learnbyexample

Vim Reference Guide

Sundeep Agarwal

Table of contents

Preface

Vim Reference Guide is intended as a concise learning resource for beginner to intermediate level Vim users. It has more in common with cheatsheets than a typical text book. Most features are presented using a sample usage. Topics like Regular Expressions and Macros have more detailed explanations and examples due to their complexity.

The features covered in this guide are shaped and limited by my own experiences since 2007. Fourteen years would seem a long time to have already become an expert, but I'm not there yet (nor do I have a pressing need for such expertise). The earlier version of this guide was written five years back and I still took more than three months to get it fit for publication. A large portion of that time was spent correcting my understanding of Vim commands, going through user and reference manuals, getting good at using built-in help, learning more features and so on.

Prerequisites

I do give a brief introduction to get started with using Vim, but having prior experience would be ideal before using this resource. As a minimum requirement, you should be able to use `vimtutor` on your own.

See my Vim curated list for links to tutorials, books, interactive resources, cheatsheets, tips, tricks, forums and so on.

Conventions

- This guide is based on **Vim version 8.1** and some instructions assume Unix/Linux like operating system. Where possible, details and resources are mentioned for other platforms.
- I prefer using **GVim**, so you might find some differences if you are using **Vim**.
- Built-in help command examples are also linked to an online version. For example, clicking :h usr_toc.txt will take you to table of contents for Vim User Manual. `:h usr_toc.txt` is also a command that you can use from within Vim.
- External links are provided throughout the book for exploring some topics in more depth.
- vim_reference repo has markdown source and other details related to the book. If you are not familiar with `git` command, click the **Code** button on the webpage to get the files.

How to use this guide

- Since many chapters take the form of cheatsheet with examples, this is a densely packed guide. Feel free to skim read some sections (because you already know them, not applicable for your use cases, etc), but try not to skip them entirely.
- If you are not able to understand a particular feature, go through the Vim user manual for that topic first. Each chapter has related documentation links at the top and external learning resources are often mentioned at the end of command descriptions.
- Practice the commands multiple times to build muscle memory.
- Building your own cheatsheet is highly recommended. You wouldn't need to refer most of the basic commands often, so you'll end up with a manageable reference sheet. As you continue to build muscle memory, you can prune the cheatsheet further.
- This guide covers a lot, but not everything. So, you'll need to learn from other resources too and add to your personal cheatsheet.

Acknowledgements

- Vim help files — user and reference manuals
- /r/vim/ and vi.stackexchange — helpful forums
- canva — cover image
- Warning and Info icons by Amada44 under public domain
- oxipng, pngquant and svgcleaner for optimizing images
- Rodrigo Girão Serrão for feedback and suggestions
- Andy for cover image suggestions

Feedback and Errata

I would highly appreciate if you'd let me know how you felt about this book, it would help to improve this book as well as my future attempts. Also, please do let me know if you spot any error or typo.

Issue Manager: https://github.com/learnbyexample/vim_reference/issues

E-mail: learnbyexample.net@gmail.com

Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a lazy being who prefers to work just enough to support his modest lifestyle. He accumulated vast wealth working as a Design Engineer at Analog Devices and retired from the corporate world at the ripe age of twenty-eight. Unfortunately, he squandered his savings within a few years and had to scramble trying to earn a living. Against all odds, selling programming ebooks saved his lazy self from having to look for a job again. He can now afford all the fantasy ebooks he wants to read and spends unhealthy amount of time browsing the internet.

When the creative muse strikes, he can be found working on yet another programming ebook (which invariably ends up having at least one example with regular expressions). Researching materials for his ebooks and everyday social media usage drowned his bookmarks, so he maintains curated resource lists for sanity sake. He is thankful for free learning resources and open source tools. His own contributions can be found at https://github.com/learnbyexample.

List of books: https://learnbyexample.github.io/books/

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License

Code snippets are available under MIT License

Resources mentioned in Acknowledgements section above are available under original licenses.

Book version

1.0

See Version_changes.md to track changes across book versions.

Introduction

Back in 2007, I had a rough beginning as a design engineer at a semiconductor company in terms of software tools. Linux command line, Vim and Perl were all new to me. I distinctly remember progressing from `dd` (delete current line) to `d↓` (delete current line as well as the line below) and feeling happy that it reduced time spent on editing. Since I was learning on the job, I didn't know about count prefix or the various ways I could've deleted all the lines from the beginning of the file to the line containing a specific phrase. Or even better, I could've automated editing multiple files if I had been familiar with `sed` or progressed that far with Perl.

I also remember that we got a two-sided printed cheatsheet that we kept pinned to our cabins. That was one of the ways I kept adding commands to my repertoire. But, I didn't have a good insight to Vim's philosophy and I didn't know how to apply many of the cheatsheet commands. At some point, I decided to read the Vim book by Steve Oualline and that helped a lot, but it was also too long and comprehensive for me to read it all. My memory is hazy after that, and I don't recall what other resources I used. However, I'm sure I didn't effectively utilize built-in help. Nor did I know about stackoverflow or /r/vim until after I left my job in 2014.

Still, I knew enough to conduct a few Vim learning sessions for my colleagues. That came in handy when I got chances to teach Vim as part of scripting course for college students. From 2016 to 2018, I started maintaining my tutorials on Linux command line, Vim and scripting languages as GitHub repos. As you might guess, I then started polishing these materials and published them as ebooks. This is an ongoing process, with **Vim Reference Guide** being the twelfth ebook.

Why Vim?

