Visit http://learnbyexample.net/ for more information

Vim: Introduction

Vim is a text editor, powerful at that. It has evolved from vi to vim (Vi Improved) and gvim (Vim with GUI). Gvim is suggested for a newbies to start with (has menu and toolbar like other easy-to-use text editors)

- Vim is not a full fledged writing software like LibreOffice Writer or MS Word. It is a text editor, mainly used for writing programs
- Notepad in Windows is an example for text editor. There are no formatting options like bold, heading, page numbers etc
- If you have written C program within IDE, you might have noticed that keywords are colored, automatic indentation for code within {}, etc. Vim provides all of that and more

Like the Linux shell, Vim has programmable abilities. One can execute shell commands right within Vim and incorporate their outputs. Text can be modified, sorted, regular expressions be applied for complicated search and replace. Record a series of editing commands and execute on other portions of text in same file or another file. To sum it up - for most of editing tasks, no need to write an external script program, they can be managed within Vim

Now, one might wonder, what is all this need for complicated editing features? Why does a text editor require programming capablities? Why is there even a requirement to learn using a text editor? All I need from editor is writing text with keyboard, use Backspace or Delete key, have some GUI button for opening and saving, preferrably a search and replace feature and that is all.

A simple and short answer is: to reduce repititive manual task and the time taken to type a computer code. Faster editing probably makes sense. Reduce typing time? seems absurd. But when you are in the middle of coding a large program, thinking about complicated logic, typing should be as minimal as possible

Ice Breaker – first Vim editing

Open a terminal and try these steps:

1. **gvim** first_vim.txt → open file first_vim.txt for editing (use **vim** if gvim is not available)
2. Press i key (yes, the lowercase alphabet i, not any special key)
3. Start typing, say 'Hello Vim'
4. Press Esc key
5. Press : key
6. Type wq
7. Press Enter key
8. **cat** first_vim.txt → sanity check to see what you typed is saved or not

Phew, what a complicated procedure to write a simple line of text, isn't it? This is the most challenging and confusing part for a newbie to Vim. Don't proceed any further if you haven't got a hang of this, take help of youtube videos if you must. Master this procedure and you are ready for awesomeness of Vim

Vim help and tutorial

- **gvimtutor** → shell command, opens a tutor file, has instructions to get started with vim, **vimtutor** if gvim is not available
- :help → use Help menu in gvim, or type :help in normal mode

- how to grok vi → other answers are helpful as well in this stackoverflow thread
- Vim primer → good tutorial on vim basics to advanced topics
- Learn Vimscript → customize Vim, learn Vimscript, creating plugins
- Vim Tips and Tricks
- Vim cheat sheet , another cheat sheet
- Vim startup options
- Online courses: Udemy , Pluralsight , Lynda , openvim , shortcutfoo , youtube , youtube , vimeo , vimeo
- Books: oualline , practical vim
- vimcasts → videos and articles on different Vim topics
- vim-adventures → learn basic vim commands as a game, requires payment for more content

Opening Vim

- **gvim** script.sh → opens file script.sh, blank document if script.sh doesn't exist and new document if filename is not specified
- **gvim** report.log power.log area.log → open specified files, but only report.log is in the current buffer. Use :n (or :next) to go to next file for editing
- **gvim** report.log power.log area.log -p → open specified files in separate tabs
- **gvim** +/while script.sh → open script.sh and place the cursor under first occurrence of while

Modes of Operation

Vim is best described as a modal editor. To keep it simple, there are three modes of operation:

Insert mode

This is the mode where required text is typed. Usual editing options available are Delete, Backspace and Arrow keys. Some smart editing short-cuts:

- Ctrl+w → delete the current word
- Ctrl+p → auto complete word based on matching words in backward direction, if more than one word match, they are displayed as a list
- Ctrl+n → auto complete word based on matching words in forward direction
- Ctrl+t → indent current line
- Ctrl+d → unindent current line

