CS 45, Lecture 5

Text Editors

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Spring 2023

Outline

Contents

1 Text Editing: An Overview

1.1 Rich Text

When we think about editing a document, we usually think of doing that in a *rich text editor*, something like Word or Google Docs.

Rich text is for *humans communicating with humans*– its elements are structured around elements of prose, such as words, paragraphs, headings, etc., and its features are centered around making consuming written text easier for humans– things like varying fonts, emphasizing text with bold, italic, or underline, the ability to insert pictures or other multimedia, and so on and so forth.

However, while this information is helpful and sometimes really necessary in human-to-human communication, it's unnecessary and gets in the way when we're desiring to communicate with a computer (or give it instructions). This is why we use *plain text* for computers!

1.2 Plain Text Editors

Lots of different kinds of programs have been developed to edit plain text– in fact, it's really one of the core affordances a computer offers. Some plain text editors, like Windows' Notepad or macOS' TextEdit are extremely basic, and fulfill the mantle of a plain text editor with no frills. However, knowing how we frequently use plain text to communicate, configure, and program computers, many more plain text editors integrate additional tools to make these tasks easier.

Computer programs in general tend to fall into one of three categories:

1. *GUI (Graphical User Interface) programs.* These are programs that display a graphical interface in some way, and are the kinds of applications you're almost certainly most familiar with.
2. *TUI (Text User Interface) programs.* These are programs that display a sort of imitation of a graphical interface using text within a terminal. These are distinct from CLI programs in that the application stays open and allows for continued operation, taking over the terminal and designed for interaction directly with the user via the keyboard and sometimes mouse.
3. *CLI (Command Line Interface) programs.* These are programs that are usable only from the command line, by invoking the program with a set of flags and arguments, and potentially information from standard input.

As a caveat, TUI applications especially are often fairly inaccessible to screen readers; since they just display information– including UI– as a bunch of text that isn't distinguished or hierarchical in any way.

1.3 Learning a new editor

No matter what kind of editor you choose to jump into learning, there's going to be a learning curve. Our recommendation is to *choose one GUI editor and one TUI editor*, then stick with them for a while. We'd estimate that you'll reach the same speed as you'd use any other editor after about 10-20 hours of use, and often will be much faster than others after about 20 hours of use.

Don't be afraid to look some stuff up, too– often, there's a faster way to go about doing things that's just a google search away!

Since we expect that you all are pretty familiar with GUI editors (such as PyCharm), we'll jump into doing a short demo with `vim`.

1.4 Why `vim`? Or a TUI editor at all?

The biggest reason is for *remote editing.* Computers whose purpose is to serve content or provide resources– i.e. servers– often do not have GUIs installed at all, as they are a significant resource use overhead that has no good purpose when CLI and TUI applications exist that don't require all that overhead. Thus, it's a good idea to get used to some editor you can use via only a terminal.

2 Vim

2.1 A Quick History

`vim` was inspired by and spun off of an editor called `vi`, and stands for VI iMitation (or VI iMproved, depending on who you ask). `vi` was one of the first TUI editors, based on the line-editor `ed` (and also the `vi`sual mode of a CLI tool called `ex`), which required you to edit line by line using certain commands.

`vi`, and `vim`, continue to use that idea of *commands and modes*!

2.2 A Modal Editor

`vim` *uses different "modes" to control editing.*

1. You always start in *normal mode*, used for navigating around the file
2. You press `i` to enter *insert mode*, to write new text
3. You press `R` to enter *replace mode*, to overwrite text
4. You press `v` to enter *visual mode*, for copying or deleting lines or blocks of text at a time
5. You press `:` to enter *command mode*, which allows you to do all sorts of things (like save, quit, find-replace, etc.).

2.3 Learning to Navigate `vim`

Example 2.1 (codeblock). We did a demo in class. You can access the demo using the commands below:

(Curious what these commands do? We introduced a handy-dandy website called explainshell.com that can break down commands for you using text from the `man` pages. In the case of the commands above, the first one (`curl`) downloads a file, and the second instructs vim to open the file that was downloaded.)

You should open the file in `vim` and try navigating through it; we did this exercise for 15 or so minutes live in class.

2.4 Windows & Buffers

We're all used to the idea that each window is responsible for one file. If you want to open a new file in another window, you can do that.

`vim` plays with this idea a little bit. *Buffers* refer to open files, and are an abstract concept. A single buffer may be open in one *or more* windows!

Meanwhile, *windows* are "views" into a buffer. This means you could have *multiple windows open to the SAME buffer!*– if you do this, that means your changes to the buffer in one window would instantly reflect in the other. This is useful if e.g. you want to scroll to multiple different parts of a file at once.

`:q` always closes the current window. You can also "split" a window using the `:sp` ("split") command, or vertically with the `:vsp` command.

2.5 Configuring `vim`

You can customize your installation of `vim` by writing a `.vimrc` file in your home directory (i.e. `/.vimrc`).

`.vimrc` files contain a list of commands that run when you start `vim`. For example, mine makes the mouse work, adds line numbers, and makes backspace and the arrow keys work in a manner I'd expect from an editor.

You can also add 3rd-party plugins to `vim`, either manually or using a plugin manager like `vundle`.

2.6 Demoing `.vimrc`

Example 2.2 (codeblock). We did a demo in class. You can access the demo using the commands below:

(The above command will open a temporary copy of the file at that URL, without saving it permanently.)

The above `.vimrc` is a slightly simpler version of the one I use, commented so you can see what each part does. It is based on a `.vimrc` file that was distributed as part of CS107 back when they used vim.

3 Visual Studio Code

Visual Studio Code is our IDE of choice, as it stands right now!

3.1 What's an IDE?

An IDE, or *Integrated Developer Environment*, is an application for software development and software code editing that bundles together *lots of functionality for developer productivity into one place.*

In particular, this usually means bundling code editing tools together with syntax highlighting and smart autocomplete, as well as error checking, build tools, testing tools, the ability to run code, and some other tools that scan and index your code automatically to help you understand and navigate it quickly, all in one tool.

3.2 Why VSCode?

We like VSCode for a couple reasons; besides the fact that it is totally free, it also has *plug-and-play language support* (you can make it richly support new languages by just installing a plugin to it)– and there are a *lot* of available language plugins that are very good for VSCode.

Another key reason is that VSCode offers strong support for *remote editing*, which means that you can access and edit resources *on a server*, without needing a GUI shell to be installed on the server at all.

3.3 VSCode Demo

We did a demo in class, demonstrating some of the features of an IDE, the ability to install plugins for language support, and showing off remote editing.