You've probably already heard that Vim is a text editor, powerful one at that. Vim's editing features feel like a programming language and you can customize the editor using scripting languages. Apart from plethora of editing commands and support for regular expressions, you can also incorporate external commands. To sum it up, most editing tasks can be managed from within Vim itself instead of having to write a script.

Now, you might wonder, what is all this need for complicated editing features? Why does a text editor require programming capabilities? Why is there even a requirement to *learn* how to use a text editor? Isn't it enough to have the ability to enter text, use Backspace/Delete/Home/End/Arrow/etc, menu and toolbar, some shortcuts, a search and replace feature and so on? A simple and short answer — to reduce repetitive manual task.

What I like the most about Vim:

- Lightweight and fast
- Modal editing helps me to think logically based on the type of editing task
- Composing commands and the ability to record them for future use
- Settings customization and creating new commands
- Integration with shell commands

There's a huge ecosystem of plugins, packages and colorschemes as well, but I haven't used them much. I've used Vim for a long time, but not really a power user. I prefer using GVim, tab pages, mouse, arrow keys, etc. So, if you come across tutorials and books suggesting you should avoid using them, remember that they are subjective preferences.

Here are some more opinions by those who enjoy using Vim:

- stackoverflow: What are the benefits of learning Vim?
- Why Vim
- Vim Creep

 Should everybody use Vim? Is it suitable for all kinds of editing tasks? I'd say no. There are plenty of other well established text editors and new ones are coming up all the time. The learning curve isn't worth it for everybody. If Vim wasn't being used at job, I probably wouldn't have bothered with it. Don't use Vim for the wrong reasons article discusses this topic in more detail.

Installation

I use the following command on Ubuntu (a Linux distribution):

```
sudo apt install vim vim-gui-common
```

- :h usr_90.txt — user manual for installation on different platforms, common issues, upgrading, uninstallation, etc
- vi.stackexchange: How can I get a newer version of Vim? — building from source, using distribution packages, etc

 See also https://github.com/vim/vim for source code and other details.

Ice Breaker

Open a terminal and follow these steps:

- `gvim ip.txt` opens a file named `ip.txt` for editing
  - You can also use `vim` if you prefer terminal instead of GUI, or if `gvim` is not available
- Press `i` key (yes, the lowercase alphabet `i`, not some alien key)
- Start typing, for example `What a weird editor`
- Press `Esc` key
- Press `:` key
- Type `wq`
- Press `Enter` key
- `cat ip.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 Vim newbie. Here's a brief explanation of the above steps:

- Vim is a **modal editor**. You have to be aware which mode you are in and use commands or type text accordingly
- When you first launch Vim, it starts in **Normal mode** (primarily used for editing and moving around)
- Pressing `i` key is one of the ways to enter **Insert mode** (where you type the text you want to save in a file)
- After you've entered the text, you need to save the file. To do so, you have to go back to Normal mode first by pressing the `Esc` key

- Then, you have to go to yet another mode! Pressing `:` key brings up the **Command-line mode** and awaits further instruction
- `wq` is a combination of **write** and **quit** commands
  - use `wq ip.txt` if you forgot to specify the filename while launching Vim, or perhaps if you opened Vim from Start menu instead of a terminal
- `Enter` key completes the command you've typed

If you launched GVim, you'll likely have **Menu** and **Tool** bars, which would've helped with operations like saving, quitting, etc. Nothing wrong with using them, but this book will not discuss those operations. In fact, you'll learn how to configure Vim to hide them in the Customizing Vim chapter.

Don't proceed any further if you aren't comfortable with the above steps. Take help of youtube videos if you must. Master this basic procedure and you will be ready for Vim awesomeness that'll be discussed in the coming sections and chapters.

 Material presented here is based on GVim (GUI), which has a few subtle differences compared to Vim (TUI). See this stackoverflow thread for more details.

 Options and details related to opening Vim from the command line will be discussed in the CLI options chapter.

Built-in tutor

- `gvimtutor` command that opens a tutorial session with lessons to get started with Vim
  - don't worry if something goes wrong as you'll be working with a temporary file
  - use `vimtutor` if `gvim` is not available
  - **pro-tip**: go through this short tutorial multiple times, spread over multiple days

 Next step is :h usr_02.txt, which provides enough information about editing files with Vim.

Built-in help

Vim comes with comprehensive user and reference manuals. The user manual reads like a text book and reference manual has more details than you are likely to need. There's also an online site with these help contents, which will be linked as appropriate throughout this book.

- You can access built-in help in several ways:
  - type `:help` from Normal mode (or just the `:h` short form)
  - GVim has a `Help` menu
  - press `F1` key from Normal mode
- :h usr_toc.txt table of contents for User Manual
  - Task oriented explanations, from simple to complex. Reads from start to end like a book
- :h reference_toc table of contents for Reference Manual
  - Precise description of how everything in Vim works

- :h quickref quick reference guide
- :h help-summary effectively using help depending on the topic/feature you are interested in
    - See also vi.stackexchange: guideline to use help

Here's a neat table from :h help-context:

WHAT	PREPEND	EXAMPLE
Normal mode command		`:help x`
Visual mode command	`v_`	`:help v_u`
Insert mode command	`i_`	`:help i_<Esc>`
Command-line command	`:`	`:help :quit`
Command-line editing	`c_`	`:help c_<Del>`
Vim command argument	`-`	`:help -r`
Option	`'`	`:help 'textwidth'`
Regular expression	`/`	`:help /[`

 You can go through a copy of the documentation online at https://vimhelp.org/. As shown above, all the `:h` hints in this book will also be linked to the appropriate online help section.

