A Byte of Vim

Swaroop C H

Table of Contents

A Byte of Vim

"A Byte of Vim" is a book which aims to help you to learn how to use the Vim editor (version 7), even if all you know is how to use the computer keyboard.

The first part of this book is meant for new users who want to understand what Vim is and learn how to use it.

The second part of this book is for people who already know how to use Vim and want to learn about features that make Vim so powerful, such as windows and tabs, personal information management, making it a programmer's editor, how to extend Vim with your own plugins, and more.

Who reads A Byte of Vim?

> Thanks to A Byte of Vim, I have learnt how to use vim. Thank you very much for this excellent book! ;) -- Camille L (France)

> Well done!!! I've been only using vim for like 2/3 weeks now, and thus I can say that it's just perfect for beginners like me!!! -- Jay

> The book is very good and fun to read too. Thank you for sharing it. -- Yosi Izaq

> Your books should sell like hot cakes for their way of presentation. -- Deepak

> Awesome! Thank you for all of your hard work. It is especially nice that the beginning starts the reader off gradually. After using vim for a couple of years I've forgotten how weird it seems at first, so I probably wouldn't explain it that good to someone. I will be sure to share your book to spread the vim propaganda. ;-) -- Joseph Sullivan

> What I am trying to say is if you have basic computer competency you should immediately get Vim on your machine and improve your life. The best place to learn about is to read Swaroop C H's awesome eBook A Byte of Vim, it will revolutionize how you think about text editors in the future. -- "wooden nickels"

> Have been thumbing through 'byte of vim'. learning a ton even having used vim for years. -- Josh Nichols

> Great Book !! Although I use vim everyday as an editor as well as an ide, the book makes u realise how much more it can do. -- Raseel Bhagat

> Wonderful! This was one most-required-book. I was a vim user for the past years, but never have seen these much of facilities inside that! Thanks for the Book, Swaroop! -- Hiran Venugopalan

> What a nice book. I'm a long-time vim user, but never managed to get my head around vim scripting (apart from fixing some bugs in others' scripts). This is the best introduction to Vim scripting (writing plugins, syntax files, ...) I have seen so far. Thanks for putting it online! -- Anonymous (132.230.122.35)

> Thank you Swaroop! I've begun reading it and must say it's very well written. And I have no doubts this great community of us vim users here will improve it through fixes, additions or small corrections. -- Eduard Fabra

> I'd recommend A Byte of Vim as a good intro for new and experienced users alike. -- James Kovacs, Germany

> Este libro me ayudó muchísimo a sacarle provecho a vim (This book helped me a lot to take advantage of vim) -- Antonio Touriño, Panamá

> I have used vi for quite a while and am pretty comfortable with it. But all I do in vi is first get into insert mode and edit the file as if I am using notepad. I started searching for tutorials on vi and found one that is so simple yet so amazing. I thank Swaroop CH, yes the same guy who wrote the book A Byte of Python, for another amazing book A Byte of Vim. It was so amazing and I started using vi like a novice vimmer. -- Nishanth

> A Byte of Vim saved my day once again... -- @reku

Also:

- The book was listed as the top tip for Dec 2008 on the Official Vim Tips wiki.

License

This book is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

This means:

- You are free to Share i.e. to copy, distribute and transmit this book
- You are free to Remix i.e. to make changes to this book (especially translations)
- You are free to use it for commercial purposes

Please note:

- Please do *not* sell electronic or printed copies of the book unless you have clearly and

prominently mentioned in the description that these copies are *not* from the original author of this book.

- Attribution *must* be shown in the introductory description and front page of the document by linking back to http://vim.swaroopch.com/ and clearly indicating that the original text can be fetched from this location.
- All the code/scripts provided in this book is licensed under the 3-clause BSD License unless otherwise noted.

Read Now

You can read it online at http://vim.swaroopch.com/

Buy The Book

A printed hardcopy of the book can be purchased at http://www.swaroopch.com/buybook/ for your offline reading pleasure, and to support the continued development and improvement of this book.

Download

Visit https://www.gitbook.com/book/swaroopch/byte-of-vim/details for the following types of downloads:

- PDF (for desktop reading, etc.)
- EPUB (for iPhone/iPad, ebook readers, etc.)
- Mobi (for Kindle)

Visit https://github.com/swaroopch/byte-of-vim for the raw content (for suggesting corrections, changes, translating, etc.)

Read the book in your native language

If you are interested in reading or contributing translations of this book to other human languages, please see "Translations" chapter.

Preface

About Vim

Vim is a computer program used for writing, and it provides a range of features that help you **write better**.

Why Vim?

Let's face it, it's very rare to produce your best work on the first attempt. Most likely, you will keep editing it frequently until it becomes 'good'.

As Louis Brandeis once said:

> There is no great writing, only great rewriting.

Making these numerous rapid changes would be a lot easier if we had a capable editor to help us, and that is *exactly* where Vim shines, and is far better compared to most plain text editors and rich document editors.

Why Write This Book?

I have been using the Vim editor ever since I learned to use the old vi editor during Unix classes in college. Vim is one of the few pieces of software that I use for nearly 10 hours a day. I knew there were just so many features that I didn't know about but could potentially be useful to me, so I started exploring Vim little by little.

To crystallize my understanding and to help others also explore Vim, I started writing this collection of notes, and called it a book.

Some of the principles I have tried to keep in mind while writing these notes are:

1. Simple literature. The importance of this should be reinforced again and again.
2. Emphasis on examples and how-to.
3. The one-stop shop for readers to learn Vim - from getting started to learning advanced stuff.
4. Get the user to understand how to do things the Vim way - from modes to buffers to customization. Most people learn only the basic vi commands and do not attempt to learn anything beyond that. Learning such concepts is the tipping point, they become

hardcore Vim users i.e. Vimmers, which means they extract the most out of Vim, which is the intent of this book.

5. A lot of things are documented and stored here as a reference for people such as how to use Vim as an IDE, etc. There are various ways of doing it and instead of the user struggling to figure out which plugins to try out, the book already has the basic background work already for the reader.

6. Just enough info to get you to understand and use, not everything required (Pareto principle)

7. Relatedly, the book shouldn't attempt to rewrite the reference manual. Where appropriate, it should simply point out the relevant parts. This way, there is no redundancy, the user learns to use the awesome built-in reference manual which is important, and the book can stand on its own strengths as well.

To summarize, the mantra is *Concepts. Examples. Pithy.*

Status of the Book

The book was a work-in-progress and last updated in 2008. Eight years later (2016), I recreated the book in GitBook format. So let's just say it was a "1.0" book in 2008 :-)

Constructive suggestions are most welcome. Please send your thoughts and suggestions via email or github issues.

Official Website

The official website of the book is http://vim.swaroopch.com/ . From the website, you can read the whole book online or download the latest versions of the book, and also send me feedback.

Something To Think About

> Books aren't written - they're rewritten. Including your own. It is one of the hardest things to accept, especially after the seventh rewrite hasn't quite done it. -- Michael Crichton

> Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away. -- Antoine de Saint-Exupery

Introduction

What is Vim?

Vim is a computer program used for writing any kind of text, whether it is your shopping list, a book, or software code.

What makes Vim special is that it is one of those few software which is both **simple and powerful**.

Simple means it is easy to get started with. Simple means that it has a minimalistic interface that helps you to concentrate on your main task - writing. Simple means it is built around few core concepts that helps you learn deeper functionality easily.

Powerful means getting things done faster, better and easier. Powerful means making not-so-simple things possible. Powerful does not mean it has to be complicated. Powerful means following the paradigm of **"Minimal effort. Maximal effect."**

What can Vim do?

I can hear you say, "So it's a text editor. What's the big deal anyway?"

Well, a lot.

Let's see some random examples to compare Vim with your current choice of editor. The point of this exercise is for you to answer the question *"How would I do this in the editor I currently use?"* for each example.

> NOTE: Don't worry too much about the details of the Vim commands here, the point here is to enlighten you with the possibilities, not to start explaining how these things work. That is what the rest of the book is for.

Edit	In Vim	In your editor
How do you move the cursor down by 7 lines?	Press `7j`	(Fill this column)
How do you delete a word? Yes, a "word"	Press `dw`	
How do you search the current file for the current	Press `*`	

word that the cursor is at?		
How to find and replace only in lines 50-100?	Run `:50,10s/old/new/g`	
How to view two different parts of the same file simultaneously?	Run `:sp` to 'split' the view	
The cursor is at a file name, how to jump to that file?	Press `gf` (which means 'g'o to 'f'ile)	
Switch to a better theme?	Run `:colorscheme desert` to choose the `desert` theme	
How to map `ctrl-s` to save the file?	Run `:nmap <c-s> :w<CR>` (`<CR>` means 'c'arriage 'r'eturn, i.e. the enter key)	
How to save the current set of open files & settings so that you can restart the session later?	Run `:mksession ~/session.vim` and then open Vim next time with `vim -S ~/session.vim`	
How to see colors for different parts of your code?	Run `:syntax on` . If it doesn't recognize the language properly, use `set ft=python` for example.	
How to hide different parts of the file so that you can concentrate on only one part at a time?	Run `:set foldmethod=indent` assuming your file is properly indented.	
How to open multiple files in tabs?	Use `:tabedit <file>` to open multiple files in "tabs" (just like browser tabs), and use `gt` to switch between tabs	
You use some words frequently in your document and wish there was a way that it could be quickly filled in the next time you use the same word?	Press `ctrl-n` to see the list of "completions" for the current word, based on all the words that you have used in the current document. Alternatively, use `:ab mas Maslow's hierarchy of needs` to expadn the abbreviation automatically when you type `m a s <space>` .	
You have some data where only the first 10 characters in each line are useful and the rest is no longer useful for you. How do you get only that data?	Press `ctrl-v` , select the text and press `y` to copy the selected rows and columns of text.	
What if you received a document from someone	(1) Run the following: `:for i in range(0, line('$'))	call setline(i, tolower(getline(i)))	endfor` . (2) Don't worry, details will be explored in later	

which is all in capitals, find it irritating and want to convert it to lower case?	chapters. A more succinct way would be to run `:%s#\\(.\\)#\\l\\1#g` , but the first way would be simpler. (3) Select all the text using `1GVG` and then using the `u` operator to convert the selection to lowercase.

