2: Handling intermediate issue"
24          return super().handle(request)
25
26  class Level3Support(SupportHandler):
27      def handle(self, request):
28          if request == "complex":
29              return "Level 3: Handling complex issue"
30          return super().handle(request)
31
32  # Usage
33  l1 = Level1Support()
34  l2 = Level2Support()
35  l3 = Level3Support()
36
37  l1.set_next(l2).set_next(l3)
38
39  print(l1.handle("basic"))         # Level 1: Handling basic issue
40  print(l1.handle("intermediate"))  # Level 2: Handling intermediate issue
41  print(l1.handle("complex"))       # Level 3: Handling complex issue
```

10.3.7   Interpreter

Motivation.

Definition 10.20 (Interpreter Pattern)

In an **interpreter pattern**, it defines a grammar and interprets sentences in that language. It is characterized by the following.
   1.

Results.

Example 10.20 (Case Study)

```
1  class Context:
2      def __init__(self):
3          self.variables = {}
4
5      def set_variable(self, name, value):
6          self.variables[name] = value
```

```
 7
 8      def get_variable(self, name):
 9          return self.variables.get(name, 0)
10
11  class Expression:
12      def interpret(self, context):
13          pass
14
15  class Number(Expression):
16      def __init__(self, value):
17          self.value = value
18
19      def interpret(self, context):
20          return self.value
21
22  class Variable(Expression):
23      def __init__(self, name):
24          self.name = name
25
26      def interpret(self, context):
27          return context.get_variable(self.name)
28
29  class Add(Expression):
30      def __init__(self, left, right):
31          self.left = left
32          self.right = right
33
34      def interpret(self, context):
35          return self.left.interpret(context) + self.right.interpret(context)
36
37  # Usage
38  context = Context()
39  context.set_variable("x", 10)
40  context.set_variable("y", 5)
41
42  # Expression: x + y + 3
43  expression = Add(Add(Variable("x"), Variable("y")), Number(3))
44  print(expression.interpret(context))  # 18
```

10.3.8   Mediator

Motivation.

Definition 10.21 (Mediator Pattern)

In a **mediator pattern**, we define an object that coordinates communication between other objects. It is characterized by the following.
  1.

Results.

Example 10.21 (Case Study)

```python
class ChatMediator:
    def __init__(self):
        self.users = []

    def add_user(self, user):
        self.users.append(user)

    def send_message(self, message, sender):
        for user in self.users:
            if user != sender:
                user.receive(message, sender)

class User:
    def __init__(self, name, mediator):
        self.name = name
        self.mediator = mediator
        mediator.add_user(self)

    def send(self, message):
        print(f"{self.name} sends: {message}")
        self.mediator.send_message(message, self)

    def receive(self, message, sender):
        print(f"{self.name} receives from {sender.name}: {message}")

# Usage
mediator = ChatMediator()

alice = User("Alice", mediator)
bob = User("Bob", mediator)
charlie = User("Charlie", mediator)

alice.send("Hello everyone!")
# Bob receives from Alice: Hello everyone!
# Charlie receives from Alice: Hello everyone!
```

10.3.9 Memento

Motivation.

Definition 10.22 (Memento)

In a **memento pattern**, it captures and restores an objects internal state without violating encapsulation.

Results.

Example 10.22 (Case Study)

```python
class TextEditor:
    def __init__(self):
        self.content = ""
```

```
 4
 5      def type(self, text):
 6          self.content += text
 7
 8      def get_content(self):
 9          return self.content
10
11      def save(self):
12          return TextMemento(self.content)
13
14      def restore(self, memento):
15          self.content = memento.get_state()
16
17  class TextMemento:
18      def __init__(self, state):
19          self._state = state
20
21      def get_state(self):
22          return self._state
23
24  class History:
25      def __init__(self):
26          self.mementos = []
27
28      def push(self, memento):
29          self.mementos.append(memento)
30
31      def pop(self):
32          if self.mementos:
33              return self.mementos.pop()
34          return None
35
36  # Usage
37  editor = TextEditor()
38  history = History()
39
40  editor.type("Hello ")
41  history.push(editor.save())
42
43  editor.type("World!")
44  history.push(editor.save())
45
46  editor.type(" How are you?")
47  print(editor.get_content())  # Hello World! How are you?
48
49  # Undo
50  editor.restore(history.pop())
51  print(editor.get_content())  # Hello World!
52
53  # Undo again
54  editor.restore(history.pop())
55  print(editor.get_content())  # Hello
```

10.3.10   Visitor

Motivation.

Definition 10.23 (Visitor Pattern)

In a **visitor pattern**, it separates algorithms from the objects they operate on.

Results.

Example 10.23 (Case Study)

```python
class Element:
    def accept(self, visitor):
        pass

class Book(Element):
    def __init__(self, title, price):
        self.title = title
        self.price = price

    def accept(self, visitor):
        visitor.visit_book(self)

class Electronics(Element):
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def accept(self, visitor):
        visitor.visit_electronics(self)

class Visitor:
    def visit_book(self, book):
        pass

    def visit_electronics(self, electronics):
        pass

class PriceCalculator(Visitor):
    def __init__(self):
        self.total = 0

    def visit_book(self, book):
        print(f"Book '{book.title}': ${book.price}")
        self.total += book.price

    def visit_electronics(self, electronics):
        print(f"Electronics '{electronics.name}': ${electronics.price}")
        self.total += electronics.price