Pressing **Esc** key changes the mode back to Normal mode

Normal mode

This is the default mode when Vim is opened. This mode is used to run commands for editing operations like copy, delete, paste, recording, saving and running series of edit commands, moving around file, etc. It is also referred to as Command mode

Below commands are commonly used to change modes from Normal mode:

- i → insert, pressing i (lowercase) key changes the mode to Insert mode, places cursor to the left. Use I to place the cursor at start of line

- a → append, pressing a (lowercase) key changes the mode to Insert mode, places cursor to the right. Use A to place the cursor at end of line
- o → open new line below the current line and change to Insert mode, use O to open above
- s → delete character under the cursor and change to Insert mode, S to delete entire line
- : → changes the mode to Command Line mode
- / → change to Command Line mode for searching in forward direction, use ? to start searching in backward direction

Visual mode is a subset of Normal mode where text is first selected before editing

Command Line mode

Also known as Ex mode or Last Line mode. After : is pressed in Normal mode, the prompt appears in Command Line of Vim window – hence the name. This mode is used to perform file operations like save, quit, search, replace, shell commands, etc. Any operation is completed by pressing **Enter** key after which the mode changes back to Normal mode. Press **Esc** key to ignore whatever is typed and return to Normal mode

Vim: Basic editing in Normal mode

Let's start with most basic editing. Remember to be in Normal mode, press Esc key from other modes to switch back to Normal mode. Gvim provides visual clue to which mode you are in too – in Insert mode, the cursor is a blinking | character, also the word 'INSERT' can be seen on the Command Line, in Command Line mode, the cursor is of course on the Command Line. In Normal mode, the cursor is a rectangular block like this ▌

> Material presented here is based on Gvim, which has few subtle differences from vim – see here

Cut

There are various ways to delete text, which can then be pasted elsewhere using paste command

- Select text (using mouse or v command), press d to delete the text
- dd → delete current line
- 2dd → delete current line and the line below it - total 2 lines
  - dj and d↓ also does the same (j and ↓ both function as down arrow key)
- dk → delete current line and the line above it (d↑ does the same)
- 10dd → delete current line and 9 lines below it - total 10 lines
- D → delete from current character to end of line
- x → delete only the character under cursor
- 5x → delete character under cursor and 4 characters to its right - total 5 characters
- cc → delete current line and change to Insert mode
- 4cc → delete current line and 3 lines below it and change to Insert mode - total 4 lines
- C → delete from current character to end of line and change to Insert mode
- s → delete only the character under cursor and change to Insert mode
- 5s → delete character under cursor and 4 characters to its right and change to Insert mode - total 5 characters
- S → delete current line and change to Insert mode (same as cc)

Copy

There are various ways to copy text using the **yank** command y

- Select text to copy (using mouse or v command), press y to copy the text
- yy → copy current line, Y also does the same
- 2yy → copy current line and the line below it - total 2 lines
  - yj and y↓ does the same
- yk → copy current line and the line above it (y↑ does the same)
- 10yy → copy current line and 9 lines below it (total 10 lines)

Paste

Use the **paste** command p. Used after cut or copy operations

- p → paste the copied content one time. If the copied text was less than a line, paste the content to **right**

the cursor, otherwise paste **below** the current line

- P → paste the copied content one time. If the copied text was less than a line, paste the content to **left** the cursor, otherwise paste **above** the current line
- 3p → paste the copied content three times

Undo

Quite simple, **undo** command is u. Keep pressing u for multiple undo

Redo

Ctrl+r to redo a change

Replace character(s)

Sometimes, we just need to change one character (ex: changing i to j)

- rj → replace the charcter under cursor with j
- ry → replace the character under cursor with y
- 3ra → replace character under cursor and 2 characters to the right with aaa

To replace mutliple characters with different characters, use uppercase R

- Rlion+Esc → replace character under cursor and 3 characters to right with lion. Esc key marks the completion of R command returns to Normal mode