Phew. Are you convinced yet?

In these examples, you can see the power of Vim in action. Any other editor would make it insanely hard to achieve the same level of functionality. And yet, amazingly, all this power is made as understandable as possible.

Notice that we didn't use the mouse even once during these examples! This is a good thing. Count how many times you shift your hand between the keyboard and the mouse in a single day, and you'll realize why it is good to avoid it when possible.

Don't be overwhelmed by the features here. The best part of Vim is that you don't need to know all of these features to be productive with it, you just need to know a few basic concepts. After learning those basic concepts, all the other features can be easily learned when you need them.

Installation

Let's see how to get Vim installed on your computer.

Windows

If you use Microsoft Windows, then the following steps will help you get the latest version of Vim 7 installed on your computer:

1. Visit http://www.vim.org/download.php#pc
2. Download the "Self-installing executable" (gvim72.exe [1] as of this writing)
3. Double-click the file and install Vim like any other Windows-based software.

Mac OS X

If you use Mac OS X, then you already have the terminal version of Vim installed. Run the menu command Finder → Applications → Utilities → Terminal. In the terminal, run the command vim and press enter, you should now see the Vim welcome screen.

If you want to use a graphical version of Vim, download the latest version of the Cocoa-based MacVim project. Double-click the file (such as `MacVim-7_2-stable-1_2.tbz`), it will be unarchived and a directory called `MacVim-7_2-stable-1_2` will be created. Open the directory, and copy the MacVim app to your Applications directory.

For more details MacVim diffrences, including how to run MacVim from the terminal see the macvim reference:

1. Click on Finder → Applications → MacVim.
2. Type :help macvim and press the Enter key.

Linux / BSD

If you are using a Linux or *BSD system, then you will have at least a minimal console version of Vim already installed. Open a terminal program such as `konsole` or `gnome-terminal` , run `vim` and you should see the Vim welcome screen.

If you get a message like `vim: command not found`, then Vim is not installed. You will have to use your system-specific tools to install Vim, such as aptitude in Ubuntu/Debian Linux, yum in Fedora Linux, pkg_add or port in FreeBSD, etc. Please consult your specific system's documentation and forums on how to install new packages.

If you want the graphical version, install the vim-gnome package or alternatively, the gvim package.

Summary

Depending on how it is installed, you can run the `vim` command in the shell or use your operating system's menus to open a graphical version of the Vim application.

Now that we have Vim installed on your computer, let us proceed to use it in the next chapter.

First Steps

Starting Vim

First step is, of course, to learn how to start Vim.

Graphical Version

Windows

Click on Start → Programs → Vim 7 → gVim.

Mac OS X

Click on Finder → Applications → MacVim.

Linux / BSD

Click on Applications → Accessories → GVim Text Editor, or press `Alt+F2`, type `gvim` and press the `enter` key.

Terminal version

Windows

Click on Start → Run, type `vim` and press the `enter` key.

Mac OS X

Click on Finder → Applications → Utilities → Terminal, type `vim` and press the `enter` key.

Linux / BSD

Click on Applications → Accessories → Terminal, or press `Alt+F2`, type `konsole` / `gnome-terminal` and press the `enter` key. Then, type `vim` and press the `enter` key.

Summary

From now onwards when we say 'open Vim', use either of the two methods mentioned above.

> NOTE: When you started Vim, you might have noticed that you can't immediately start typing text. Don't panic, all will be explained in a little while.

Graphical or Terminal?

The graphical version of Vim has menus at the top of the application as well as various options accessible via the mouse, but note that this is completely optional. You can still access all the features of Vim using *only* the keyboard.

Why is this important? Because once a person becomes efficient at typing, using only the keyboard makes the person much faster and less error-prone, as opposed to using the mouse. This is because the hand movement required to switch between the keyboard and the mouse is slow and there is a context switch required in the mind of the person when shifting the hand between the keyboard and the mouse. If we make it a habit to use the keyboard as much as possible, you're saving valuable hand movement.

Of course, this is subjective. Some people prefer the mouse and some prefer the keyboard. I encourage you to use the keyboard as much as possible to experience the real power of Vim.

Introduction to Modes

Imagine it's a Saturday evening and you're bored of the shows on television. You want to watch an old favorite movie instead. So, you *switch the TV to video mode* so that it shows what the DVD player is displaying instead of the cable channels. Note that the television is still displaying video, but you switch the context on whether you want to watch a DVD or a live television channel.

Similarly, Vim has modes. For example, Vim has a mode for writing text, a mode for running commands, etc. They are all related to the main purpose of editing text, but you switch context on whether you want to simply type stuff or you want to run some commands on the text.

Isn't that simple?

Traditionally, the concept of modes is the most oft-cited reason by beginners on why they find Vim "confusing". I compare it to riding a bicycle - you'll trip a few times, but once you've got the hang of it, you'll wonder what the fuss was all about.

So why does Vim have modes? To make things as simple as possible, even though it's usage may seem "strange" at first.

What do I mean by that? Let's take an example - one of the key goals in Vim is to make it fully accessible from the keyboard without ever having to need to use a mouse (you can still use the mouse if you want to but it is strictly optional). In such a scenario, how would you distinguish between the text you want to write, and the commands that you want to run? Vim's solution is to have a "normal" mode where you can execute commands and an "insert" mode where you are simply writing text. You can switch between the two modes any time. For example, pressing `i` switches Vim to insert mode, and pressing `<Esc>` switches Vim back to normal mode.

How do traditional editors achieve this distinction between commands and writing text? By using graphical menus and keyboard shortcuts. The problem is that this does not scale. First of all, if you have hundreds of commands, creating menus for each of these commands would be insane and confusing. Secondly, customizing how to use each of these commands would be even more difficult.

Let's take a specific example. Suppose you want to change all occurrences of the word "from" to the word "to" in a document. In a traditional editor, you can run a menu command like Edit → Replace (or use a keyboard shortcut like `ctrl-r`) and then enter the 'from' word and the 'to' word and then click on 'Replace'. Then, check the 'Replace All' option. In Vim, you simply run `:%s/from/to/g` in the normal mode. The `:s` is the "substitute" command. See how simpler this is?

What if you want to now run this substitution only in the first 10 lines of the text and you want to have a yes/no confirmation for each replacement? In traditional text editors, achieving the yes/no confirmation is easy by unchecking the 'Replace All' option, but notice that you have to first search for this option and then use the mouse to click on the option (or use a long series of keys using the keyboard). But how will you run the Replace for only the first 10 lines? In Vim, you can simply run `:0,10s/from/to/gc`. The new `c` option we are using means we want a 'c'onfirmation message for every replace.

By separating the writing (insert) and command (normal) modes, Vim makes it easy for us to switch the two contexts easily.

Notice how the first steps to using Vim seem a little "weird", a little "strange", but once you have seen it in action, it starts to make sense. The best part is that these core concepts will help you to understand all you need to know on how to use Vim.

Since you now understand the difference between normal mode and insert mode, you can look up the various commands you can run in the normal mode, and you can immediately start using them. Compare that to learning new commands in traditional editors which

generally means having to read a lot of documentation, searching a lot of menus, a lot of trial and error or plain asking someone for help.

