Editors (Vim) · the missing semester of your cs education

missing.csail.mit.edu/2020/editors

Editors (Vim)

Writing English words and writing code are very different activities. When programming, you spend more time switching files, reading, navigating, and editing code compared to writing a long stream. It makes sense that there are different types of programs for writing English words versus code (e.g. Microsoft Word versus Visual Studio Code).

As programmers, we spend most of our time editing code, so it's worth investing time mastering an editor that fits your needs. Here's how you learn a new editor:

- Start with a tutorial (i.e. this lecture, plus resources that we point out)
- Stick with using the editor for all your text editing needs (even if it slows you down initially)
- Look things up as you go: if it seems like there should be a better way to do something, there probably is

If you follow the above method, fully committing to using the new program for all text editing purposes, the timeline for learning a sophisticated text editor looks like this. In an hour or two, you'll learn basic editor functions such as opening and editing files, save/quit, and navigating buffers. Once you're 20 hours in, you should be as fast as you were with your old editor. After that, the benefits start: you will have enough knowledge and muscle memory that using the new editor saves you time. Modern text editors are fancy and powerful tools, so the learning never stops: you'll get even faster as you learn more.

Which editor to learn?

Programmers have strong opinions about their text editors.

Which editors are popular today? See this Stack Overflow survey (there may be some bias because Stack Overflow users may not be representative of programmers as a whole). Visual Studio Code is the most popular editor. Vim is the most popular command-line-based editor.

Vim

All the instructors of this class use Vim as their editor. Vim has a rich history; it originated from the Vi editor (1976), and it's still being developed today. Vim has some really neat ideas behind it, and for this reason, lots of tools support a Vim emulation mode (for

example, 1.4 million people have installed Vim emulation for VS code). Vim is probably worth learning even if you finally end up switching to some other text editor.

It's not possible to teach all of Vim's functionality in 50 minutes, so we're going to focus on explaining the philosophy of Vim, teaching you the basics, showing you some of the more advanced functionality, and giving you the resources to master the tool.

Philosophy of Vim

When programming, you spend most of your time reading/editing, not writing. For this reason, Vim is a *modal* editor: it has different modes for inserting text vs manipulating text. Vim is programmable (with Vimscript and also other languages like Python), and Vim's interface itself is a programming language: keystrokes (with mnemonic names) are commands, and these commands are composable. Vim avoids the use of the mouse, because it's too slow; Vim even avoids using the arrow keys because it requires too much movement.

The end result is an editor that can match the speed at which you think.

Modal editing

Vim's design is based on the idea that a lot of programmer time is spent reading, navigating, and making small edits, as opposed to writing long streams of text. For this reason, Vim has multiple operating modes.

- **Normal**: for moving around a file and making edits
- **Insert**: for inserting text
- **Replace**: for replacing text
- **Visual** (plain, line, or block): for selecting blocks of text
- **Command-line**: for running a command

Keystrokes have different meanings in different operating modes. For example, the letter `x` in Insert mode will just insert a literal character 'x', but in Normal mode, it will delete the character under the cursor, and in Visual mode, it will delete the selection.

In its default configuration, Vim shows the current mode in the bottom left. The initial/default mode is Normal mode. You'll generally spend most of your time between Normal mode and Insert mode.

You change modes by pressing `<ESC>` (the escape key) to switch from any mode back to Normal mode. From Normal mode, enter Insert mode with `i`, Replace mode with `R`, Visual mode with `v`, Visual Line mode with `V`, Visual Block mode with `<C-v>` (Ctrl-V, sometimes also written `^V`), and Command-line mode with `:`.

You use the `<ESC>` key a lot when using Vim: consider remapping Caps Lock to Escape (macOS instructions).

Basics

Inserting text

From Normal mode, press `i` to enter Insert mode. Now, Vim behaves like any other text editor, until you press `<ESC>` to return to Normal mode. This, along with the basics explained above, are all you need to start editing files using Vim (though not particularly efficiently, if you're spending all your time editing from Insert mode).

Buffers, tabs, and windows

Vim maintains a set of open files, called "buffers". A Vim session has a number of tabs, each of which has a number of windows (split panes). Each window shows a single buffer. Unlike other programs you are familiar with, like web browsers, there is not a 1-to-1 correspondence between buffers and windows; windows are merely views. A given buffer may be open in *multiple* windows, even within the same tab. This can be quite handy, for example, to view two different parts of a file at the same time.

By default, Vim opens with a single tab, which contains a single window.

Command-line

Command mode can be entered by typing `:` in Normal mode. Your cursor will jump to the command line at the bottom of the screen upon pressing `:`. This mode has many functionalities, including opening, saving, and closing files, and quitting Vim.