The advantage of r and R commands is that one remains in Normal mode, without the need to switch to Insert mode and back

Repeat

- . → the dot command repeats the last command. If the last command was deleting two lines and dot key is pressed, two more lines will get deleted. If last command was to select five characters and delete them, dot key will select five characters and delete

Open new line

- o → open new line below the current line and change to Insert mode
- O → open new line above the current line and change to Insert mode

Indenting

- >> → indent current line
- 3>> → indent current line and two lines below
- << → unindent current line
- 5>> → unindent current line and four lines below

Vim: Moving Around

Apart from getting the cursor to desired location, the movement commands in Normal mode are very useful in actual editing too. They can be paired with y, d, c for advanced editing and macro editing with q command

Arrow Movements

The four arrow keys can be used in Vim to move around as in any other text editor. Vim also maps them to four characters in Normal mode (improves typing speed compared to arrow keys)

- h → left
- j → down
- k → up
- l → right

Within current line

- 0 → move to beginning of current line – column number 1
- ^ → move to beginning of first non-white-space character of current line, useful when code is indented within control structures
- $ → move to end of current line

Word and Character based move

- w → move to start of next word or punctuation mark (192.1.168.43 requires multiple w movements)
- W → move to start of word after the next white-space (192.1.168.43 requires one W movement)
- b → move to beginning of current word (if cursor is not at start of word) or beginning of previous word or previous punctuation mark
- B → move to beginning of current word (if cursor is not at start of word) or beginning of previous word (punctuation is treated as part of word)
- e → move to end of current word (if cursor is not already at end of word) or end of next word or next punctuation mark
- E → move to end of current word (if cursor is not already at end of word) or end of next word (punctuation is treated as part of word)
- 3w → move 3 words forward, similarly number can be prefixed for W,b,B,e,E
- f(→ move forward in the current line to next occurrence of character (, fb to move to character b, etc
- 3f" → move forward third occurrence of character " in current line
- t; → move forward in the current line to character just before ; (t stands for tail)
- 3tx → move forward to character just before third occurrence of character x in current line
- Fa → move backward in the current line to character a
- Ta → move backward in the current line to character just after a

Within current file

- gg → move to first non-white-space character of first line of file
- G → move to first non-white-space character of last line of file

- 5G → move to first non-white-space character of 5th line of file, similarly 10G for 10th line and so on
- :5 → move to first non-white-space character of 5th line of file, similarly :10 for 10th line and so on
- % → move to matching pair of brackets like () {} [] (Nesting is taken care) It is possible to match begin and end pair (used in Verilog) with %. Refer here

Scrolling

- Ctrl+d → scroll half page down
- Ctrl+u → scroll half page up
- Ctrl+f → scroll one page forward
- Ctrl+b → scroll one page backward
- Ctrl+Mouse Scroll → scroll one page forward or backward

Marking frequently used locations

- ma → mark any location in the file with variable a (use any of the 26 alphabets)
- `a → move from anywhere in the file to exact location marked by a
  - 'a will move to first non-white-space character of the line marked by a
  - :marks will show existing marks

Jumping back and forth

If using marks doesn't suit you or if there are too many frequent locations, try these

- Ctrl+o → navigate to previous location (remember it by thinking o as old)
- Ctrl+i → navigate to next location
- Read more

Vim: Command Line Mode

Used for saving, opening new file, new tab, new split screen, executing Vim as well as shell commands, etc. All commands are completed by pressing the Enter key after which mode is changed back to Normal mode

Saving changes

- :w → save changes
- :w *filename* → provide a filename if it is a new file or if you want to save to another file
- :w! → save changes even if file is read-only, provided user has appropriate permissions
- :wa → save changes made to all the files opened

Exit Vim

- :q → quit the current file (if other tabs are open, they will remain) – if unsaved changes are there, you will get an error message
- :qa → quit all
- :q! → quit and ignore any unsaved changes