Personally, I find the names of the modes not intuitive to beginners. I prefer calling the insert mode as "writing" mode and the normal mode as "rewriting" mode, but we will stick to the standard Vim terminology to avoid confusion.
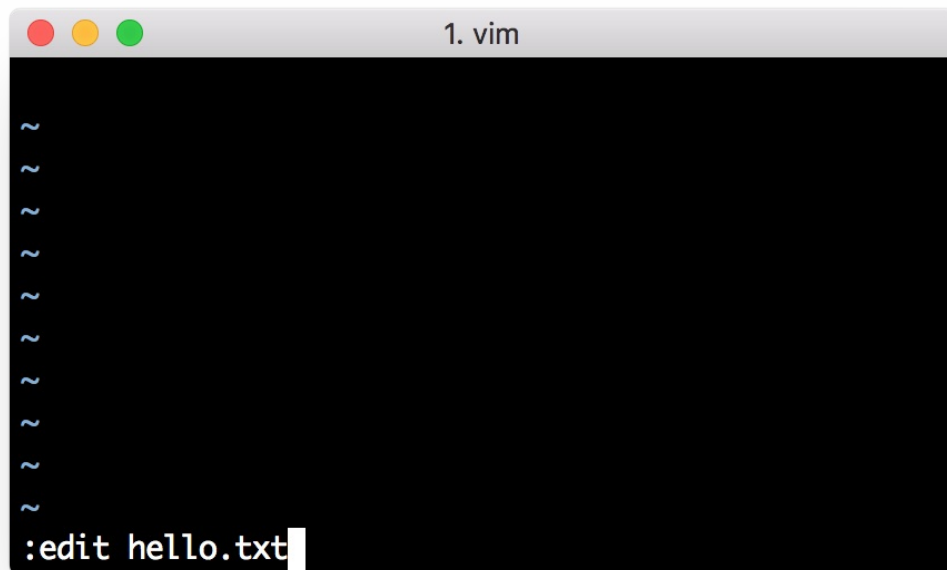
> NOTE: All commands in the normal mode should end with the enter key to signal Vim that we have written the full command. So, when we say run `:help vim-modes-intro`, it means you should type `:help vim-modes-intro` and then press the enter key at the end of the command.
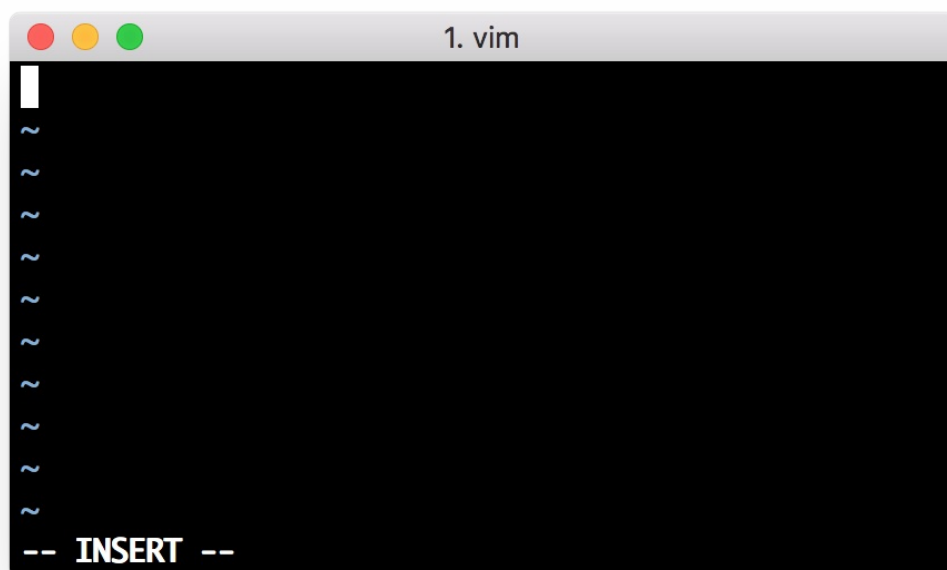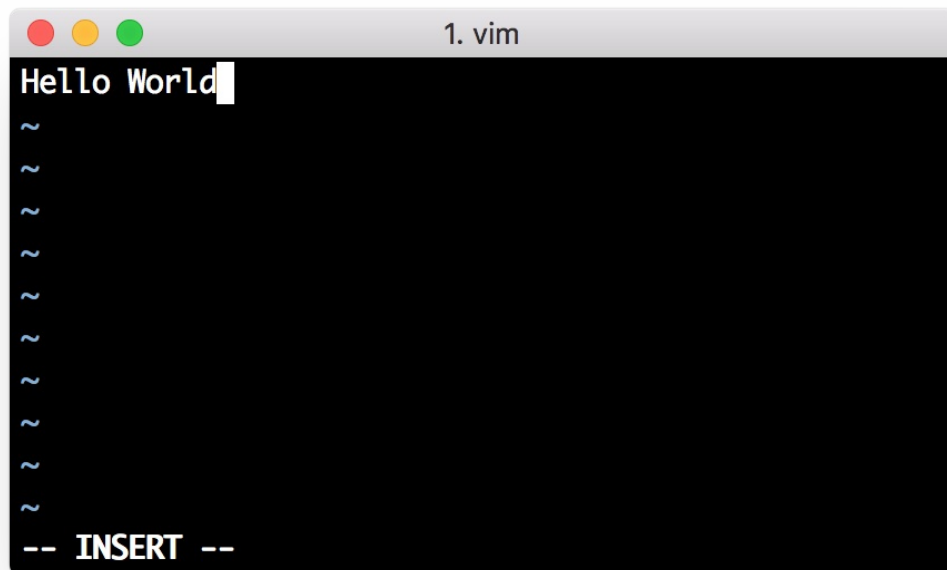
Writing a file

1. Open Vim.
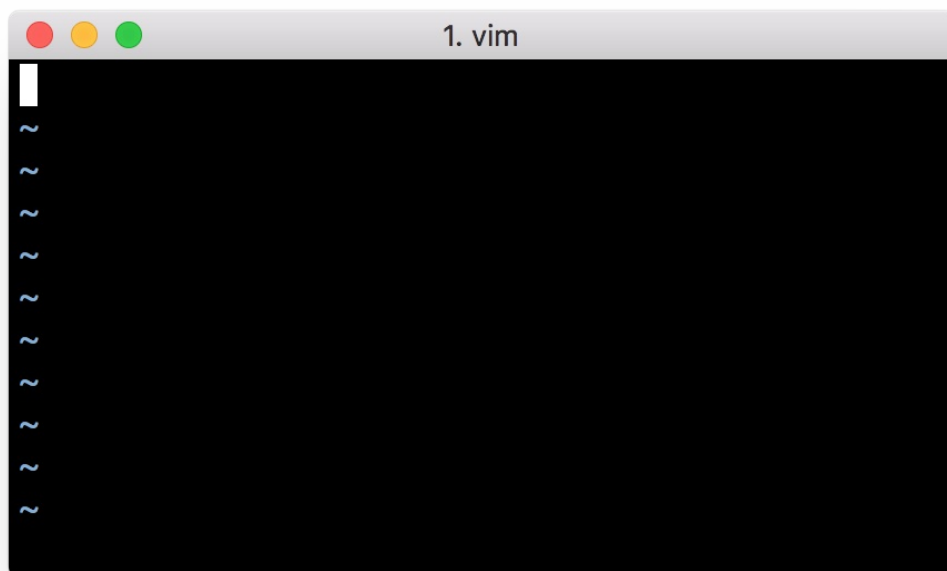
2. Type `:edit hello.txt` and press the enter key.

```
                                 1. vim

~
~
~
~
~
~
~
~
~
:edit hello.txt
```

3. Press `i` to switch to insert mode.

```
                                 1. vim

~
~
~
~
~
~
~
~
~
-- INSERT --
```
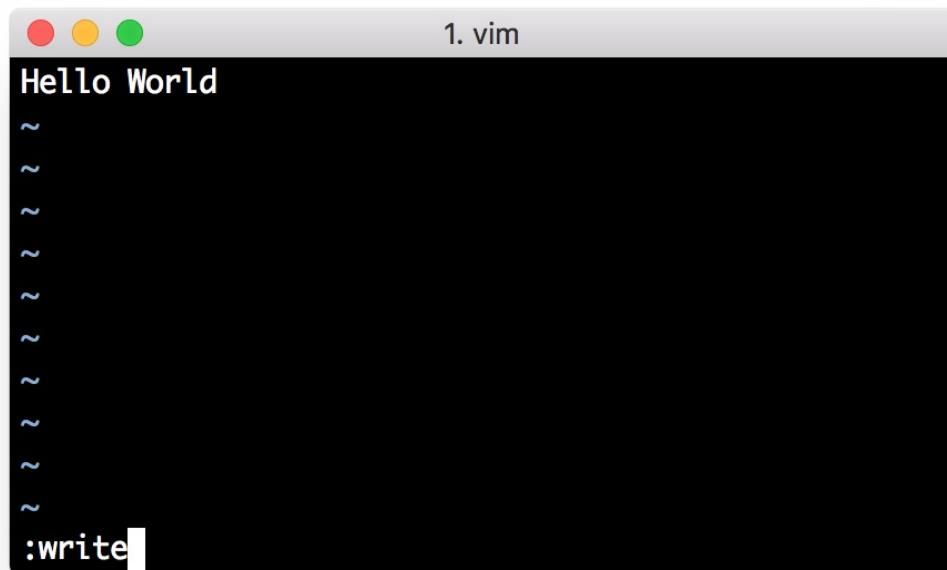
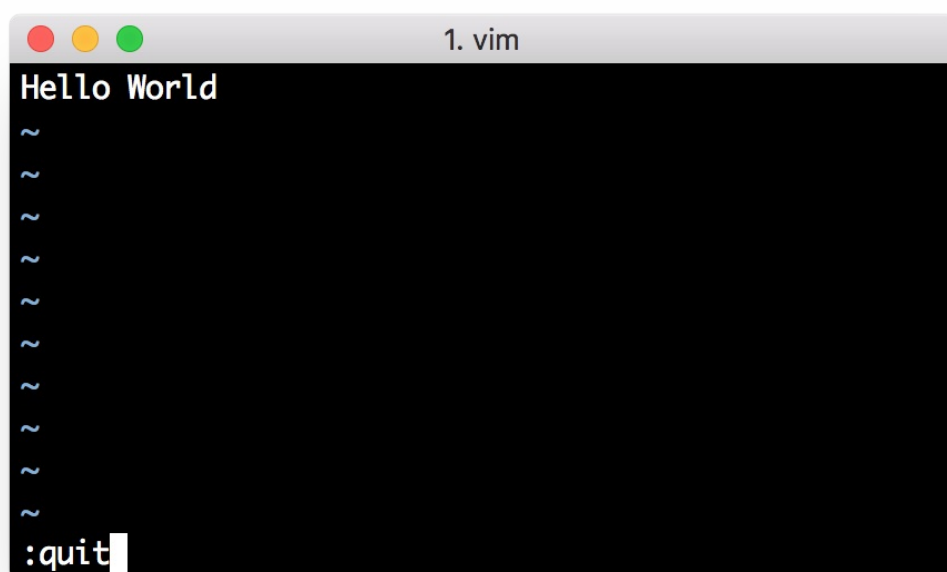4. Type the text `Hello World` .

