Table of Contents

- Notation
- Philosophy
- Modes
- Buffers, tabs, windows
- Command-line
- Movement
- Selection
- Edits
  - deletion example
- Counts
- Modifiers
- Shell
- Readline
- Others
- Search and replace
- Execution
- Multiple windows
- Macros
- Commands

Based on https://missing.csail.mit.edu/2020/editors/

- learning a new tool
  - steep learning curve initially, slow speed
  - ~ 20 hrs with a new editor you'll be up to speed with benefits saving you time
- look up what to do: if you think there's probably a better way, there most likely is
- Most popular @ 2020
  - GUI editor: VSCode
  - command line editor: vim
- Many tools support vim emulation mode (e.g. VSCode)

Notation

Control-V can be expressed as: * ^V * Ctrl-V * <C-V>

Philosophy

- most of time coding is spent reading/editing rather than writing long blocks of text, so different modes for different tasks
- vim is programmable, and **the interface is a programming language**: keystrokes are commands which are **composable**; this enables efficient movement and edits, especially once the commands become muscle memory
- vim avoids use of mouse and arrow keys because it is too slow/too much movement
- helps editor work at the speed you think

Modes

- vim is a modal editor, i.e. vim has multiple operating modes for different tasks:
  - **normal**: for moving around and making edits;
    - default startup mode
    - return to normal: `<ESC>`
  - **insert**: for inserting text; enter key: `i`
  - **replace**: to overwrite text; `R`
  - selection: for selecting blocks of text
    - visual, `v`
    - visual line, `V`
    - visual block, `<C-V>`
  - command-line: for running a command; `:`
- many people rebind `ESC` to `CapsLock` for ease of use
- The letter `x`:
  - insert mode: inserts a literal character "x"
  - normal mode: deletes character under cursor
  - visual mode: deletes selection
- vim shows current mode in bottom left

Buffers, tabs, windows

- buffers: set of open files
- a vim session has a number of tabs, each of which has a number of windows (split panes), with each window showing a single buffer

- default: Vim opens with a single tab, which contains a single window
- a buffer can be open in multiple windows within the same tab, allowing you to view different parts of a file simultaneously

Command-line

Command mode can be entered by typing `:` in normal mode - `: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

Movement

- should spend most of your time in normal mode, using movement commands to navigate the buffer
- movements in Vim are called "nouns" because they refer to chunks of text.
- Basic movement: `hjkl` (left, down, up, right)
  - NB historical reason - hjkl keyboard was arranged as a d-pad
- 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)
- Move forwards: `Ctrl-f`, move backwards: `Ctrl-b`
- Centre screen on cursor: `zz`
- File: `gg` (beginning of file), `G` (end of file)
- Current line: `Ctrl-g`
- 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
- Visual Line
- Visual Block

Can use movement keys to make selection.

Edits

- Mouse actions are done 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 `xi`)
- 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

deletion example

Many commands that change text are made of an operator (noun) and a motion (verb): e.g. when used with deletion operator: `d` * `w`: delete until the start of the next word, EXCLUDING first character * `e`: delete until the end of the current word, INCLUDING the last character * `$`: delete to end of the line, INCLUDING the last 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

- modifiers 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:

```python
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
- main is never called
  - `G` end of file
  - `o` open new line below
  - type in "if **name** …" thing
- starts at 0 instead of 1
  - search for `/range`
  - `ww` to move forward 2 words
  - `i` to insert text, "1,"
  - `ea` to insert after limit, "+1"
- newline for "fizzbuzz"
  - `jj$i` to insert text at end of line
  - add ", end=' '"
  - `jj.` to repeat for second print
  - `jjo` to open line below if
  - add "else: print()"
- fizz fizz
  - `ci'` to change fizz
- command-line argument
  - `ggO` to open above
  - "import sys"
  - `/10`
  - `ci(` to "int(sys.argv[1])"

Customizing Vim

- plain-text configuration file: `~/.vimrc`
- missing semester vimrc

Extending Vim with plugins

- you do *not* need to use a plugin manager for Vim (since Vim 8.0),
- you can use the built-in package management system: create the directory `~/.vim/pack/vendor/start/`, and put plugins in there (e.g. via `git clone`).
- ctrlp.vim: fuzzy file finder
- ack.vim: code search
- nerdtree: file explorer
- vim-easymotion: magic motions
- instructors' dotfiles: Anish Jon Jose)
- Vim Awesome for more awesome 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

For vim keybindings: * bash: `set -o vi` * Zsh, `bindkey -v`

Set default editor: `export EDITOR=vim`

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

- 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.

Advanced Vim

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

- `/foo` search forward for foo
- `?foo` search backward for foo
- `n` goes to next occurrence in same direction; `N` goes to next occurrence in opposite direction
- `%` on a bracket `([{` goes to its match

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

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

Execution

Type `:!` followed by command to execute external command

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

Integration with PyCharm

1. Install IdeaVim extension
2. Add .ideavimrc file in your home directory:
   - Windows: `C:\Users\James`
   - Linux: ~/
3. Activate plugins

Commands

- surround helps you surround code
  - `ys`: you surround; surround current with
  - `cs`: change surround; change surrounding characters
  - `ds`: delete surrounding characters
  - `S`: surround in visual mode?
  - examples: `Hello world!`
    * `ysiw]`: `[Hello] world!`
    * `cs]}`: `{Hello} world!`; change to braces without spaces
    * `yss(`: `( {Hello} world! )`; surround whole line with parentheses
    * `ds}ds(`: `Hello world!`
    * `ysiw<em>`: `<em>Hello world!</em>`; add emphasis tags
    * `VS<p class="important">`: `<p class="important> <em>Hello world!</em> </p>` surround in linewise visual mode
- easymotion helps you to move around
  - at time of writing `<leader>` was configured as \
  - `\\w`: trigger word motion
  - `\\fo`: trigger find motion looking for character `o`
  - Tutorial
- commentary: comments out lines
  - `gcc`: comment out a line
  - `gc`: comment target of a motion
    * `gcap`: comment out a paragraph
  - `:g`: comment command e.g. `:g/TODO/Commentary`
- multiple-cursors:

  - `<A-n>`
  - `<A-x>`
  - `<A-p>`
  - `g<A-n>`

- nerdcommenter:

  - `<ldr>cc`: comment the current line (or selection in visual mode) out
  - `<ldr>c<space>`: toggle current line (or selection)

Resources

- `vimtutor` is a tutorial that comes installed with Vim
- 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.