Combining Save and Quit

- :wq → save changes and quit
- :wq! → save changes even if file is read-only and quit

Editing buffer(s)

When multiple files are opened without -p option, they get indexed as various buffers and different ways are there to navigate through them

- :e → refreshes the current buffer
- :e *filename* → open file for editing, existing file if any gets added to list of open buffers
- :tabnew *filename* → open file for editing in new tab instead of adding to buffers
- :split *filename* → open file for editing in new horizontal split screen, Vim adds a highlighted horizontal bar with filename for each file thus opened
- :vsplit *filename* → open file for editing in new vertical split screen
- :e# → switch back to previous buffer (Ctrl+^ also does same)
- :e#1 → open first buffer
    - :e#2 open second buffer and so on
- :n → open next buffer (gt to go to next tab)
- :b → open previous buffer (gT to go to previous tab)
- :silent! bufdo %s/search pattern/replacepattern/g | update → if multiple buffers are open and you want to apply common editing across all of them, use bufdo command. But it is not an efficient way to open buffers just to search and replace a pattern across multiple files, use linux **sed** command instead. Read more about argdo and bufdo here

Search

- /search pattern → search the given pattern in forward direction. Use n command to move to next match and N for previous match
- ?search pattern → search the given pattern in backward direction. n command goes to next match in backward direction and N moves to next match in forward direction
- :set hlsearch → highlights the matched pattern
- :set nohlsearch → do not highlight matched pattern
- :noh → clear highlighted patterns, doesn't affect hlsearch settings

Search and Replace

General syntax is **:range s/searchpattern/replacepattern/options**. s is short-form for substitute command. The substitution behavior can be changed using options

- :1,5 s/call/jump/g → replace all occurrences of call to jump in lines 1-5. Without g option, only first occurrence in each line will get replaced
- :1,$ s/call/jump/g → replace all occurrences of call to jump in entire file, $ points to Command Line of file
- :% s/call/jump/g → replace all occurrences of call to jump in entire file, % is short-cut for 1,$

Deleting lines based on pattern

Some times, we need to delete lines containing particular pattern or delete lines not having a particular pattern. Vim has got you covered

- :g/call/d → delete all lines containing the word call
- :1,5 g/call/d → delete lines containing the word call only from first five lines
- :v/jump/d → delete all lines NOT containing the word jump

Read more about the power of g command

Search and Replace on lines matching a pattern

The g command also allows qualifying search and replace to lines matching a patterns

- :g/cat/ s/animal/mammal/g → replace 'animal' with 'mammal' only on lines containing the pattern 'cat'

Shell Commands

One can use shell commands within Vim

- :.! date → replace current line with output of **date** command
- :%! sort → sort all lines of file
- :3,8! sort → sort lines 3 to 8 of file
- One can also select text visually, press : and use ! followed by shell command
- :r! date → insert output of **date** command below current line
- :r report.log → insert contents of file report.log below current line
- :sh → open a shell session within Vim, use **exit** command to return to editing
- Read more

Vim: Advanced editing

Misc

- gf → open the file pointed by file path under the cursor
- * → searches the word under the cursor in forward direction - matches only the whole word
- g* → searches the word under the cursor in forward direction - matches as part of another word also
- # → searches the word under the cursor in backward direction - matches only the whole word
- g# → searches the word under the cursor in backward direction - matches as part of another word also
- Ctrl+g →display file information like name, number of lines, etc at bottom of screen
- J → join current and next line with one space character in between (can also be operated on selected line using mouse or visual mode selection)
- gJ → join current and next line without any character in between

Changing Case

- ~ → invert the case of selected text, i.e lowercase becomes uppercase and vice versa
- U or gU → change selected text to uppercase
- gu → change selected text to lowercase
- Capitalize words and regions

Numbers

- :set number → prefix line numbers (it is a visual guideline, won't modify text)
- :set nonumber → remove line number prefix
- Ctrl+a → increment a number (decimal/octal/hex will be automatically recognized - octal is a number prefixed by '0' and hex by '0x')
- Ctrl+x → decrement a number