Vim learning resources

As mentioned in the Preface chapter, this **Vim Reference Guide** is more like a cheatsheet instead of a typical book for learning Vim. In addition to built-in features already mentioned in the previous sections, here are some resources you can use:

Tutorials

- Vim primer — learn Vim in a way that will stay with you for life
- Vim galore — everything you need to know about Vim
- Learn Vim progressively — short introduction that covers a lot
- Vim from the ground up — article series for beginners to expert users

Books

- Practical Vim
- Mastering Vim Quickly
- Learn Vim (the Smart Way)

Interactive

- OpenVim — interactive tutorial
- Vim Adventures — learn Vim by playing a game
- vimmer.io — master Vim from the comfort of your web browser
- vim.so — interactive lessons designed to help you get better at Vim faster

 See my Vim curated list for a more complete list of learning resources, cheatsheets, tips, tricks, forums, etc.

Modes of Operation

As mentioned earlier, Vim is a **modal editor**. This book will mainly discuss these four modes:

- Insert mode
- Normal mode
- Visual mode
- Command-line mode

This section provides a brief description for these modes. Separate chapters will discuss their features in more detail.

 For a complete list of modes, see :h vim-modes-intro and :h mode-switching.

Insert mode

This is the mode where the required text is typed. There are also commands available for moving around, deleting, autocompletion, etc.

Pressing the `Esc` key takes you back to Normal mode.

Normal mode

This is the default mode when Vim is opened. This mode is used to run commands for operations like cut, copy, paste, recording, moving around, etc. This is also known as the Command mode.

Visual mode

Visual mode is used to edit text by selecting them first. Selection can either be done using mouse or using visual commands.

Pressing the `Esc` key takes you back to the Normal mode.

Command-line mode

This mode is used to perform file operations like save, quit, search, replace, execute shell commands, etc. Any operation is completed by pressing the `Enter` key after which the mode changes back to the Normal mode. The `Esc` key can be used to ignore whatever is typed and return to the Normal mode.

The space at the end of the file used for this mode is referred to as Command-line area. It is usually a single line, but can expand for cases like auto completion, shell commands, etc.

Identifying current mode

- In Insert mode, you get a blinking `|` cursor
  - also, `-- INSERT --` can be seen on the left hand side of the Command-line area
- In Normal mode, you get a blinking rectangular block cursor, something like this
- In Visual mode, the Command-line area shows `-- VISUAL --` or `-- VISUAL LINE --` or `-- VISUAL BLOCK --` according to the visual command used
- In Command-line mode, the cursor is of course in the Command-line area

 See also :h 'showmode' setting.

Vim philosophy and features


Commands discussed in this section will be covered again in later chapters. The idea here is to give you a brief introduction to modes and notable Vim features. See also:

- Best introduction to Vi and its core editing concepts explained as a language (this stackoverflow thread also has numerous Vim tips and tricks)
- Seven habits of effective text editing

As a programmer, I love how composable Vim commands are. For example, you can do this in Normal mode:

- `dG` delete from the current line to the end of the file
  - where `d` is the delete command awaiting further instruction
  - and `G` is a motion command to move to the last line of the file
- `yG` copy from the current line to the end of the file
  - where `y` is the yank (copy) command awaiting further instruction

Most Normal mode commands accept a count prefix. For example:

- `3p` paste the copied content three times
- `5x` delete the character under the cursor and 4 characters to its right (total 5 characters)
- `3` followed by `Ctrl` + `a` add `3` to the number under the cursor

There are context aware operations too. For example:

- `diw` delete a word regardless of where the cursor is on that word
- `ya}` copy all characters within `{}` including the `{}` characters

If you are a fan of selecting text before editing them, you can use the Visual mode. There are several commands you can use to start Visual mode. If enabled, you can even use mouse to select the required portions.

- `~` invert the case of the visually selected text (i.e. lowercase becomes UPPERCASE and vice versa)
- `g` followed by `Ctrl` + `a` for visually selected lines, increment number by `1` for the first line, by `2` for the second line, by `3` for the third line and so on

The Command-line mode is useful for file level operations, search and replace, changing Vim configurations, talking to external commands and so on.

- `/searchpattern` search the given pattern in the forward direction
- `:g/call/d` delete all lines containing `call`
- `:g/cat/ s/animal/mammal/g` replace `animal` with `mammal` only for the lines containing `cat`
- `:3,8! sort` sort only lines `3` to `8` (uses external command `sort`)
- `:set incsearch` highlights current match as you type the search pattern

Changes to Vim configurations from the Command-line mode are applicable only for that particular session. You can use the `vimrc` file to load the settings at startup.

- `colorscheme murphy` use a dark theme
- `set tabstop=4` width for the tab character (default is `8`)
- `nnoremap <F5> :%y+<CR>` map `F5` key to copy everything to system clipboard in Normal mode

- `inoreabbrev teh the` automatically correct `teh` to `the` in Insert mode

There are many more Vim features that'd help you with text processing and customizing the editor to your needs, some of which you'll get to know in the coming chapters.

Finally, you can apply your Vim skills elsewhere too. Vim-like features have been adopted across a huge variety of applications and plugins, for example:

- less command supports vim-like navigation
- Extensible vi layer for Emacs
- Vimium (browser extension), qutebrowser (keyboard-driven browser with vim-like navigation), etc
- JetBrains IdeaVim, VSCodeVim, etc
- Huge list of Vim-like applications and plugins

Vim's history

See Where Vim Came From if you are interested in knowing Vim's history that traces back to the 1960s with `qed`, `ed`, etc.