5. Press the `<Esc>` key.

6. Type `:write` and press the `enter` key.

7. Close Vim by running `:quit` .

Congratulations! You just wrote your first file using Vim :-).

Does this seem like a lot of steps? Yes, it does, *at first*. That is because this is the first time we are getting used to opening Vim, writing a file and closing Vim. You have to keep in mind that this will only be a minor part of your time compared to the actual time that goes into in

editing the content of the document.

Let us see what the above commands do.

- `:edit hello.txt` or simply `:e hello.txt`
  - This opens a file for editing. If the file with the specified name does not exist, it will <mark>be created the first time we "save" the file</mark>.
- Press `i`
  - This switches Vim to the insert mode
- Type the text `Hello World`
  - This is where you type the actual text that you want to write.
- Press `<Esc>`
  - This escapes Vim back to normal mode
- `:write` or simply `:w`
  - This tells Vim to write the text (which is currently stored in the computer's memory) to the file on the hard disk. This means that whatever we wrote so far is now permanently stored.
- `:quit` or simply `:q`
  - This tells Vim to quit the file in the "window" that we are editing. If there was only one "window" open, this will also close Vim (Concept of windows will be discussed in a later chapter).

Try to repeat this process a few times with different file names, different text, etc. so that you get used to the basic set of steps in using Vim.

Notice that when you are in insert mode, Vim displays `-- INSERT --` at the bottom left corner. When you switch to normal mode, it will not display anything. This is because normal mode is the *default* mode in which Vim runs.

Take some time to soak in this information, this is probably the hardest lesson there is to learn about Vim, the rest is easy :)

And don't worry, help is not too far away. Actually, it's just a `:help` command away. For example, run `:help :edit` and you'll see the documentation open up. Go ahead, try it.

Summary

We have now discussed the basic concepts and usage of Vim. See `:help notation` and `:help keycodes` also.

Be sure to understand these concepts well. Once you start "thinking in Vim", understanding the rest of Vim's features is easy.

Modes

We had our first encounter with modes in the previous chapter. Now, let us explore this concept further regarding types of modes available and what we can do in each mode.

Types of modes

There are three basic modes in Vim:

1. Normal mode is where you can run commands. This is the default mode in which Vim starts up.
2. Insert mode is where you insert i.e. write the text.
3. Visual mode is where you visually select a bunch of text so that you can run a command/operation only on that part of the text.

Normal mode

By default, you're in normal mode. Let's see what we can do in this mode.

Type `:echo "hello world"` and press enter. You should see the famous words hello world echoed back to you. What you just did was run a Vim command called `:echo` and you supplied some text to it which was promptly printed back.

Type `/hello` and press the enter key. Vim will search for that phrase and will jump to the first occurrence.

This was just two simple examples of the kind of commands available in the normal mode. We will see many more such commands in later chapters.

How to use the help

Almost as important as knowing the normal mode, is knowing how to use the `:help` command. This is where you learn more about the commands available in Vim.

Remember that you do not need to know every command available in Vim, it's better to simply know where to find them when you need them. For example, see `:help usr_toc` takes us to the table of contents of the reference manual. You can see `:help index` to

search for the particular topic you are interested in, for example, run `/insert mode` to see the relevant information regarding insert mode.

If you can't remember these two help topics at first, just press `F1` or run `:help`.

Insert mode

When Vim starts up in normal mode, we have seen how to use i to get into insert mode. There are other ways of switching from normal mode to insert mode as well:

1. Run `:e dapping.txt`
2. Press `i`
3. Type the following paragraph (including all the typos and mistakes, we'll correct them later):

   > means being determined about being determined and being passionate about being passionate

4. Press `<Esc>` key to switch back to normal mode.

5. Run `:w`

Oops, we seem to have missed a word at the beginning of the line, and our cursor is at the end of the line, what do we do now?

What would be the most efficient way of going to the start of the line and insert the missing word? Should we use the mouse to move the cursor to the start of the line? Should we use arrow keys to travel all the way to the start of the line. Should we press home key and then press i to switch back to insert mode again?

It turns out that the most efficient way to be press `I` (upper case I):

1. Press `I`
2. Type `Dappin`
3. Press `<Esc>` key to switch back to the normal mode.

Notice that we used a different key to switch to insert mode, its specialty is that it moves the cursor to the start of the line and then switches to the insert mode.

Also notice how important it is to *switch back to the normal mode as soon as you're done typing the text*. Making this a habit will be beneficial because most of your work (after the initial writing phase) will be in the normal mode - that's where the all-important rewriting/editing/polishing happens.

Now, let's take a different variation of the `i` command. Notice that pressing `i` will place your cursor before the current position and then switch to insert mode. To place the cursor 'a'fter the current position, press `a`.

1. Press `a`
2. Type `g` (to complete the word as "Dapping")
3. Press `<Esc>` to switch back to normal mode

Similar to the relationship between `i` and `I` keys, there is a relationship between the `a` and `A` keys - if you want to append text at the end of the line, press the `A` key.

1. Press `A`
2. Type `.` (put a dot to complete the sentence properly)
3. Press `<Esc>` to switch back to the normal mode

To summarize the four keys we have learnt so far:

Command	Action
`i`	insert text just before the cursor
`I`	insert text at the start of the line
`a`	append text just after the cursor
`A`	append text at the end of the line

Notice how the upper case commands are 'bigger' versions of the lower case commands.

Now that we are proficient in quickly moving in the current line, let's see how to move to new lines. If you want to 'o'pen a new line to start writing, press the `o` key.

1. Press `o`
2. Type `I'm a rapper`.
3. Press `<Esc>` to switch back to the normal mode.

Hmmm, it would be more appealing if that new sentence we wrote was in a paragraph by itself.

1. Press `o` (upper case 'O')
2. Press `<Esc>` to switch back to the normal mode.

To summarize the two new keys we just learnt:

Command	Action
`o`	open a new line below
`O`	open a new line above

Notice how the upper and lower case 'o' commands are opposite in the direction in which they open the line.

Was there something wrong in the text that we just wrote? Aah, it should be 'dapper', not 'rapper'! It's a single character that we have to change, what's the most efficient way to make this change?

We could press `i` to switch to insert mode, press `<Del>` key to delete the `r`, type `d` and then press `<Esc>` to switch back to the insert mode. But that is four steps for such a simple change! Is there something better? You can use the `s` key - s for 's'ubstitute.

1. Move the cursor to the character `r` (or simply press `b` to move 'b'ack to the start of the word)
2. Press `s`
3. Type `d`
4. Press `<Esc>` to switch back to the normal mode

Well, okay, it may not have saved us much right now, but imagine repeating such a process over and over again throughout the day! Making such a mundane operation as fast as possible is beneficial because it helps us focus our energies to more creative and interesting aspects. As Linus Torvalds says, *"it's not just doing things faster, but because it is so fast, the way you work dramatically changes."*

Again, there is a bigger version of the `s` key, `s` which substitutes the whole line instead of the current character.

1. Press `s`
2. Type `Be a sinner`.
3. Press `<Esc>` to switch back to normal mode.

Command	Action
s	substitute the current character
S	substitute the current line

Let's go back our last action... Can't we make it more efficient since we want to 'r'eplace just a single character? Yes, we can use the `r` key.

1. Move the cursor to the first character of the word `sinner`.
2. Press `r`
3. Type `d`

Notice <mark>we're already back in the normal mode and didn't need to press `<Esc>`.</mark>

There's a bigger version of `r` called `R` which will replace continuous characters.

1. Move the cursor to the 'i' in sinner.
2. Press `R`
3. Type `app` (the word now becomes 'dapper')
4. Press `<Esc>` to switch back to normal mode.

Command	Action
`r`	replace the current character
`R`	replace continuous characters

The text should now look like this:

> Dapping means being determined about being determined and being passionate about being passionate.
> Be a dapper.

Phew. We have covered a lot in this chapter, but I guarantee that this is the only step that is the hardest. Once you understand this, you've pretty much understood the heart and soul of how Vim works, and all other functionality in Vim, is just icing on the cake.