Multiple copy-paste using " registers

One can use lowercase alphabets a-z to save content for future processing. And append content to those registers by using corresponding uppercase alphabets A-Z at later stage

- "ayy → copy current line to "a register
- "ap → paste content from "a register
- "ryiw → copy word under cursor to "r register
- "Ayj → append current line and line below it to "a register ("a has total 3 lines now)

Special registers

Vim also has special purpose registers with pre-defined behavior

- " → all yanked/deleted text is stored in this register. p command is a short-cut for ""p
- "0 → yanked text is stored in this register. Use "0p to paste the text. Useful for this sequence: yanking content, deleting something and then pasting

- "1 to "9 → deleted contents are stored in these registers and get shifted with each new deletion. 1 has the most recent deleted content
    - "2p → paste content deleted before the last deletion
- "+ → this register stores system clipboard contents
    - gg"+yG → copy entire file contents to clipboard
    - "+p → paste content from clipboard
- "* → this register stores visually selected text, can be pasted using middle mouse button click or "*p
- how to use Vim registers
- Using registers on Command Line mode

Combining editing commands with movement commands

- dG → delete from current line to end of file
- dgg → delete from current line to beginning of file
- d`a → delete from current character upto location marked by a
- d% → delete upto matching bracket sets like (), {}, []
- ce → delete till end of word and change to Insert mode
- yl → copy character under cursor

Context editing

We have seen movement using w,%,f etc But they require precise positioning to be effective. Vim provides a way to modify commands that accepts movement like y,c,d to recognize context. These are i and a constructs – easy way to remember their subtle differences is to think of i as *inside* and a as *around*. Examples:

- diw → deletes a word regardless of where the cursor is on that word. Equivalent to using de when cursor is on first character of the word
- daw → deletes a word regardless of where the cursor is on that word and a smart deletion of space character to left/right of the word depending on its position as part of sentence
- dis → delete a sentence regardless of where the cursor is on that sentence
- yas → copy a sentence regardless of where the cursor is on that sentence and a smart space left/right
- cip → delete a paragraph regardless of where the cursor is on that paragraph and change to Insert mode
- di" → delete all characters within pair of double quotes, regardless of where cursor is within quotes
- da" → delete all characters within pair of double quotes as well as the quotes
- ci(→ delete all characters within () and change to Insert mode. Works even if matching parenthesis are spread over multiple lines
- ya} → copy all characters within {} including the {}. Works even if matching braces are spread over multiple lines

Visual Mode

As the name indicates, editing is done after visually selecting text. Selection can either be done using mouse or using visual commands

- v → start visual selection, use any movement command to complete selection

- V → select current line
- Ctrl+v → visually select column(s)

Editing

After visual selection

- d → delete the selected text
- c → clear the selected text, type any text and press Esc key. The typed text is repeated across all lines if the selection was column using Ctrl+v
- I → after selecting with Ctrl+v, press I, type text and press Esc key to replicate text across all lines to left of the selected column
- A → after selecting with Ctrl+v, press A, type text and press Esc key to replicate text across all lines to right of the selected column
- ra → replace every character of the selected text with a
- : → perform Command Line mode editing commands like g,s,! on selected text

Indenting

After visual selection

- > → indent the selected lines
- 3> → indent the selected lines three times
- < → unindent the selected lines
- Read more

Regular Expressions

The search pattern can be significantly improved to match exact requirements using regular expressions. Regular expressions can be applied whereever pattern is used, like /,?,:s,:g,:v

Example given in indented bullet points is for forward direction searching using / Use ? For backward search

Modifiers for Search & Replace

- g → all occurrences within line
- c → ask for confirmation before every replacement
- i → ignorecase while using the search pattern
    - :% s/cat/Dog/gi → replace every occurrence of cat (ignoring case, so it matches Cat, cAt, etc) with Dog (note that i doesn't affect the replacement pattern Dog)