# Usage
items = [
    Book("Design Patterns", 45),
    Electronics("Laptop", 1200),
    Book("Clean Code", 40)
]

calculator = PriceCalculator()
for item in items:
```

```
49        item.accept(calculator)
50
51   print(f"Total: ${calculator.total}")
```

10.3.11   State

Motivation.

Definition 10.24 (State Pattern)

In a **state pattern**, we can have an object change its behavior when its internal state changes. It is characterized by the following.
  1. It reduces the need for conditional statements depending on some attribute value.[a]
  2. This is very similar to strategy, and sometimes interchangeable, depending on who you ask.

---
  [a]Known as "switch hell."

Example 10.24 (Coin Machine)

```python
1    from abc import ABC, abstractmethod
2
3    # State interface
4    class State(ABC):
5        @abstractmethod
6        def insert_coin(self, machine):
7            pass
8
9        @abstractmethod
10       def eject_coin(self, machine):
11           pass
12
13       @abstractmethod
14       def dispense(self, machine):
15           pass
16
17   # Concrete States
18   class NoCoinState(State):
19       def insert_coin(self, machine):
20           print("Coin inserted")
21           machine.set_state(machine.has_coin_state)
22
23       def eject_coin(self, machine):
24           print("No coin to eject")
25
26       def dispense(self, machine):
27           print("Insert coin first")
28
29   class HasCoinState(State):
30       def insert_coin(self, machine):
31           print("Coin already inserted")
32
33       def eject_coin(self, machine):
34           print("Coin ejected")
35           machine.set_state(machine.no_coin_state)
```

```
36
37      def dispense(self, machine):
38          print("Dispensing item...")
39          machine.set_state(machine.sold_state)
40
41  class SoldState(State):
42      def insert_coin(self, machine):
43          print("Please wait, dispensing item")
44
45      def eject_coin(self, machine):
46          print("Cannot eject, already dispensing")
47
48      def dispense(self, machine):
49          print("Item dispensed!")
50          machine.set_state(machine.no_coin_state)
51
52  # Context
53  class VendingMachine:
54      def __init__(self):
55          self.no_coin_state = NoCoinState()
56          self.has_coin_state = HasCoinState()
57          self.sold_state = SoldState()
58
59          self.current_state = self.no_coin_state
60
61      def set_state(self, state):
62          self.current_state = state
63
64      def insert_coin(self):
65          self.current_state.insert_coin(self)
66
67      def eject_coin(self):
68          self.current_state.eject_coin(self)
69
70      def dispense(self):
71          self.current_state.dispense(self)
72
73  # Usage
74  machine = VendingMachine()
75
76  machine.insert_coin()  # Coin inserted
77  machine.dispense()     # Dispensing item...
78  machine.dispense()     # Item dispensed!
79
80  machine.dispense()     # Insert coin first
81  machine.eject_coin()   # No coin to eject
82
83  machine.insert_coin()  # Coin inserted
84  machine.eject_coin()   # Coin ejected
```

11   Hardware

11.1   GPU

Look at this article[13] for a thorough high-level explanation of how GPUs work. Here are some notes I took for things to consider when buying or using a GPU, in order of importance.

1. *Memory.* Also known as the VRAM, or "global memory." If you can't fit your model in your GPU, then forget about training it or doing inference. So this should be the first thing to consider. Plus, higher memory allows you to increase your batch size.

2. *Memory Bandwidth.* This is the internal bandwidth from the GPU memory (VRAM) to the actual cores/registers. According to the article, this is pretty much the single biggest bottleneck for GPU speed. For diagnosis, if there is high memory access (wandb: time spent accessing memory) and low GPU utilization (wandb: GPU Utilization), then, there is a memory bottleneck, and you need higher throughput.

3. *Tensor Cores.* The tensor cores is what gives Nvidia their edge over AMD. I've also done a quick test and epoch time is roughly linearly proportional to cuda/tensor cores, for smaller batch sizes. These allow you to do matrix multiplication fast in just 1 cycle.

The next few are things that I might think are important.

1. *PCIe Bandwidth.* This is the bandwidth between the GPU and the CPU. This should be important (?) but is stated not to be in the article. More benchmarking will need to be done.

2. *Cuda Cores.* This seems to also have strong correlation with performance and is the first metric that most users look at.

3. *FLOPs.* FP32 performance is prob most important. This seems to be more of a "summary statistic."

The next ones are properties that are not as important for me.

1. *NVLink.* Basically the speed between GPUs for multi GPU.

2. *Architecture.* If you are looking at the basic stat of the GPU, then the architecture may be irrelevant, minus a few technical upgrades for nuanced situations.

3. *Ray Tracing (RT) Cores.* These are only relevant for graphics rendering. Not relevant for training unless you're working with models that do graphics.

4. *Power Consumption.* Usually these aren't on your machine, so doens't matter how much power you're using on the cluster.

I should prob do studies on this. Ideally, make a script that generates a report on this for every new machine.

11.2   Nvidia Architectures

---

[13]https://timdettmers.com/2023/01/30/which-gpu-for-deep-learning/