To repeat, understanding how modes work and how switching between modes work is the key to becoming a Vimmer, so if you haven't digested the above examples yet, please feel free to read them again. Take all the time you need.

If you want to read more specific details about these commands, see `:help inserting` and `:help replacing`.

Visual mode

Suppose that you want to select a bunch of words and replace them completely with some new text that you want to write. What do you do?

One way would be to use the mouse to click at the start of the text that you are interested in, hold down the left mouse button, drag the mouse till the end of the relevant text and then release the left mouse button. This seems like an awful lot of distraction.

We could use the `<Del>` or `<Backspace>` keys to delete all the characters, but this seems even worse in efficiency.
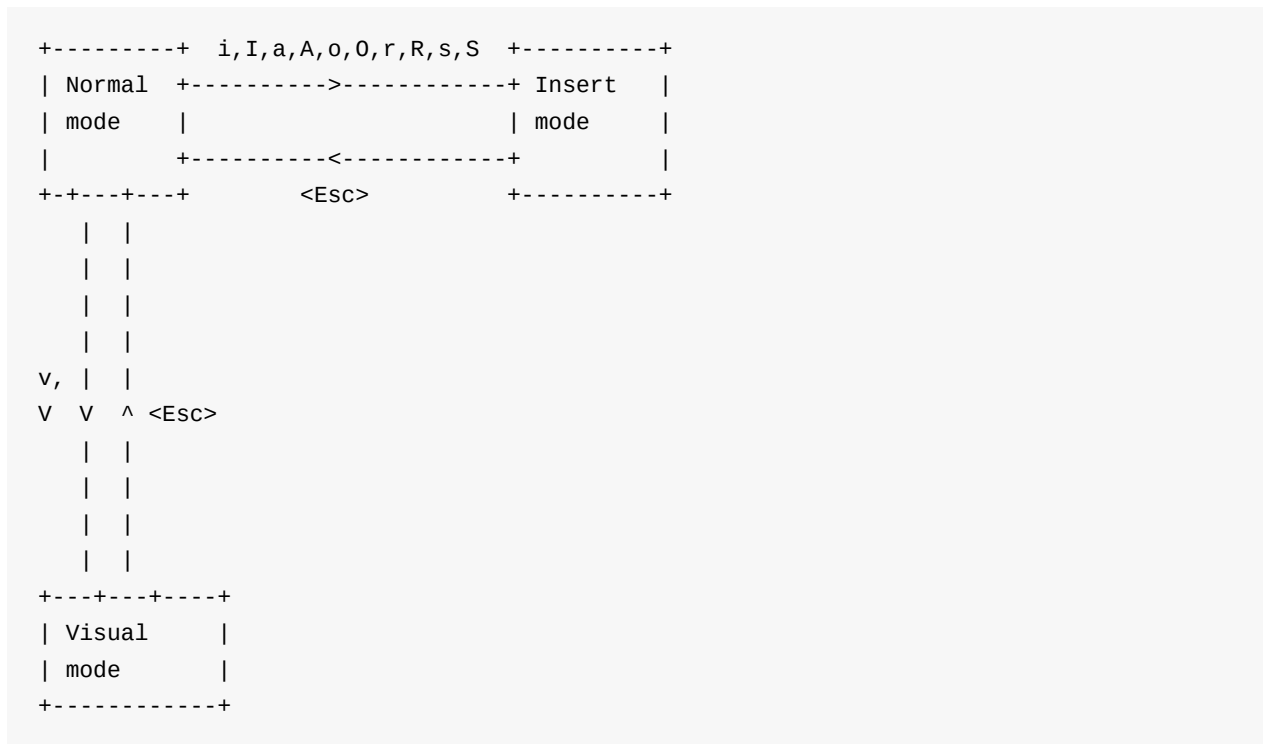
The most efficient way would be to position the cursor at the start of the text, press v to start the visual mode, use arrow keys or any text movement commands to the move to the end of the relevant text (for example, press `5e` to move to the end of the 5th word counted from the current cursor position) and then press `c` to 'c'hange the text. Notice the improvement in efficiency.

In this particular operation (the `c` command), you'll be put into insert mode after it is over, so press `<Esc>` to return to normal mode.

The `v` command works on a character basis. If you want to operate in terms of lines, use the upper case `V`.

Summary

Here is a drawing of the relationship between the different modes:

```
+---------+  i,I,a,A,o,O,r,R,s,S  +----------+
| Normal  +---------->------------+ Insert   |
| mode    |                       | mode     |
|         +---------<-----------+ |          |
+-+---+---+         <Esc>         +----------+
   |  |
   |  |
   |  |
   |  |
v, |  |
V  V  ^ <Esc>
   |  |
   |  |
   |  |
   |  |
+---+---+----+
| Visual     |
| mode       |
+-----------+
```

> NOTE: This drawing was created using Vim and Dr.Chip's DrawIt plugin.

See `:help vim-modes-intro` and `:help mode-switching` for details on the various modes and how to switch between them respectively.

If you remain unconvinced about why the concept of modes is central to Vim's power and simplicity, do read the articles on "Why Vi" and about the vi input model on why it is a better way of editing.

Typing Skills

We just learned about how to switch between the three basic modes in Vim. Isn't it hard to remember all those keys? How do you remember which key is for which operation? Well, that's the "key" question. The answer is that you shouldn't "remember", *your fingers should automatically know what to do!*. It must (literally) be on your finger tips.

So how do we do that? By making it a habit. Walking is not a nature that humans have at birth, but after trying a bit and by habit, we learn to walk. Same with Vim, although it requires less effort.

Vim was designed for people who are familiar with the keyboard. Why? Because we spend most of our time in editing tasks, which implicitly make extensive use of the keyboard, and the faster we are able to type, the faster we get the work done.

Let's start with a basic technique to get you comfortable with the keyboard.

Home Row Technique

Place your fingers on the home row of the keyboard by positioning your hands such that the fingers of your left hand are on the `ASDF` keys and the fingers of your right hand are on the `JKL;` keys as shown in the drawing (artist unknown).

Getting your hands to be comfortable in this position is one of the most important steps in learning how to use the keyboard effectively. The idea is that you should be able to type any key using the finger that is closest to that key and then your finger should automatically come back to its original position. It might seem difficult at first but try it a couple of times and you will see that you will type much faster this way.

Note that most keyboards have some home row markers on the `F` and `J` keys which serves as a reminder for you on where your fingers should be placed.

Now, try typing the alphabets A-Z using the home row technique.

Relatedly, there is also a free online typing tutorial available that explains the basics of typing skills. I would encourage you to try it for just ten minutes and explore.

Vim graphical keyboard cheat sheet

If you want to know how each key can map to something useful in Vim, see this Vim graphical cheat sheet by 'jng'.

Although a *lot* of commands are listed, for now you only need to learn the basic 'hjkl' keys which translates to left, down, up, right keys respectively. You'll learn more about this in the next chapter.

Summary

Notice that our efficiency in using Vim is directly proportional to the efficiency of using the keyboard.

Moving Around

Once you've written the initial text, editing and rewriting requires a lot of movement between the various parts of the document. For example, you're writing a story and you suddenly get an idea for a new plot, but to develop this plot you need to go back to the part where the protagonist enters the new city (or something like that)... how do you quickly move around the text so that you don't lose your train of thought?

Let's see a few examples of how Vim makes this fast.

- Want to move the cursor to the next word? Press `w` .
- Want to move to to the next paragraph? Press `}` .
- Want to move to the 3rd occurrence of the letter 'h'? Press `3fh` .
- Want to move 35 lines downwards? Press `35j` .
- After one of the above movements, want to jump back to the previous location? Press `ctrl-o` .

Want to learn how all these work? Let's dive in.

First, open a file called `chandrayaan.txt` and type the following [text from Wikipedia](#):

> Chandrayaan-1 is India's first mission to the moon. Launched by India's national space agency the Indian Space Research Organisation (ISRO). The unmanned lunar exploration mission includes a lunar orbiter and an impactor. The spacecraft was launched by a modified version of the PSLV XL on 22 October 2008 from Satish Dhawan Space Centre, Sriharikota, Andhra Pradesh at 06:23 IST (00:52 UTC). The vehicle was successfully inserted into lunar orbit on 8 November 2008. The Moon Impact Probe was successfully impacted at the lunar south pole at 20:31 hours on 14 November 2008.
>
> The remote sensing satellite had a mass of 1,380 kilograms (3,042 lb) at launch and 675 kilograms (1,488 lb) at lunar orbit and carries high resolution remote sensing equipment for visible, near infrared, and soft and hard X-ray frequencies. Over a two-year period, it is intended to survey the lunar surface to produce a complete map of its chemical characteristics and 3-dimensional topography. The polar regions are of special interest, as they might contain ice. The lunar mission carries five ISRO payloads and six payloads from other international space agencies including NASA, ESA, and the Bulgarian Aerospace Agency, which were carried free of cost.

Move your cursor, the Vim way

The most basic keys that you should use are the 'hjkl' keys. These 4 keys correspond to the left, down, up and right arrow keys respectively. Notice these keys are situated directly under your right hand when they are placed on the home row.