Meta characters

- ^ → start matching from beginning of a line
    - /^for → match for only at beginning of line
- $ → match pattern should terminate at end of a line
    - /)$ → match) only at end of line
    - /^$ → match empty line

- . → any 'one' character
  - /c.t → match 'cat' or 'cot' or 'c2t' but not 'cant'

Pattern Qualifiers

- * →greedy match preceding character 0 or more times
  - /abc* → match 'ab' or 'abc' or 'abccc' or 'abccccc' etc
- \+ → match preceding character 1 or more times
  - /abc\+ → match 'abc' or 'abccc' but not 'ab'
- \= → match preceding character 0 or 1 times
  - /abc\= → match 'ab' or 'abc' but not 'abcc'
- {-} → non-greedy match preceding character 0 or more times
  - excellent answers on non-greedy matching in vim . Consider a line has the text 'This is a sample text'
    - /h.{-}s → will match: **his**
    - /h.*s → will match: **his is a s**
- \{min,max} → match preceding character min to max times (including min and max)
  - min or max can be left unspecified as they default to 0 and infinity respectively
- \{number} → match exactly with specified number
  - /c\{5} → match exactly 'ccccc'

Multiple and Saving Patterns

- \| → allows to specify two or more patterns to be matched
  - /min\|max → match 'min' or 'max'
- \(pattern\) → allows to save matched patterns and use special variables \1, \2, etc to represent them in same search pattern and/or replace pattern when using substitute command
  - /\(\a\)\1 → match repeated alphabets

Word Boundary

- \< → Bind the search pattern to necessarily be starting characters of a word
  - /\<his → matches 'his' and 'history' but not 'this'
- \> → Bind the search pattern to necessarily be ending characters of a word
  - /his\> → matches 'his' and 'this' but not 'history'
- \<pattern\> → Bind the search pattern to exactly match whole word
  - /\<his\> → matches 'his' and not 'this' or 'history'

Character Patterns

- [abcde] → match any of 'a' or 'b' or 'c' or 'd' or 'e' ONE time, use [a-e] as shortform
- [aeiou] → match any vowel character
- [^abcde] → match any character other than 'a' or 'b' or 'c' or 'd' or 'e', use [^a-e] as shortform
- [^aeiou] → match any consonant character

- \a → matches any alphabet, short-cut for [a-zA-Z]
- \A → matches any non-alphabet, short-cut for [^a-zA-Z]
- \d → matches any number, short-cut for [0-9]
- \D → matches any non number, short-cut for [^0-9]
- \x → matches hexademical character, short-cut for [0-9a-fA-F]
- \X → matches non hexademical character, short-cut for [^0-9a-fA-F]
- \w → matches any alphanumeric character or underscore, short-cut for [a-zA-Z0-9_]
- \W → match other than alphanumeric character or underscore, short-cut for [^a-zA-Z0-9_]
- \u → matches uppercase alphabets [A-Z]
- \l → matches lowercase alphabets [a-z]
- \s → matches white-space characters **space** or **tab**
- \S → matches other than white-space characters

The \v and \V modes

With so many meta characters and escape sequences, it is helpful to have alternate cleaner syntax

- \v → no need to use \ for escape sequence characters
  - /\vc{5} → match exactly 'ccccc'
  - /\vmin|max → match 'min' or 'max'
  - /\v(\a)\1 → match repeated alphabets
  - /\vabc+ → match 'abc' or 'abccc' but not 'ab'
  - /\vabc= → match 'ab' or 'abc' but not 'abcc'
  - /\v<his> → match whole word 'his', not 'this' or 'history'
  - :% s/\v(\d+) (\d+)/\2 \1/ → swap around two numbers separated by space
- \V → no need to use \ for meta characters
  - /V(.*) → literally match the sequence (.*)
- Read more