- `:q` quit (close window)
- `:w` save ("write")
- `:wq` save and quit
- `:e {name of file}` open file for editing
- `:ls` show open buffers
- `:help {topic}` open help
  - `:help :w` opens help for the `:w` command
  - `:help w` opens help for the `w` movement

Vim's interface is a programming language

The most important idea in Vim is that Vim's interface itself is a programming language. Keystrokes (with mnemonic names) are commands, and these commands *compose*. This enables efficient movement and edits, especially once the commands become muscle memory.

Movement

You should spend most of your time in Normal mode, using movement commands to navigate the buffer. Movements in Vim are also called “nouns”, because they refer to chunks of text.

- Basic movement: `hjkl` (left, down, up, right)
- Words: `w` (next word), `b` (beginning of word), `e` (end of word)
- Lines: `0` (beginning of line), `^` (first non-blank character), `$` (end of line)
- Screen: `H` (top of screen), `M` (middle of screen), `L` (bottom of screen)
- Scroll: `Ctrl-u` (up), `Ctrl-d` (down)
- File: `gg` (beginning of file), `G` (end of file)
- Line numbers: `:{number}<CR>` or `{number}G` (line {number})
- Misc: `%` (corresponding item)
- Find: `f{character}`, `t{character}`, `F{character}`, `T{character}`
    - find/to forward/backward {character} on the current line
    - `,` / `;` for navigating matches
- Search: `/{regex}`, `n` / `N` for navigating matches

Selection

Visual modes:

- Visual: `v`
- Visual Line: `V`
- Visual Block: `Ctrl-v`

Can use movement keys to make selection.

Edits

Everything that you used to do with the mouse, you now do with the keyboard using editing commands that compose with movement commands. Here’s where Vim’s interface starts to look like a programming language. Vim’s editing commands are also called “verbs”, because verbs act on nouns.

- `i` enter Insert mode
  but for manipulating/deleting text, want to use something more than backspace
- `o` / `O` insert line below / above
- `d{motion}` delete {motion}
  e.g. `dw` is delete word, `d$` is delete to end of line, `d0` is delete to beginning of line
- `c{motion}` change {motion}
    - e.g. `cw` is change word
    - like `d{motion}` followed by `i`
- `x` delete character (equal do `dl`)
- `s` substitute character (equal to `cl`)

- Visual mode + manipulation
    select text, `d` to delete it or `c` to change it
- `u` to undo, `<C-r>` to redo
- `y` to copy / "yank" (some other commands like `d` also copy)
- `p` to paste
- Lots more to learn: e.g. `~` flips the case of a character

Counts

You can combine nouns and verbs with a count, which will perform a given action a number of times.

- `3w` move 3 words forward
- `5j` move 5 lines down
- `7dw` delete 7 words

Modifiers

You can use modifiers to change the meaning of a noun. Some modifiers are `i`, which means "inner" or "inside", and `a`, which means "around".

- `ci(` change the contents inside the current pair of parentheses
- `ci[` change the contents inside the current pair of square brackets
- `da'` delete a single-quoted string, including the surrounding single quotes

Demo

Here is a broken fizz buzz implementation:

```
def fizz_buzz(limit):
    for i in range(limit):
        if i % 3 == 0:
            print('fizz')
        if i % 5 == 0:
            print('fizz')
        if i % 3 and i % 5:
            print(i)

def main():
    fizz_buzz(10)
```

We will fix the following issues:

- Main is never called
- Starts at 0 instead of 1
- Prints "fizz" and "buzz" on separate lines for multiples of 15
- Prints "fizz" for multiples of 5
- Uses a hard-coded argument of 10 instead of taking a command-line argument

See the lecture video for the demonstration. Compare how the above changes are made using Vim to how you might make the same edits using another program. Notice how very few keystrokes are required in Vim, allowing you to edit at the speed you think.

Customizing Vim

Vim is customized through a plain-text configuration file in `~/.vimrc` (containing Vimscript commands). There are probably lots of basic settings that you want to turn on.

We are providing a well-documented basic config that you can use as a starting point. We recommend using this because it fixes some of Vim's quirky default behavior. **Download our config here and save it to `~/.vimrc`.**

Vim is heavily customizable, and it's worth spending time exploring customization options. You can look at people's dotfiles on GitHub for inspiration, for example, your instructors' Vim configs (Anish, Jon (uses neovim), Jose). There are lots of good blog posts on this topic too. Try not to copy-and-paste people's full configuration, but read it, understand it, and take what you need.

Extending Vim

There are tons of plugins for extending Vim. Contrary to outdated advice that you might find on the internet, you do *not* need to use a plugin manager for Vim (since Vim 8.0). Instead, you can use the built-in package management system. Simply create the directory `~/.vim/pack/vendor/start/`, and put plugins in there (e.g. via `git clone`).