Chapters

Here's the list of remaining chapters:

- Insert mode
- Normal mode
- Command-line mode
- Visual mode
- Regular Expressions
- Macro
- Customizing Vim
- CLI options

Insert mode

This is the mode where the required text is typed. There are also commands available for moving around, deleting, autocompletion, etc.

Documentation links:

- :h usr_24.txt — overview of the most often used Insert mode commands
- :h insert.txt — reference manual for Insert and Replace mode

Recall that you need to add `i_` prefix for built-in help on Insert mode commands, for example :h i_CTRL-P.

Motion keys and commands

- `←` move left by one character within the current line
- `→` move right by one character within the current line
- `↓` move down by one line
- `↑` move up by one line
- `Ctrl` + `←` and `Ctrl` + `→` move to the start of the current/previous and next word respectively
  - From :h word "A word consists of a sequence of letters, digits and underscores, or a sequence of other non-blank characters, separated with white space"
  - you can also use `Shift` key instead of `Ctrl`
- `Home` move to the start of the line
- `End` move to the end of the line
- `PageUp` move up by one screen
- `PageDown` move down by one screen
- `Ctrl` + `Home` move to the start of the file
- `Ctrl` + `End` move to the end of the file

You can use the `whichwrap` setting (`ww` for short) to allow `←` and `→` arrow keys to cross lines. For example, `:set ww+=[,]` tells Vim to allow left and right arrow keys to move across lines in Insert mode (`+=` is used here to preserve existing options for the `whichwrap` setting).

Deleting

- `Delete` delete the character after the cursor
- `Backspace` delete the character before the cursor
  - `Ctrl` + `h` also deletes the character before the cursor
- `Ctrl` + `w` delete characters before the cursor until start of a word
  - From :h word "A word consists of a sequence of letters, digits and underscores, or a sequence of other non-blank characters, separated with white space"
- `Ctrl` + `u` delete all the characters before the cursor in the current line, preserves indentation if any

Autocomplete word

- `Ctrl` + `p` autocomplete word based on matching words in the backward direction
- `Ctrl` + `n` autocomplete word based on matching words in the forward direction

If more than one word matches, they are displayed using a popup menu. You can use `↑` / `↓` arrow keys or `Ctrl` + `p` / `Ctrl` + `n` to move through this list.

With multiple matches, you'll notice that the first match is automatically inserted and moving through the list doesn't change the text that was inserted. You'll have to press `Ctrl` + `y` or `Enter` key to choose a different completion text. If you were satisfied with the first match, typing any character will make the popup menu disappear and insert whatever character you had typed. Or press `Esc` to select the first match and go to Normal mode.

Autocomplete line

- `Ctrl` + `x` followed by `Ctrl` + `l` autocomplete line based on matching lines in the backward direction

If more than one line matches, they are displayed using a popup menu. You can use `↑` / `↓` arrow keys or `Ctrl` + `p` / `Ctrl` + `n` to move through this list. You can also use `Ctrl` + `l` to move up the list.

Autocomplete assist

- `Ctrl` + `e` cancels autocomplete
  - you'll retain the text you had typed before invoking autocomplete
- `Ctrl` + `y` or `Enter` change the autocompletion text to the currently selected item from the popup menu

See :h ins-completion for more details and other autocomplete features. See :h 'complete' setting for customizing autocomplete commands.

Execute single Normal mode command

- `Ctrl` + `o` execute a Normal mode command and return to Insert mode
  - `Ctrl` + `o` followed by `A` moves the cursor to the end of the current line
  - `Ctrl` + `o` followed by `3j` moves the cursor three lines below

Indenting

- `Ctrl` + `t` indent the current line
- `Ctrl` + `d` unindent the current line
- `0` followed by `Ctrl` + `d` deletes all indentation in the current line


Indentation depends on the `shiftwidth` setting. See :h 'shiftwidth' for more details.

Insert register contents

- `Ctrl` + `r` helps to insert the contents of a register
  - `Ctrl` + `r` followed by `%` inserts the current file name
  - `Ctrl` + `r` followed by `a` inserts the content of `"a` register
- `Ctrl` + `r` followed by `=` allows you to insert the result of an expression
  - `Ctrl` + `r` followed by `=12+1012` and then `Enter` key inserts `1024`
  - `Ctrl` + `r` followed by `=strftime("%Y/%m/%d")` and then `Enter` key inserts the current date, for example `2022/02/02`

From :h 24.6:

If the register contains characters such as `<BS>` or other special characters, they are interpreted as if they had been typed from the keyboard. If you do not want this to happen (you really want the `<BS>` to be inserted in the text), use the command `CTRL-R CTRL-R {register}` .


Registers will be discussed in more details in the Normal mode chapter. See :h usr_41.txt to get started with Vim script.

Insert special characters

- `Ctrl` + `v` helps to insert special keys literally
  - `Ctrl` + `v` followed by `Esc` gives `^[`
  - `Ctrl` + `v` followed by `Enter` gives `^M`
- `Ctrl` + `q` alias for `Ctrl` + `v` , helps if it is mapped to do something else


You'll see a practical usage of this command in Macro chapter. You can also specify the character using decimal, octal or hexadecimal formats. See :h 24.8 for more details.

Insert digraphs

- `Ctrl` + `k` helps to insert digraphs (two character combinations used to represent a single character, such characters are usually not available on the keyboard)
  - `Ctrl` + `k` followed by `Ye` gives `¥`


You can use `:digraphs` to get a list of combinations and their respective characters. You can also define your own combinations using the `:digraph` command. See :h 24.9 for more details.