Specifying Case Sensitivity while searching

- \c → case insensitive search
  - /\cthis → matches 'this', 'This', 'thiS', etc
- \C → case sensitive search
  - /\Cthis → match exactly 'this', not 'This', 'thiS', etc
- Excellent examples and other Vim settings on case sensitivity

Changing Case in Search and Replace

- :% s/\v(\a+)/\u\1/g → \u modifier will Capitalize words (i.e only first character of word is capitalized)
- :% s/\v(\a+)/\U\1/g → \U modifier will change case to UPPERCASE

- \l and \L are equivalent for lowercase
- Changing case with regular expressions – also see how to use \e and \E to end further case changes

Delimiters in Search and Replace

One can also use other characters like #,$ instead of /

- :% s#/project/adder/#/verilog/half_adder/#g → this avoids mess of having to use \/ for every / character used as part of search and replace patterns

Recording Macro – advanced repeat command

The repeat command can only record and repeat the last editing command. It also gets overwritten with every editing command. The q command allows user to record any sequence of editing commands to effectively create a user defined command, which can then be applied on other text across files. It can also be prefixed with a number to repeat the command. Powerful indeed!

1. Press q to start recording session
2. Use any alphabet to store the recording, say a
3. Use the various editing commands in combination with movement commands to accomplish the sequence of editing required
4. Press q again to stop recording
5. Use @a (the character typed in step 2) to execute the recorded command elsewhere

Suggested Reading

- Macros
- Macro example - youtube video

Further reading on Advanced Editing

- Productive Vim tips – has mutliple pages on numerous useful short-cuts and tips. The top answer is one of the best introduction to Vi and its core editing concepts explained as a language
- Visual selection
- Vim registers
- Advanced editing - plain text or code
- Search and replace

Vim: Customizing the editor

Different programming languages require different syntax, indentation, etc. The company you work for might have its own text format and guidelines. You might want to create a short-cut for frequently used commands or create your own command. Vim's customizable options are versatile as well

- ~/.vimrc → put your custom settings in this file
    - Using settings in ~/.vim folder is not covered here, refer to links provided at end of this topic

All the settings presented below is also available as a downloadable file

General Settings

- **set** nocompatible → Use Vim defaults and not those of Vi
- **set** history=100 → increase default history from 20 to 100
- **set** nobackup → don't create backup files
- **set** noswapfile → don't create swap files
    - Read more on backup and swap files as well as better ways to manage them - disabling swap files and vim backup files
- **colorscheme** murphy → a dark colored theme

Text and Indent Settings

- **set** textwidth=80 → no. of characters in a line after which Vim will automatically create new line
- **filetype** plugin indent on → Vim installation comes with lot of defaults. :echo $VIMRUNTIME gives your installation directory. One of them is 'indent' directory which has indent styles for 100+ different file types. Using this setting directs Vim to apply based on file extensions like .pl for Perl, .c for C, .v for verilog, etc
- **set** shiftwidth=4 → defines how many characters to use when using indentation commands like >>
- **set** tabstop=4 → defines how many characters Tab key is made of
- **set** expandtab → change Tab key to be made up of Space characters instead of single Tab character
- **set** cursorline → highlight the cursor line

Mappings

Mappings allow to create new commands or redefine existing ones. Mappings can be defined for specific mode. Examples given here are restricted to specific mode and non-recursive

- **nnoremap** → Normal mode mapping
- **vnoremap** → Visual mode mapping
- **inoremap** → Insert mode mapping

Search Settings

- **set** incsearch → search as you type

- **set** hlsearch → highlight search pattern
- **vnoremap** * y/<C-R>"<CR> → press * to search visually selected text in forward direction
- **vnoremap** # y?<C-R>"<CR> → press # to search visually selected text in backward direction
- **nnoremap** / /\v → automatically add magic mode modifier for forward direction search
- **nnoremap** ? ?\v → automatically add magic mode modifier for backward direction search
- **nnoremap** <silent> <Space> :noh<CR><Space> → Press Space to clear highlighted searches. The <silent> modifier executes the command without displaying on Command Line. Note that command also retains default behavior of Space key