But why not use the arrow keys themselves? The problem is that they are located in a separate location in the keyboard and it requires as much hand movement as it requires to use a mouse.

Remember, that the right hand fingers should always be placed on `jkl;` keys (and the thumb on the space bar). Now, let's see how to use these 4 keys:
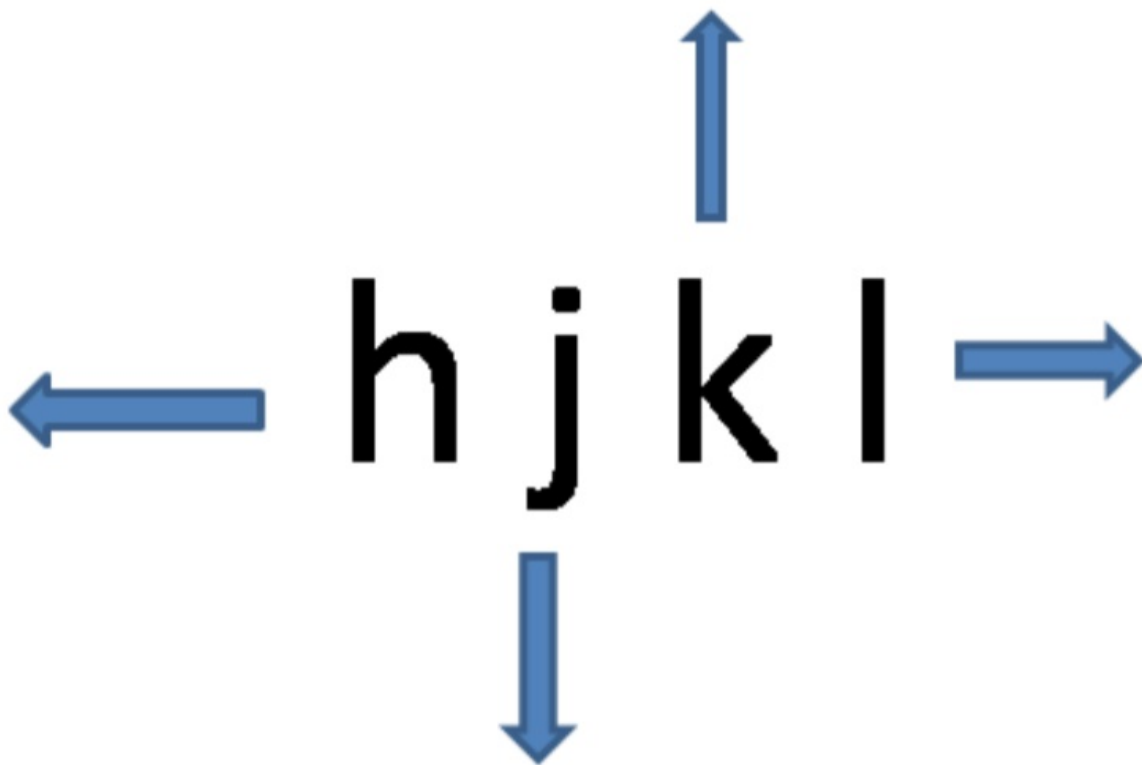
Using h,j,k,l instead of arrow keys

Key	Mnemonic
h	You have to stretch your index finger (which is on 'j') to the left to press the 'h'. This is the left-most key and signifies going left.
j	The drooping 'j' key signifies going down.
k	The upward pointing 'k' key signifies going up.
l	The right-most 'l' key signifies going right.

Note that we can repeat the operation by prefixing a count. For example, `2j` will repeat the `j` operation 2 times.

Open up the `chandrayaan.txt` text document and start practicing these keys:

1. Position your cursor at the first letter 'C' of the document.
2. Press `2j` and it should skip the current long line, the blank line and go to the second line i.e. second paragraph.
3. Press `2k` to get back to where we were. Or alternatively, press `ctrl-o` to jump back.
4. Press `5l` to move 5 characters to the right.
5. Press `5h` to move left by 5 characters. Or alternatively, press `ctrl-o` to jump back.

Make it a habit to use the 'hjkl' keys instead of the arrow keys. Within a few tries, you'll notice how much faster you can be using these keys.

Similarly, there are more simple keys that replace the following special movements. Notice that this again is intended to reduce hand movement. In this particular case, people are prone to searching and hunting for these special keys, so we can avoid that altogether.

Traditional	Vim
'home' key moves to the start of the line	`^` key (think 'anchored to the start')
'end' key moves to the end of the line	`$` key (think 'the buck stops here')
'pgup' key moves one screen up	`ctrl-b` which means move one screen 'b'ackward
'pgdn' key moves one screen down	`ctrl-f` which means move one screen 'f'orward

If you know the absolute line number that you want to jump to, say line 50, press `50G` and Vim will jump to the 50th line. If no number is specified, `G` will take you to the last line of the file. How do you get to the top of the file? Simple, press `1G`. Notice how a single key can do so much.

- Move the cursor to the first line by pressing `1G`.
- Move 20 characters to the right by pressing `20l`.
- Move back to the first character by pressing `^`.
- Jump to the last character by pressing `$`.
- Press `G` to jump to the last line.

What if you wanted to the middle of the text that is currently being shown in the window?

- Press `H` to jump as 'h'igh as possible (first line of the window)
- Press `M` to jump to the 'm'iddle of the window
- Press `L` to jump as 'l'ow as possible (last line being displayed)

You must have started to notice the emphasis on touch-typing and never having to move your hands off the main area. That's a good thing.

Words, sentences, paragraphs

We have seen how to move by characters and lines. But we tend to think of our text as words and how we put them together - sentences, paragraphs, sections, and so on. So, why not move across such text parts i.e. "text objects"?

Let's take the first few words from our sample text:

> The polar regions are of special interest, as they might contain ice.

First, let's position the cursor on the first character by pressing `^` .

> [T]he polar regions are of special interest, as they might contain ice.

> NOTE: We are using the square brackets to mark the cursor position.

Want to move to the next 'w'ord? Press `w` . The cursor should now be at the 'p' in 'polar'.

> The [p]olar regions are of special interest, as they might contain ice.

How about moving 2 words forward? Just add the prefix count to 'w': `2w` .

> The polar regions [a]re of special interest, as they might contain ice.

Similarly, to move to the 'e'nd of the next word, press `e` .

> The polar regions ar[e] of special interest, as they might contain ice.

To move one word 'b'ackward, press `b` . By prefixing a count, `2b` will go back by 2 words.

> The polar [r]egions are of special interest, as they might contain ice.

See `:help word-motions` for details.

We have seen character motions and word motions, let's move on to sentences.

> [C]handrayaan-1 is India's first mission to the moon. Launched by India's national space agency the Indian Space Research Organisation (ISRO). The unmanned lunar exploration mission includes a lunar orbiter and an impactor. The spacecraft was launched by a modified version of the PSLV XL on 22 October 2008 from Satish Dhawan Space Centre, Sriharikota, Andhra Pradesh at 06:23 IST (00:52 UTC). The vehicle was successfully inserted into lunar orbit on 8 November 2008. The Moon Impact Probe was successfully impacted at the lunar south pole at 20:31 hours on 14 November 2008.

Position the cursor at the first character using `^` .

To move to the next sentence, press `)` .

> Chandrayaan-1 is India's first mission to the moon. [L]aunched by India's national space agency the Indian Space Research Organisation (ISRO). The unmanned lunar exploration mission includes a lunar orbiter and an impactor. The spacecraft was launched by a modified version of the PSLV XL on 22 October 2008 from Satish Dhawan Space Centre, Sriharikota, Andhra Pradesh at 06:23 IST (00:52 UTC). The vehicle was successfully inserted into lunar orbit on 8 November 2008. The Moon Impact Probe was successfully impacted at the lunar south pole at 20:31 hours on 14 November 2008.

Isn't that cool?

To move to the previous sentence, press `(` .

Go ahead, try it out and see how fast you can move. Again, you can prefix a count such as `3)` to move forward by 3 sentences.

Now, use the whole text and try out moving by paragraphs. Press `}` to move to the next paragraph and `{` to move to the previous paragraph.

Notice that the 'bigger' brackets is for the bigger text object. If you had already noticed this, then congratulations, you have already started to think like a winner, err, "think like a Vimmer".

Again, don't try to *remember* these keys, try to make it a *habit* such that your fingers naturally use these keys.

See `:help cursor-motions` for more details.

Make your mark

You are writing some text but you suddenly remember that you have to update a related section in the same document, but you do not want to forget where you are currently so that you can come back to this later. What do you do?

Normally, this would mean scrolling to that section, update it, and then scroll back to where you were. This is a lot of overhead and we may tend to forget where we were last at.

We can do things a bit smarter in Vim. Move the cursor to the 5th line in the following text (the words by John Lennon). Use `ma` to create a mark named 'a'. Move the cursor to wherever you want, for example `4j` .