Normal mode

Make sure you are in Normal mode before trying out the commands in this chapter. Press `Esc` key to return to Normal mode from other modes. Press `Esc` again if needed.

Documentation links:

- :h usr_03.txt — moving around
- :h usr_04.txt — making small changes
- :h motion.txt — reference manual for motion commands
- :h change.txt — reference manual for commands that delete or change text
- :h undo.txt — reference manual for undo and redo

Arrow motions

The four arrow keys can be used in Vim to move around, just like other text editors. Vim also maps them to four characters in Normal mode.

- `h` or `←` move left by one character within the current line
- `j` or `↓` move down by one line
- `k` or `↑` move up by one line
- `l` or `→` move right by one character within the current line

Vim offers plenty of other motion commands. Several sections will discuss them later in this chapter.

 You can use the `whichwrap` setting to allow `←` and `→` arrow keys to cross lines. For example, `:set ww+=<,>` tells Vim to allow left and right arrow keys to move across lines in Normal and Visual modes. Add `h` and `l` to this comma separated list if want those commands to cross lines as well.

Cut

There are various ways to delete text. All of these commands can be prefixed with a **count** value. `d` and `c` commands can accept any motion commands. Only arrow motion examples are shown in this section, many more variations will be discussed later in this chapter.

- `dd` delete the current line
- `2dd` delete the current line and the line below it (total 2 lines)
  - `dj` or `d↓` can also be used
- `10dd` delete the current line and 9 lines below it (total 10 lines)
- `dk` delete the current line and the line above it
  - `d↑` can also be used
- `d3k` delete the current line and 3 lines above it (total 4 lines)
  - `3dk` can also be used
- `D` delete from the current character to the end of line (same as `d$`, where `$` is a motion command to move to the end of line)
- `x` delete only the current character under the cursor (same as `dl`)
- `5x` delete the character under the cursor and 4 characters to its right (total 5 characters)
- `X` delete only the current character before the cursor (same as `dh`)

    - if the cursor is on the first character in the line, deleting would depend on the `whichwrap` setting as discussed earlier
- `5X` delete 5 characters to the left of the cursor
- `cc` delete the current line and change to Insert mode
    - indentation will be preserved depending on the `autoindent` setting
- `4cc` delete the current line and 3 lines below it and change to Insert mode (total 4 lines)
- `C` delete from the current character to the end of line and change to Insert mode
- `s` delete only the character under the cursor and change to Insert mode (same as `cl`)
- `5s` delete the character under the cursor and 4 characters to its right and change to Insert mode (total 5 characters)
- `S` delete the current line and change to Insert mode (same as `cc`)
    - indentation will be preserved depending on the `autoindent` setting

 You can also select text (using mouse or visual commands) and then press `d` or `x` or `c` or `s` to delete the selected portions. Example usage will be discussed in the Visual mode chapter.

 The deleted portions can also be pasted elsewhere using the paste command (discussed later in this chapter).

Copy

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

- `yy` copy the current line
    - `Y` also copies the current line
- `y$` copy from the current character to the end of line
    - use `:nnoremap Y y$` if you want `Y` to behave similarly to the `D` command
- `2yy` copy the current line and the line below it (total 2 lines)
    - `yj` and `y↓` can also be used
- `10yy` copy the current line and 9 lines below it (total 10 lines)
- `yk` copy the current line and the line above it
    - `y↑` can also be used

 You can also select text (using mouse or visual commands) and then press `y` to copy them.

Paste

The **put** (paste) command `p` is used after cut or copy operations.

- `p` paste the copied content once
    - If the copied text was line based, content is pasted **below** the current line
    - If the copied text was part of a line, content is pasted to the **right** of the cursor
- `P` paste the copied content once
    - If the copied text was line based, content is pasted **above** the current line

    - If the copied text was part of a line, content is pasted to the **left** of the cursor
- `3p` and `3P` paste the copied content three times
- `]p` paste the copied content like `p` command, but changes the indentation level to match the current line
- `[p` paste the copied content like `P` command, but changes the indentation level to match the current line

Undo

- `u` undo last change
    - press `u` again for further undos
- `U` undo latest changes on last edited line
    - press `U` again to redo changes


See :h 32.3 for details on `g-` and `g+` commands that you can use to undo branches.

Redo

- `Ctrl` + `r` redo a change undone by `u`
- `U` redo changes undone by `U`

Replace characters

Often, you just need to change one character. For example, changing `i` to `j`, `2` to `4` and so on.

- `rj` replace the character under the cursor with `j`
- `ry` replace the character under the cursor with `y`
- `3ra` replace the character under cursor as well as the two characters to the right with `aaa`
    - no changes will be made if there aren't sufficient characters to match

To replace multiple characters with different characters, use `R`.

- `Rlion` followed by `Esc` replace the character under cursor and three characters to the right with `lion`
    - `Esc` key marks the completion of `R` command
    - `Backspace` key will act as an undo command to give back the character that was replaced
    - if you are replacing at the end of a line, the line will be automatically extended if needed

The advantage of `r` and `R` commands is that you remain in the Normal mode, without needing to switch to Insert mode and back.

Repeat a change

- `.` the dot command repeats the last change
- If the last change was `2dd` (delete current line and the line below), dot key will repeat `2dd`

- If the last change was `5x` (delete current character and four characters to the right), dot key will repeat `5x`
- If the last change was `C123<Esc>` and dot key is pressed, it will clear from the current character to the end of the line, insert `123` and go back to Normal mode

From :h 4.3:

> The `.` command works for all changes you make, except for `u` (undo), `CTRL-R` (redo) and commands that start with a colon (`:`).