Here are some of our favorite plugins:

- ctrlp.vim: fuzzy file finder
- ack.vim: code search
- nerdtree: file explorer
- vim-easymotion: magic motions

We're trying to avoid giving an overwhelmingly long list of plugins here. You can check out the instructors' dotfiles (Anish, Jon, Jose) to see what other plugins we use. Check out Vim Awesome for more awesome Vim plugins. There are also tons of blog posts on this topic: just search for "best Vim plugins".

Vim-mode in other programs

Many tools support Vim emulation. The quality varies from good to great; depending on the tool, it may not support the fancier Vim features, but most cover the basics pretty well.

Shell

If you're a Bash user, use `set -o vi`. If you use Zsh, `bindkey -v`. For Fish, `fish_vi_key_bindings`. Additionally, no matter what shell you use, you can `export EDITOR=vim`. This is the environment variable used to decide which editor is launched when a program wants to start an editor. For example, `git` will use this editor for commit messages.

Readline

Many programs use the GNU Readline library for their command-line interface. Readline supports (basic) Vim emulation too, which can be enabled by adding the following line to the `~/.inputrc` file:

```
set editing-mode vi
```

With this setting, for example, the Python REPL will support Vim bindings.

Others

There are even vim keybinding extensions for web browsers - some popular ones are Vimium for Google Chrome and Tridactyl for Firefox. You can even get Vim bindings in Jupyter notebooks. Here is a long list of software with vim-like keybindings.

Advanced Vim

Here are a few examples to show you the power of the editor. We can't teach you all of these kinds of things, but you'll learn them as you go. A good heuristic: whenever you're using your editor and you think "there must be a better way of doing this", there probably is: look it up online.

Search and replace

`:s` (substitute) command (documentation).

- `%s/foo/bar/g`
  replace foo with bar globally in file
- `%s/\[.*\](\(.*\))/\1/g`
  replace named Markdown links with plain URLs

Multiple windows

- `:sp` / `:vsp` to split windows
- Can have multiple views of the same buffer.

Macros

- `q{character}` to start recording a macro in register `{character}`
- `q` to stop recording

- `@{character}` replays the macro
- Macro execution stops on error
- `{number}@{character}` executes a macro {number} times
- Macros can be recursive
  - first clear the macro with `q{character}q`
  - record the macro, with `@{character}` to invoke the macro recursively (will be a no-op until recording is complete)
- Example: convert xml to json (file)
  - Array of objects with keys “name” / “email”
  - Use a Python program?
  - Use sed / regexes
    - `g/people/d`
    - `%s/<person>/{/g`
    - `%s/<name>\(.*\)<\/name>/"name": "\1",/g`
    - ...
  - Vim commands / macros
    - `Gdd`, `ggdd` delete first and last lines
    - Macro to format a single element (register `e`)
      - Go to line with `<name>`
      - `qe^r"f>s": "<ESC>f<C"<ESC>q`
    - Macro to format a person
      - Go to line with `<person>`
      - `qpS{<ESC>j@eA,<ESC>j@ejS},<ESC>q`
    - Macro to format a person and go to the next person
      - Go to line with `<person>`
      - `qq@pjq`
    - Execute macro until end of file

      `999@q`
    - Manually remove last `,` and add `[` and `]` delimiters

Resources

- `vimtutor` is a tutorial that comes installed with Vim - if Vim is installed, you should be able to run `vimtutor` from your shell
- Vim Adventures is a game to learn Vim
- Vim Tips Wiki
- Vim Advent Calendar has various Vim tips
- Vim Golf is code golf, but where the programming language is Vim’s UI
- Vi/Vim Stack Exchange
- Vim Screencasts
- Practical Vim (book)

Exercises

1. Complete `vimtutor`. Note: it looks best in a 80x24 (80 columns by 24 lines) terminal window.
2. Download our basic vimrc and save it to `~/.vimrc`. Read through the well-commented file (using Vim!), and observe how Vim looks and behaves slightly differently with the new config.
3. Install and configure a plugin: ctrlp.vim.
    1. Create the plugins directory with `mkdir -p ~/.vim/pack/vendor/start`
    2. Download the plugin: `cd ~/.vim/pack/vendor/start; git clone https://github.com/ctrlpvim/ctrlp.vim`
    3. Read the documentation for the plugin. Try using CtrlP to locate a file by navigating to a project directory, opening Vim, and using the Vim command-line to start `:CtrlP`.
    4. Customize CtrlP by adding configuration to your `~/.vimrc` to open CtrlP by pressing Ctrl-P.
4. To practice using Vim, re-do the Demo from lecture on your own machine.
5. Use Vim for *all* your text editing for the next month. Whenever something seems inefficient, or when you think "there must be a better way", try Googling it, there probably is. If you get stuck, come to office hours or send us an email.
6. Configure your other tools to use Vim bindings (see instructions above).
7. Further customize your `~/.vimrc` and install more plugins.
8. (Advanced) Convert XML to JSON (example file) using Vim macros. Try to do this on your own, but you can look at the macros section above if you get stuck.