> I am eagerly awaiting my next disappointment. -- Ashleigh Brilliant
>
> Every man's memory is his private literature. -- Aldous Huxley
>
> Life is what happens to you while you're busy making other plans. -- John Lennon
>
> Life is really simple, but we insist on making it complicated. -- Confucius
>
> Do not dwell in the past, do not dream of the future, concentrate the mind on the present moment. -- Buddha
>
> The more decisions that you are forced to make alone, the more you are aware of your freedom to choose. -- Thornton Wilder

Press `'a` (i.e. single quote followed by the name of the mark) and voila, Vim jumps (back) to the line where that mark was located.

You can use any alphabet (a-zA-Z) to name a mark which means you can have ==up to 52 named marks for each file==.

Jump around

In the various movements that we have learned, we might often want to jump back to the previous location or to the next location after a movement. To do this, simply press `ctrl-o` to jump to the previous location and `ctrl-i` to jump forward to the next location again.

Parts of the text

There are various ways you can specify text objects in Vim so that you can pass them to a command. For example, you want to visually select a part of the text and then convert the case (from upper to lower or from lower to upper case) of the text using the `~` key.

Open the `dapping.txt` file that we created in previous chapters. Use the various keys to move to the first letter of the word 'dapper' in the second paragraph. Hint: Use `}`, `j`, `w`.

> Dapping means being determined about being determined and being passionate about being passionate.
>
> Be a dapper.

Press `v` to start the visual mode, and press `ap` to select 'a' 'p'aragraph. Press `~` to flip the case of the text. If you want to cancel the selection, simply press `<Esc>`.

> Dapping means being determined about being determined and being passionate about being passionate.
>
> bE A DAPPER.

Other text object mnemonics are `aw` which means 'a' 'w'ord, `a"` means a quoted string (like "this is a quoted string"), `ab` means 'a' 'b'lock which means anything within a pair of parentheses, and so on.

See `:help object-motions` and `:help text-objects` for more details.

Summary

We have seen the rich number of methods that Vim gives us to move around the text. It is not important to remember each of these movements, it is more important to make them a habit whenever you can, especially the ones that are most relevant to you, and when they become a habit they reduce the movement of your hands, you become faster, and ultimately spend more time on thinking about your writing rather than on the software you use to write.

See `:help various-motions` as well as `:help motion` for more interesting ways of movement.

Help

Vim has such a diverse list of commands, keyboard shortcuts, buffers, and so on. It's impossible to remember how all of them work. In fact, it is not even useful to know all of them. The best situation is that you know how to look for certain functionality in Vim whenever you need it.

For example, you want to avoid having to type a long name every time, you suddenly remember there is an abbreviations feature in Vim that'll help you do just that, but don't remember how to use it. What do you do?

Let's look at the various ways of finding help about how to use Vim.

The `:help` command

The first and most important place to try to look for help is the built-in documentation and Vim has one of the most comprehensive user manuals that I've ever seen.

In our case, just run `:help abbreviation` and you'll be taken to the help for abbreviations and you can read about how to use the `:ab` and `:iab` commands.

Sometimes, it can be as simple as that. If you don't know what you're looking for, then you can run `:help user-manual` and browse through the list of contents of the entire user manual and read the chapter that you feel is relevant to what you're trying to do.

How to read the `:help` topic

Let us take some sample text from `:help abbreviate` :

```
    :ab[breviate] [<expr>] {lhs} {rhs}
                add abbreviation for {lhs} to {rhs}.  If {lhs} already
                existed it is replaced with the new {rhs}.  {rhs} may
                contain spaces.
  See |:map-<expr>| for the optional <expr> argument.
```

Notice that there is a standard way of writing help in Vim to make it easy for us to figure out the parts that are needed for us instead of trying to understand the whole command.

The first line explains the syntax i.e. how to use this command.

The square brackets in `:ab[breviate]` indicate that the latter part of the full name is optional. The minimum you have to type is `:ab` so that Vim recognizes the command. You can also use `:abb` or `:abbr` or `:abbre` and so on till the full name `:abbreviate`. Most people tend to use the shortest form possible.

The square brackets in `[<expr>]` again indicate that the 'expression' is optional.

The curly brackets in `{lhs}` `{rhs}` indicate that these are placeholders for actual arguments to be supplied. The names are short for <mark>'left hand side' and 'right hand side'</mark> respectively.

Following the first line is an indented paragraph that briefly explains what this command does.

Notice the second paragraph which points you to further information. You can position the cursor on the text between the two pipe symbols and press <mark>`ctrl-]` to follow the "link" t</mark>o the corresponding `:help` topic. To jump back, press `ctrl-o`.

The `:helpgrep` command

If you do not know what the name of the topic is, then you can search the entire documentation for a phrase by using `:helpgrep`. Suppose you want to know how to look for the beginning of a word, then just run `:helpgrep beginning of a word`.

You can use `:cnext` and `:cprev` to move to the next and previous part of the documentation where that phrase occurs. Use `:clist` to see the whole list of all the occurrences of the phrase.

Quick help

Copy the following text into a file in Vim and then also run it:

```
:let &keywordprg=':help'
```

Now, position your cursor anywhere on the word `keywordprg` and just press `K`. You'll be taken to the help immediately for that word. This shortcut avoids having to type `:help keywordprg`.

Mailing List

If you are still not able to figure out what you want to do, then the next best thing is to approach other Vim users to help you out. Don't worry, this is actually very easy and it is amazing how other Vimmers who are willing to help you out.

Search the Vim mailing list to see if someone has already answered your question. Just go to the Vim Group search page and then enter the keywords of your question. Most of the times, many common questions will be already answered since this is such a high-traffic mailing list i.e. lots and lots of people ask questions and give answers in this group.

If you cannot find any relevant answer, then post your question in the same mailing list.

Summary

There is a wealth of information on how to do things using Vim, and many Vimmers would gladly help you out as well. The Vim community is one of the greatest strengths of the Vim editor, so make sure to use the resources and do join the growing community as well.

> The true delight is in the *finding out* rather than in the *knowing*. -- Isaac Asimov

Editing Basics

Let's learn the basic editing commands in Vim for reading/writing files, <mark>cut/copy/paste, undo/redo and searching</mark>.

Reading and writing files

Buffers

When you edit a file, Vim brings the text in the file on the hard disk to the computer's RAM. This means that a copy of the file is stored in the computer's memory and any changes you make is changed in the computer's memory and immediately displayed. Once you have finished editing the file, you can save the file which means that Vim writes back the text in the computer's memory to the file on the hard disk. The computer memory used here to store the text temporarily is referred to as a "buffer". Note that this same concept is the reason why we have to "save" files in all editors or word processors that we use.

Now open up Vim, write the words `Hello World` and save it as the file `hello.txt`. If you need to remember how to do this, please refer to the First Steps chapter.

Swap

Now you will notice that another file has been created in the same directory as this file, the file would be named something like `.hello.txt.swp`. Run `:swapname` to find out the exact name of the file.

What is this file? Vim maintains a backup of the buffer in a file which it saves regularly to the hard disk so that in case something goes wrong (like a computer crash or even Vim crashes), you have a backup of the changes you have made since you last saved the original file. This file is called a "swap file" because Vim keeps swapping the contents of the buffer in the computer memory with the contents of this file on the hard disk. See `:help swap-file` to know more details.

Save my file

Now that the file has been loaded, let's do a minor editing. Press the ~ key to change the case of the character on which the cursor is positioned. You will notice that Vim now marks the file having been changed (for example a `+` sign shows up in the title bar of the GUI

version of Vim). You can open up the actual file in another editor and check that it has not changed yet i.e. Vim has only changed the buffer and not yet saved it to the hard disk.

We can write back the buffer to the original file in the hard disk by running `:write` .

> NOTE: To make saving easier, add the following line to your vimrc file:
>
> ```
> " To save, ctrl-s.
> nmap  :w
> imap  :wa
> ```
>
> Now you can simply press `ctrl-s` to save the file.

Working in my directory

Vim starts up with your home directory as the default directory and all operations will be done within that directory.

To open files located in other directories, you can use the full or relative paths such as:

```
:e ../tmp/test.txt
:e C:\\shopping\\monday.txt
```

Or you can switch Vim to that directory:

```
:cd ../tmp
```

`:cd` is short for 'c'hange 'd'irectory.

To find out the current directory where Vim is looking for files:

```
:pwd
```

`:pwd` is short for 'p'rint 'w'orking 'd'irectory.

Cut, Copy and Paste

As Sean Connery says, in the movie Finding Forrester:

> No thinking - that comes later. You must write your first draft with your heart. You rewrite with your head. The first key to writing is... to write, not to think!

When we rewrite, we frequently rearrange the order of the paragraphs or sentences i.e. we need to be able to cut/copy/paste the text. In Vim, we use a slightly different terminology:

Desktop world	Vim world	Operation
cut	delete	`d`
copy	yank	`y`
paste	paste	`p`

In normal desktop terminology, 'cut'ting text means removing the text and putting it into the clipboard. The same operation in Vim means it deletes the text from the file buffer and stores it in a 'register' (a part of the computer's memory). Since we can choose the register where we can store the text, it is referred to as the "delete" operation.

Similarly, in normal desktop terminology, 'copy' text means that a copy of the text is placed on the clipboard. Vim does the same, it "yanks" the text and places it in a register.

"Paste" means the same in both terminologies.

We have seen how you can do cut/copy/paste operations in Vim. But how do you specify which text that these operations should work on? Well, we have already explored that in the previous Text Objects section.