See :h repeat.txt for complex repeats, using Vim scripts, etc.

Open new line

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


Indentation of the new line depends on `autoindent` , `smartindent` and `cindent` settings.

Moving within current line

- `0` move to the beginning of the current line (i.e. column number 1)
  - you can also use the `Home` key
- `^` move to the beginning of the first non-blank character of the current line (useful for indented lines)
- `$` move to the end of the current line
  - you can also use the `End` key
  - `3$` move to the end of 2 lines below the current line
- `g_` move to the last non-blank character of the current line
- `3|` move to the third column character
  - `|` is same as `0` or `1|`

Moving within long lines that are spread over multiple screen lines:

- `g0` move to the beginning of the current screen line
- `g^` move to the first non-blank character of the current screen line
- `g$` move to the end of the current screen line
- `gj` move down by one screen line, prefix a count to move down by that many screen lines
- `gk` move up by one screen line, prefix a count to move up by that many screen lines
- `gm` move to the middle of the current screen line
  - **Note** that this is based on the screen width, not the number of characters in the line!
- `gM` move to the middle of the current line
  - **Note** that this is based on the total number of characters in the line


See :h left-right-motions for more details.

Character motions

These commands allow you to move based on single character search, **within the current line only**.

- `f(` move forward to the next occurrence of character `(`
- `fb` move forward to the next occurrence of character `b`
- `3f"` move forward to the third occurrence of character `"`
- `t;` move forward to the character just before `;`
- `3tx` move forward to the character just before the third occurrence of character `x`
- `Fa` move backward to the character `a`
- `Ta` move backward to the character just after `a`
- `;` repeat previous `f` or `F` or `t` or `T` motion in the same direction
- `,` repeat previous `f` or `F` or `t` or `T` motion in the opposite direction
    - for example, `tc` becomes `Tc` and vice versa

Note that the previously used count prefix wouldn't be repeated with `;` or `,` commands, but you can use a new count prefix. If you pressed a wrong motion command, use the `Esc` key to abandon the search instead of continuing with the wrongly chosen command.

Word motions

Definitions from :h word and :h WORD are quoted below to explain the difference between **word** and **WORD**.

word A word consists of a sequence of letters, digits and underscores, or a sequence of other non-blank characters, separated with white space (spaces, tabs, `<EOL>`). This can be changed with the `iskeyword` option. An empty line is also considered to be a word.

WORD A WORD consists of a sequence of non-blank characters, separated with white space. An empty line is also considered to be a WORD.

- `w` move to the start of the next word
- `W` move to the start of the next WORD
    - `192.1.168.43;hello` is considered as a single WORD, but has multiple words
- `b` move to the beginning of the current word if the cursor is *not* at the start of word. Otherwise, move to the beginning of the previous word
- `B` move to the beginning of the current WORD if the cursor is *not* at the start of WORD. Otherwise, move to the beginning of the previous WORD
- `e` move to the end of the current word if cursor is *not* at the end of word. Otherwise, move to the end of next word
- `E` move to the end of the current WORD if cursor is *not* at the end of WORD. Otherwise, move to the end of next WORD
- `ge` move to the end of the previous word
- `gE` move to the end of the previous WORD
- `3w` move 3 words forward
    - Similarly, a number can be prefixed for all the other commands mentioned above

 All of these motions will work across lines. For example, if the cursor is on the last word of a line, pressing `w` will move to the start of the first word in the next line.

Text object motions

- `(` move backward a sentence
- `)` move forward a sentence
- `{` move backward a paragraph
- `}` move forward a paragraph

 More such text objects will be discussed later under the Context editing section. See :h object-motions for a complete list of such motions.

Moving within the current file

- `gg` move to the first non-blank character of the first line
- `G` move to the first non-blank character of the last line
- `5G` move to the first non-blank character of the fifth line
  - As an alternative, you can use `:5` followed by `Enter` key (Command-line mode)
- `50%` move to halfway point
  - you can use other percentages as needed
- `%` move to matching pair of brackets like `()` , `{}` and `[]`
  - This will work across lines and nesting is taken into consideration as well
  - If the cursor is on a non-bracket character and a bracket character is present later in the line, the `%` command will move to the matching pair of that character (which could be present in some other line too)
  - Use the `matchpairs` option to customize the matching pairs. For example, `:set matchpairs+=<:>` will match `<>` as well

 It is also possible to match a pair of keywords like HTML tags, if-else, etc with `%` . See :h matchit-install for details.

Moving within the visible window

- `H` move to the first non-blank character of the top (home) line of the visible window
- `M` move to the first non-blank character of the middle line of the visible window
- `L` move to the first non-blank character of the bottom (low) line of the visible window

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` followed by **Mouse Scroll** scroll one page forward or backward

Reposition the current line

- `Ctrl` + `e` scroll up by a line
- `Ctrl` + `y` scroll down by a line
- `zz` reposition the current line to the middle of the visible window
  - useful to see context around lines that are nearer to the top/bottom of the visible window
- `zt` reposition the current line to the top of the visible window
- `zb` reposition the current line to the bottom of the visible window


See :h 'scrolloff' option if you want to always show context around the current line.

