CUBE (Coding Unafraid for Baskin Engineering)
First, set up your space

1 The Command Line

One of the starting points for getting comfortable as a programmer is using the *command line*. This means that you work by opening a terminal and doing all your work (navigating files, writing code, compiling and running code, seeing your output) through the terminal. Everything is done by typing commands at the terminal prompt, and you do not use your mouse at all!

It takes some getting used to, so liberally use Google to look up various commands (hey, I forget the syntax for various commands all the time). The command line gives access to powerful tools in Unix (grep, find, wc, vim) that make programming much easier in the long term. Any time you see/hear about a command and it doesn't make sense, Google the command. You will see many hits, where you can read up on how to use the command. You should get into the habit of searching online for Unix commands and reading about them. Alas, no course teaches students about programming tools, but there are very important for being successful.

You will begin by creating a directory for all the work you do in the course. Again, use the command line (command `mkdir`).

Writing Code: For writing code, use command line editors. There are many of them (`nano`, `vim`, `emacs`, etc.). `nano` is the simplest of them all, but it's too rudimentary for coding. I would highly recommend `vim`: it is powerful and fast for writing code (and other things). Here are a few tutorials:

- `https://danielmiessler.com/study/vim/`
- `https://linuxconfig.org/vim-tutorial`
- Vim cheatsheet: `https://vim.rtorr.com/`

I don't recommend reading/watching hours of tutorials before starting. Start reading a bit, enough to know how to open a file and type in it. Then start writing code, see how you do, read a bit more, rinse and repeat. You will need to tinker around, and this will build the habit of *experimenting with tools.* One of big weaknesses of many students is the need for someone else to tell them the solution and the fear of tinkering. The more you use various tools of this kind, the better you get at finding your own solutions.

2 Git ready

`git` is an industry standard for collaborating on code, and it's (yet another) powerful tool that makes your life easy. (After you get through the hard work of learning it.) While we will not need the entire power of git, this summer is a good testbed to get your feet wet with git.

There's a lot to say about it, but let me try to distil it down. The key concept is that of a *repository*. Think of this a folder with a bunch of files in it, that is stored in a git server (hosted by github). When you *clone* a repository, you create a local copy of the repository, which is stored in your computer. You can make changes to your local copy and treat them as an local file. You can then *push* the changes to the server. When you want to get changes from the server to your local copy, you *pull* from the server. This way, multiple people can work on the same repository, and the git system manages by merging all the changes into one consistent repository.

I have created a repo which contains files/instructions/code snippets that you will find useful. You should clone it by running the command:

```
git clone https://github.com/csesha/CUBE <DIR NAME>
```

This will create a directory `<DIR NAME>` in your current directory. In that directory, when you run the command `git pull`, you will get the latest version. You are not a collaborator in that repo, so you cannot push changes into it.

Go to your github page and create a repository for your own code. Clone that repository in a *separate* directory (don't confuse the two repos!). This is where you will write your code, and where you can experiment with git. A few basic git commands.

- `git pull:` Before making changes, always run this command. This makes your local copy have all the changes from the server. Sometimes, you may have modified a file that has also been changed at the server. In this case, git will ask you to commit your changes first.
- `git add <FILENAME>:` Suppose you changed `foo.py`. This command says that you want to add your changed file to the repository. It only sets the stage; it does not actually change the repository. You can add multiple changes.
- `git commit -m <COMMIT MESSAGE>:` You made changes, added them, and want to change the server repository. This command commits the changes to your *local copy* (again, the server copy does not change). Always write a useful commit message, just so that you can track these later on if needed.
- `git push:` Finally, this command pushes the changes to the server. If there were changes in the server since you last pulled, *git will not let you push!* You will need to pull first, in which case, git will try to *merge* the server changes with your changes. Usually, it will do it automatically, but sometimes the git system is unable to the merge. This creates a *merge conflict*, that you will need to manually fix. The latter is a more advanced concept, but one that you will need to deal with when you're a frequent git user.

How should you learn about all of this? Tinker and experiment! Start another git repo, make a friend a collaborator, and then try to collaborate on a single file. You will see many of these concepts in action and will recognize the power of git.

You can also see the git repo on your browser. The URL `https://github.com/csesha/CUBE` will lead you to a page with the CUBE repo. Since it is public, anyone can access the repo and look at its contents.

3 Get some color in your life (and file)

Writing code and navigating directories are a lot easier when things are color coded. For that, you will need to modify special "config" files made for bash (your shell/terminal) and for vim. Usually, instead of writing these ourselves, we usually steal them from someone. In your case, you steal from me.

Once you've clone my repo, you will see a directory called 'Setup'. In that are two files `.vimrc` and `.bash_profile`. Put both those files in your home directory. Careful! Because these filenames start with a dot, Unix recognizes them as config files and hides them when you run `ls`. But if you run the command `ls -a`, you will see it. (Or even better, run `ls -al` and see you what get. Do you understand what the output is?)

You can move the `.vimrc` file to the right directory by running the command:

```
cp .vimrc ~
```

Here, ~ is a shortcut for your home directory.

4 Know your way around

Some basic Unix commands you need to navigate and manage your files: `cd, cp, mv, mkdir, rm, ls, chmod`.

Some other very useful commands: `cat, less, more, grep, find, wc, du, diff`.

The best way to get familiar is to use the terminal (as much as possible) for your tasks. Learn to interpret the output of these commands by running them. And continuously tinker, experiment, and fail! See if you can perform these tasks using the Unix commands.

- Get the number of lines in a file.
- Determine the first line number where two files are different.
- Search for all files in a directory that contain a specific word.
- Find all the files with the extension '.py'.
- Find all the files in a directory with size more than 1MB.
- Get all the words in a file that are 8 letters long where the fifth letter is either e or E.