Custom Mappings

Examples here use function keys. A better approach is to use Leaders

- **nnoremap** #2 :w<CR> → Press F2 function key to save file. Pressing Esc key gets quite natural after writing text in Insert mode. Instead of multiple key presses to save using Command Line, F2 is easier
- **inoremap** <F2> <Esc>:w<CR>a → Press F2 function key to save file in Insert mode as well. Note the mapping sequence – it requires going to Normal mode first. Hence the a command to get back to Insert
- **nnoremap** #3 :wq<CR> → Press F3 to save the file and quit
- **nnoremap** #4 ggdG → Press F4 to delete entire contents of file
- **nnoremap** #5 gg"+yG → Press F5 to copy entire contents of file to system clipboard

Abbreviations

From typo correction to short-cuts to save typing, abbreviations are quite useful. Abbreviations get expanded only when they stand apart as a word by itself and not part of another word. For example, if the letter p is abbreviated, it is activated in variety of ways like pressing Esc, Space, Enter, Punctuations, etc not when it is part of words like pen, up, etc. Only Insert mode examples are given here:

- **iab** p #!/usr/bin/perl<CR>use strict;<CR>use warnings;<CR> → In Insert mode, just type p followed by Enter key – this will automatically add the Perl interpreter path and two statements
- **iab** py #!/usr/bin/python3 → In Insert mode, use py for Python interpreter path
- **iab** teh the → Automatically correct typo teh to the
- **iab** @a always @()<CR>begin<CR>end<Esc>2k$ → This one works best with @a followed by Esc key in Insert mode. This inserts an empty always block (used in Verilog) and places the cursor at end of first line. After which use i command to type code inside the parenthesis

autocmd

Execute commands based on events – read more (also shows how to organize the use of autocmd)

- **autocmd** FileType * setlocal formatoptions-=r → By default, if you type out a single line comment in programming languages like Perl, C, etc, and press Enter, Vim will automatically add comment character to the new line. If you do not like this behavior, use this autocmd
- **autocmd** BufNewFile,BufRead *.html setlocal tw=0 → autocmd are very useful to set a different config

based on file type. In this case, the textwidth option is set to 0 for html files (i.e no limit to number of characters on a single line)

- **autocmd** BufNewFile *.pl :normal ip → From previous section, we saw how **iab** is used to create Insert mode short-cuts to save typing. Using the previously defined setting for p letter, this autocmd will add those three lines whenever a new Perl file (file extension .pl) is created. Won't effect when existing files are opened
- **autocmd** BufNewFile *.py :normal ipy → Similarly for Python files

Customizing Verilog Editing Experience

- **runtime** macros/matchit.vim
- **let** b:match_words =

  \ '\<module\>:\<endmodule\>,' .

  \ '\<begin\>:\<end\>,' .

  \ '\<if\>:\<else\>'

  - This will allow % to match module-endmodule, begin-end and if-else to function as matching keywords, just like how % acts on (),{},[] It will even take care of nesting. How cool is that!
- **set** suffixesadd+=.v,.V,.sv,.SV
  - gf command opens file under the cursor. By adding these suffixes, gf will now also work on module names
- **set** path+=/path1/,/path2/,etc
  - what if modules are in different hierarchy? Add those directories to path
- For more customizations, read this

Suggested Reading

- Excellent book on Vimscript and customizing Vim
- Useful .vimrc settings
- Interesting vimrc links: vim-sensible , sane defaults , minimal for new users , generate vimrc , advanced , building vimrc from scratch
- Vim plugins
- using gf to open file in new tab or splits
- open file under cursor with gf based on current file extension
- Vim tips and tricks
- Using command line history
- Vim record history
- Indenting source code
- Indenting in Vim with all the files in folder