Indenting

- `>` and `<` indent commands, waits for motion commands similar to `d` or `y`
- `>>` indent the current line
- `3>>` indent the current line and two lines below (same as `>2j`)
- `>k` indent the current line and the line above
- `>}` indent till the end of the paragraph
- `<<` unindent the current line
- `5<<` unindent the current line and four lines below (same as `<4j`)
- `<2k` unindent the current line and two lines above
- `=` auto indent code, use motion commands to indicate the portion to be indented
  - `=4j` auto indent the current line and four lines below
  - `=ip` auto indent the current paragraph (you'll learn about `ip` later in the Context editing section)


Indentation depends on the `shiftwidth` setting. See :h shift-left-right, :h = and :h 'shiftwidth' for more details.


You can indent/unindent the same selection multiple times using a number prefix in the Visual mode.

Mark frequently used locations

- `ma` mark location in the file using the alphabet `a`
  - you can use any of the 26 alphabets
  - use lowercase alphabets to work within the current file
  - use uppercase alphabets to work from any file
  - `:marks` will show a list of the existing marks
- `` `a `` move to the exact location marked by `a`
- `'a` move to the first non-blank character of the line marked by `a`
- `'A` move to the first non-blank character of the line marked by `A` (this will work for any file where the mark was set)
- `d`a` delete from the current character to the character marked by `a`
  - marks can be paired with any command that accept motions like `d` , `y` , `>` , etc


Motion commands that take you across lines (for example, `10G`) will automatically save the location you jumped from in the default `` ` `` mark. You can move back to that exact location using `` `` `` or the first non-blank character using `` '' ``. Note that the arrow and word motions aren't considered for the default mark even if they move across lines.


See :h mark-motions for more ways to use marks.

Jumping back and forth

This is helpful if you are moving around often while editing a large file, moving between different buffers, etc. From :h jump-motions:

The following commands are **jump** commands: `'`, `` ` ``, `G`, `/`, `?`, `n`, `N`, `%`, `(`, `)`, `[[`, `]]`, `{`, `}`, `:s`, `:tag`, `L`, `M`, `H` and the commands that start editing a new file.
When making a **change** the cursor position is remembered. One position is remembered for every change that can be undone, unless it is close to a previous change.

- `Ctrl` + `o` navigate to the previous location in the jump list (`o` as in old)
- `Ctrl` + `i` navigate to the next location in the jump list (`i` and `o` are usually next to each other)
- `g;` go to the previous change location
- `g,` go to the newer change location
- `gi` place the cursor at the same position where it was left last time in the Insert mode


Use `:jumps` and `:changes` to view the jump and change lists respectively. See :h jump-motions for more details.

Edit with motion

From :h usr_03.txt:

You first type an operator command. For example, `d` is the delete operator. Then you type a motion command like `4l` or `w`. This way you can operate on any text you can move over.

- `dG` delete from the current line to the end of the file
- `dgg` delete from the current line to the beginning of the file
- `` d`a `` delete from the current character up to the location marked by `a`
- `d%` delete up to the matching pairs for `()`, `{}`, `[]`, etc
- `ce` delete till the end of word and change to Insert mode
  - `cw` will also work the same as `ce`, this inconsistency is based on **Vi** behavior
  - use `:nnoremap cw dwi` if you don't want the old behavior
- `yl` copy the character under the cursor

- `yfc` copy from the character under the cursor up to the next occurrence of `c` in the same line
- `d)` delete up to the end of the sentence

From :h usr_03.txt:

> Whether the character under the cursor is included depends on the command you used to move to that character. The reference manual calls this "exclusive" when the character isn't included and "inclusive" when it is. The `$` command moves to the end of a line. The `d$` command deletes from the cursor to the end of the line. This is an inclusive motion, thus the last character of the line is included in the delete operation.

Context editing

You have seen examples for combining motions such as `w`, `%` and `f` with editing commands like `d`, `c` and `y`. Such combination of commands require precise positioning to be effective.

Vim also provides a list of handy context based options to make certain editing use cases easier using the `i` and `a` text object selections. You can easily remember the difference between these two options by thinking `i` as **inner** and `a` as **around**.

- `diw` delete a word regardless of where the cursor is on that word
  - Equivalent to using `de` when the cursor is on the first character of the word
- `diW` delete a WORD regardless of where the cursor is on that WORD
- `daw` delete a word regardless of where the cursor is on that word as well as a space character to the left/right of the word depending on its position in the current 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 as well as a space character to the left/right
- `cip` delete a paragraph regardless of where the cursor is on that paragraph and change to Insert mode
- `dit` delete all characters within HTML/XML tags, nesting is taken care as well
  - see :h tag-blocks for details about corner cases
- `di"` delete all characters within a pair of double quotes, regardless of where the cursor is within the quotes
- `da'` delete all characters within a pair of single quotes along with the quote characters
- `ci(` delete all characters within `()` and change to Insert mode
  - Works even if the parenthesis are spread over multiple lines, nesting is taken care as well
- `ya}` copy all characters within `{}` including the `{}` characters
  - Works even if the braces are spread over multiple lines, nesting is taken care as well


You can use a count prefix for nested cases. For example, `c2i{` will clear the inner braces (including the braces, and this could be nested too) and then only the text between braces for the next level.


See :h text-objects for more details.

Named registers

You can use lowercase alphabets `a-z` to save some content for future use. You can also append some more content to those registers by using the corresponding uppercase alphabets `A-Z` at a later stage.

- `"ayy` copy the current line to the `"a` register
- `"Ayj` append the current line and the line below to the `"a` register
  - `"ayy` followed by `"Ayj` will result in total three lines in the `"a` register
- `"ap` paste content from the `"a` register
- `"eyiw` copy word under the cursor to the `"e` register


You can use `:reg` (short for `:registers`) to view the contents of the registers. Specifying one or more characters (next to each other as a single string) will display contents only for those registers.


