

Development Tools

Muchang Bahng

Fall 2024

Contents

1	Text Editing with Neovim	2
1.1	Configuration Files	2
1.2	Troubleshooting	3
1.3	Language Service Providers	4
1.4	Snippets	4
2	Documentation with LaTeX	5
3	Version Control with Git	6
3.1	Local Git Repository	6
3.2	Conflicts	7
3.3	Interactive Rebasing	8
3.4	Branches	10
	3.4.1 Working Between Branches	11
	3.4.2 Integrating Branches	11
3.5	Remote Trees	14
3.6	Reflog	18
3.7	Pull Requests and Forking	19
4	Continuous Integration (CI) with Git Actions and Docker	20
5	Unittesting	21
6	Package Management	22
6.1	Pip	22
6.2	Conda	22
6.3	Using Pip with Conda	24
6.4	Mamba	26

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 plummeted 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 summary 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 right, 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. One can go to Overleaf and have everything preconfigured, but in here I set it up on my local desktop. I will already assume you have a PDF viewer installed. I use zathura, which is lightweight and also comes with vim motions for navigation.

First install the VimTeX plugin in `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.

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 “`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/local/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.

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 .

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
```

^aSo nodes actually point to *previous nodes*.

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

$$(A) \leftarrow (B) \leftarrow (C) \leftarrow (H \mapsto D) \quad (1)$$

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

$$(A) \leftarrow (H \mapsto B) \quad (2)$$

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 \gg , $==$, \ll 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

$$(C_1) \leftarrow (C_2) \leftarrow (C_3) \leftarrow (H \mapsto C_4) \quad (3)$$

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

$$(C_1) \leftarrow (C_2) \leftarrow (C_3) \leftarrow (C_4) \leftarrow (C'_2) \quad (4)$$

So really, we are “deleting” our history by adding.

Example 3.1 (Conflicts in Reverting)

Say that you have history

$$(C_1) \leftarrow (C_2) \leftarrow (C_3) \leftarrow (H \mapsto C_4) \quad (5)$$

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

$$(A) \leftarrow (B) \leftarrow (C) \leftarrow (D) \quad (6)$$

to something like

$$(A) \leftarrow (B) \leftarrow (D) \quad (7)$$

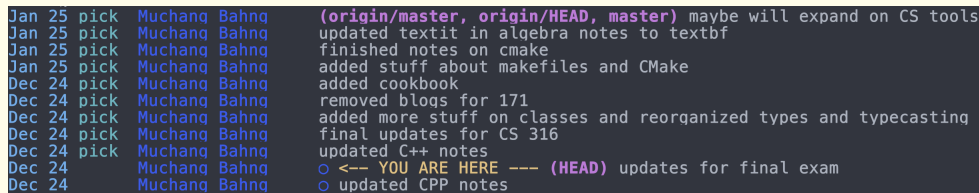
Definition 3.9 (Rebasing)

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.



```
Jan 25 pick Muchang Bahng (origin/master, origin/HEAD, master) maybe will expand on CS tools
Jan 25 pick Muchang Bahng updated textit in algebra notes to textbf
Jan 25 pick Muchang Bahng finished notes on cmake
Jan 25 pick Muchang Bahng added stuff about makefiles and CMake
Dec 24 pick Muchang Bahng added cookbook
Dec 24 pick Muchang Bahng removed blogs for 171
Dec 24 pick Muchang Bahng added more stuff on classes and reorganized types and typecasting
Dec 24 pick Muchang Bahng final updates for CS 316
Dec 24 pick Muchang Bahng updated C++ notes
Dec 24 pick Muchang Bahng o <-- YOU ARE HERE --- (HEAD) updates for final exam
Dec 24 pick Muchang Bahng o updated CPP notes
```

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.

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

^aNote 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.

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.

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.

^aThough patching really just does an interactive rebase in the backend.

```
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**.

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**.

^aWhen we get to forking, we will talk about multiple remote repositories.

^bAgain, for now it's really the only remote.

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>	1 git pull origin main
2	2
3 git fetch <remote-url> <branch>	3 git fetch origin main
4 git merge <remote-url/branch>	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.

```
1 $ git reflog
2 ab12345 HEAD@{0}: reset: moving to HEAD~1    # Most recent action
```

```
3 bc23456 HEAD@{1}: commit: Add feature X
4 cd34567 HEAD@{2}: rebase: onto main      # Rebase happened here
5 ef45678 HEAD@{3}: commit: Initial commit
```

3.7 Pull Requests and Forking

4 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.

5 Unittesting

6 Package Management

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

6.1 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.¹ 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*.² He also created a virtual environment manager, called *virtualenv*, or *venv*.

Example 6.1 (Managing VirtualEnvs)

Here are some useful commands.

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

Example 6.2 (Manage Packages Inside Venvs)

Here are some useful commands.

```
1 > pip list                        # list all packages
2 > pip install xxx                 # install package
3 > pip install xxx==0.0.1          # install version of package
4 > pip uninstall xxx               # uninstall package
```

Some notes:

1. Running `pip install package_name` 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 dependencies! 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.

6.2 Conda

While `pip` was great for managing Python packages, the main problem was that they were all focused around python, neglecting non-Python library dependencies, 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).

¹Really anybody can upload their own, so many packages may contain malware.

²From several suggestions the creator received on his blog post, and it is a recursive acronym 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 6.3 (Manage Conda Envs)

```
1 > conda env list                # list all environments
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
5 > conda env export > environment.yaml    # export conda environment to yaml
6 > conda create -f environment.yaml    # make conda env from yaml file
```

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 6.4 (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 6.5 (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 package=0.0.1 # install specific version of
   package
3 > conda install -c channel package # install package from channel
4 > conda uninstall package # uninstall package with
   dependencies
5 > conda uninstall package --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.

6.3 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 6.6 (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` columns from now on.

<pre> 1 > pip list 2 Package Version 3 ----- 4 pip 25.0.1 </pre>	<pre> 1 > conda list 2 # packages in environment at /opt/miniconda3/envs/test: 3 # 4 # Name Version Build Channel </pre>
--	---

Example 6.7 (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.

<pre> 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 . </pre>	<pre> 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 ... </pre>
--	---

Example 6.8 (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 6.9 (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	1	# packages in environment at /opt/miniconda3/envs/test:
2	-----	-----	2	# Name Version
3	numpy	2.2.0	3	numpy 1.26.4
4	pip	23.2.1	4	pip 25.0.1
5	scipy	1.12.0	5	python 3.12.9
6	.		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 6.1 (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 virtualenvs 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.

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.

6.4 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.

^aThough most widely used packages are pretty good at making sure that there are no incompatibilities.

^bFor example, installing `pandas=1.1.4` works on conda but not with pip.

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).