The named registers are also used for saving macros (will be discussed in the Macro chapter). You can record an empty macro to clear the contents, for example `qbq` clears the `"b` register.

Special registers

Vim has nine other types of registers for different use cases. Here are some of them:

- `"` all yanked/deleted text is stored in this register
  - So, `p` command is same as specifying `""p`
- `"0` yanked text is stored in this register
  - A possible use case: yank some content, delete something else and then paste the yanked content using `"0p`
- `"1` to `"9` deleted contents are stored in these registers and get shifted with each new deletion
  - `"1p` paste the contents of last deletion
  - `"2p` paste the contents of last but one deletion
- `"+` this register is used to work with the system clipboard contents
  - `gg"+yG` copy entire file contents to the clipboard
  - `"+p` paste content from the clipboard
- `"*` this register stores visually selected text
  - contents of this register can be pasted using **middle mouse button click** or `"*p`
- `"_` black hole register, when you want to delete something without saving it anywhere

Further reading

- :h registers
- stackoverflow: How to use Vim registers
- stackoverflow: Using registers on Command-line mode
- Advanced Vim registers

Search word nearest to the cursor

- `*` searches the word nearest to the cursor in the forward direction (matches only the whole word)
  - `Shift` followed by **left mouse click** can also be used in GVim
- `g*` searches the word nearest to the cursor in the forward direction (matches as part of another word as well)
  - for example, if you apply this command on the word `the`, you'll also get matches for `them`, `lather`, etc
- `#` searches the word nearest to the cursor in the backward direction (matches only the whole word)
- `g#` searches the word nearest to the cursor in the backward direction (matches as part of another word as well)


You can also provide a count prefix to these commands.

Join lines

- `J` joins the current line and the next line
  - the deleted `<EOL>` character is replaced with a space (unless there are trailing spaces or the next line starts with a `)` character)
  - indentation from the lines being joined are removed, *except the current line*
- `3J` joins the current line and next two lines with one space in between the lines
- `gJ` joins the current line and the next line
  - `<EOL>` character is deleted (space character won't be added)
  - indentation won't be removed


`joinspaces`, `cpoptions` and `formatoptions` settings will affect the behavior of these commands. See :h J and scroll down for more details.

Changing case

- `~` invert the case of the character under the cursor (i.e. lowercase becomes UPPERCASE and vice versa)
- `g~` followed by motion command to invert the case of those characters
  - for example: `g~e`, `g~$`, `g~iw`, etc
- `gu` followed by motion command to change the case of those characters to lowercase
  - for example: `gue`, `gu$`, `guiw`, etc
- `gU` followed by motion command to change the case of those characters to UPPERCASE
  - for example: `gUe`, `gU$`, `gUiw`, etc


You can also provide a count prefix to these commands.

Increment and Decrement numbers

- `Ctrl` + `a` increment the number under the cursor or the first occurrence of a number to the right of the cursor
- `Ctrl` + `x` decrement the number under the cursor or the first occurrence of a number to the right of the cursor
- `3` followed by `Ctrl` + `a` adds `3` to the number
- `1000` followed by `Ctrl` + `x` subtracts `1000` from the number


Numbers prefixed with `0`, `0x` and `0b` will be treated as octal, hexadecimal and binary respectively (you can also use uppercase for `x` and `b`).


Decimal numbers prefixed with `-` will be treated as negative numbers. For example, using `Ctrl` + `a` on `-100` will give you `-99`. While this is handy, this trips me up often when dealing with date formats like 2021-12-07.

Miscellaneous

- `gf` opens a file using the path under the cursor
  - See :h gf and :h suffixesadd for more details
  - See :h window-tag if you want to open the file under the cursor as a split window, new tab and other usecases
- `Ctrl` + `g` display file information like name, number of lines, etc at the bottom of the screen
  - See :h CTRL-G for more details and related commands
- `g` followed by `Ctrl` + `g` display information about the current location of the cursor (column, line, word, character and byte counts)
- `ga` shows codepoint value of the character under the cursor in decimal, octal and hexadecimal formats
- `g?` followed by motion command to change those characters with `rot13` transformation
  - `g?e` on start of `hello` word will change it to `uryyb`
  - `g?e` on start of `uryyb` word will change it to `hello`

Switching modes

Normal to Insert mode

- `i` place the cursor to the left of the current character (insert)
- `a` place the cursor to the right of the current character (append)
- `I` place the cursor before the first non-blank character of the line (helpful for indented lines)
- `gI` place the cursor before the first column of the line
- `gi` place the cursor at the same position where it was left last time in the Insert mode
- `A` place the cursor at the end of the line
- `o` open a new line below the current line and change to Insert mode
- `O` open a new line above the current line and change to Insert mode
- `s` delete character under the cursor and change to Insert mode
- `S` delete the current line and change to Insert mode

    - `cc` can also be used
    - indentation will be preserved depending on the `autoindent` setting
- `C` delete from the current character to the end of line and change to Insert mode

Normal to Command-line mode

- `:` change to Command-line mode, awaits further commands
- `/` change to Command-line mode for searching in the forward direction
- `?` change to Command-line mode for searching in the backward direction

Normal to Visual mode

- `v` visually select the current character
- `V` visually select the current line
- `Ctrl` + `v` visually select column
- `gv` select previously highlighted visual area

 See :h mode-switching for a complete table.