Combining the operation notation and the text object notation means we have innumerable ways of manipulating the text. Let's see a few examples.

Write this text in Vim (exactly as shown):

> This is the rthe first paragraph.
> This is the second line.
>
> This is the second paragraph.

Place the cursor at the topmost leftmost position, by pressing `1G` and `|` that moves to the first line and the first column respectively.

Let's see the first case: We have typed an extra 'r' which we have to remove. Press `3w` to move forward by 3 words.

Now, we need to delete one character at the current cursor position.

Note that there are two parts to this:

Operation	Text Object / Motion
Delete	One character at current cursor position
`d`	`l`

So, we have to just press `dl` and we delete one character! Notice that we can use `l` even though it is a motion.

Now we notice that the whole word is redundant because we have "the" twice. Now think carefully on what should be fastest key combination to fix this?

Take some time to think and figure this out for yourself. Take your time. Now read on.

Operation	Text Object / Motion
Delete	Word
d	w

So, press `dw` and you delete a word. Voila! So simple and so beautiful. The beauty is that such simple concepts can be combined to provide such a rich range of possibilities.

How do we achieve the same operation for lines? Well, lines are considered special in Vim because lines are usually how we think about our text. As a shortcut, if y<mark>ou repeat the operation name twice, it will operate on the line</mark>. So, <mark>dd</mark> will delete the current line and <mark>yy</mark> will yank the current line.

Our example text in Vim should now look like this:

> This is the first paragraph.
> This is the second line.
>
> This is the second paragraph.

Go to the second line by pressing `j` . Now press `dd` and the line should be deleted. You should now see:

> This is the first paragraph.
>
> This is the second paragraph.

Let's see a bigger case: How do we yank the current paragraph?

Operation	Text Object / Motion
Yank	A Paragraph
y	ap

So, `yap` will copy the current paragraph.

Now that we have done copying the text, how do we paste it? Just `p` it.

You should now see:

> This is the first paragraph.
> This is the first paragraph.
>
>
> This is the second paragraph.

Notice that the blank line is also copied when we do yap, so p adds that extra blank line as well.

There are two types of paste that can be done exactly like the two types of inserts we have seen before:

Key	Mnemonic
`p`	paste after current cursor position
`P`	paste before current cursor position

Taking the idea further, we can combine these into more powerful ways.

How to swap two characters? Press `xp` .

- `x` → delete one character at current cursor position
- `p` → paste after current cursor position

How to swap two words? Press `dwwP` .

- `d` → delete
- `w` → one word
- `w` → move to the next word
- `P` → paste before the current cursor position

The important thing is *not* to learn these operations by rote. These combinations of operations and text objects/motions should be automatic for your fingers, without you needing to put in mental effort. This happens when you make using these a habit.

Marking your territory

You are writing, and you suddenly realize you have to change sentences in a previous section to support what you are writing in this section. The problem is that you have to remember where you are right now so that you can come back to it later. Can't Vim remember it for me? This can be achieved using marks.

You can create a mark by pressing m followed by the name of the mark which is a single character from `a-zA-Z` . For example, pressing `ma` creates the mark called 'a'.

Pressing `'a` returns the cursor to line of the mark. Pressing `` `a `` will take you to the exact line and column of the mark.

The best part is that you can jump to this position using these marks any time thereafter.

See `:help mark-motions` for more details.

Time machine using undo/redo

Suppose you are rewriting a paragraph but you end up muddling up what you were trying to rewrite and you want to go back what you had written earlier. This is where we can "undo" what we just did. If we want to change back again to what we have now, we can "redo" the changes that we have made. Note that a change means some change to the text, it does not take into account cursor movements and other things not directly related to the text.

Suppose you have the text:

> I have coined a phrase for myself - 'CUT to the G':
>
> 1. Concentrate
> 2. Understand
> 3. Think
> 4. Get Things Done
>
> > Step 4 is eventually what gets you moving, but Steps 2 and 3 are equally important. As Abraham Lincoln once said "If I had eight hours to chop down a tree, I'd spend six hours sharpening my axe." And to get to this stage, you need to do Step 1 which boils down to one thing - It's all in the mind. That's why it's so hard.

Now, start editing the first line:

1. Press `s` to 's'ubstitute the whole line.
2. Type the text `After much thought, I have coined a new phrase to help me solidify my approach:` .
3. Press `<esc>` .

Now think about the change that we just did. Is the sentence better? Hmm, was the text better before? How do we switch back and forth?

Press `u` to undo the last change and see what we had before. You will now see the text `I have coined a phrase for myself - 'CUT to the G':` . To come back to the latest change, press `ctrl-r` to now see the line `After much thought, I have coined a new phrase to help me solidify my approach:` .

It is important to note that Vim gives unlimited history of undo/redo changes, but it is usually limited by the `undolevels` setting in Vim and the amount of memory in your computer.

Now, let's see some stuff that really shows off Vim's advanced undo/redo functionality, some thing that other editors will be jealous of: Vim is not only your editor, it also acts as a time machine.

For example, `:earlier 4m` will take you back by 4 minutes i.e. the state of the text 4 minutes "earlier".

The power here is that it is superior to all undoes and redoes. For example, if you make a change, then you undo it, and then continue editing, that change is actually never retrievable using simple u again. But it is possible in Vim using the `:earlier` command.

You can also go forward in time: `:later 45s` which will take you later by 45 seconds.

Or if you want the simpler approach of going back by 5 changes: `:undo 5` .

You can view the undo tree using `:undolist` .

See :help `:undolist` for the explanation of the output from this command.

See `:help undo-redo` and `:help usr_32.txt` for more details.

A powerful search engine but not a dotcom

Vim has a powerful built-in search engine that you can use to find exactly what you are looking for. It takes a little getting used to the power it exposes, so let's get started.

Let's come back to our familiar example:

> I have coined a phrase for myself - 'CUT to the G':
>
> 1. Concentrate
> 2. Understand
> 3. Think
> 4. Get Things Done
>
> Step 4 is eventually what gets you moving, but Steps 2 and 3 are equally important. As Abraham Lincoln once said "If I had eight hours to chop down a tree, I'd spend six hours sharpening my axe." And to get to this stage, you need to do Step 1 which boils down to one thing - It's all in the mind. That's why it's so hard.

Suppose we want to search for the word "Step". In normal mode, type `/Step<cr>` (i.e. `/Step` followed by `enter` key). This will take you to the first occurrence of those set of characters. Press `n` to take you to the 'n'ext occurrence and `N` to go in the opposite direction i.e. the previous occurrence.

What if you knew only a part of the phrase or don't know the exact spelling? Wouldn't it be helpful if Vim could start searching as and when you type the search phrase? You can enable this by running:

```
set incsearch
```

You can also tell Vim to ignore the case (whether lower or upper case) of the text that you are searching for:

```
set ignorecase
```

Or you can use:

```
set smartcase
```

When you have `smartcase` on:

- If you are searching for `/step` i.e. the text you enter is in lower case, then it will search for any combination of upper and lower case text. For example, this will match all the following four - "Step", "Stephen", "stepbrother", "misstep."
- If you are searching for `/Step` i.e. the text you enter has an upper case, then it will search for **only** text that matches the exact case. For example, it will match "Step" and "Stephen", but not "stepbrother" or "misstep."

> NOTE: I recommend that you put these two lines in your vimrc file (explained later, but see `:help vimrc-intro` for a quick introduction) so that this is enabled always.

Now that we have understood the basics of searching, let's explore the real power of searching. The first thing to note that what you provide Vim can not only be a simple phrase, it can be a "expression". An expression allows you to specify the 'kinds' of text to search for, not just the exact text to look.

For example, you will notice that /step will take you to steps as well as step and even footstep if such a word is present. What if you wanted to look for the exact word step and not when it is part of any other word? Then you can search using `/\<step\>`. The `\<` and `\>` indicate the start and end positions of a "word" respectively.

Similarly, what if you wanted to search for any number? Searching for `/\d` will look for a 'digit'. But a number is just a group of digits together. So we specify "one or more" digits together as `/\d\+` . If we were looking for zero or more characters, we can use the `*` instead of the `+` .

There are a variety of such magic stuff we can use in our search patterns. See `:help pattern` for details.

Summary

We have explored some of the basic editing commands that we will use in our everyday usage of Vim. *It is very important that you go through these concepts again and make them a habit.* It is not important to learn each and every option or nuances of these commands. If you know how to use the command and know how to find out more on it based on what you need, then you're a true Vimmer.

Now, go ahead and start editing!

Multiplicity

In this chapter, let's explore how Vim helps us work between different parts of the file, different files, different 'windows' and different tabs to help us to handle more simultaneously. After all, an important part about good editing is managing files.

Multiple Sections using Folds

If you're editing a long document, wouldn't it be easier if you can "close" all sections of the document and focus on only one at a time?

This is what we call *folding* in Vim.

Let us take the example where your document is structured such that each level of the text is indented one level higher such as the following piece of text:

```
Book I
    The Shadow of the Past
        Three Rings for the Elven-kings under the sky,
        Seven for the Dwarf-lords in their halls of stone,
        Nine for Mortal Men doomed to die,
        One for the Dark Lord on his dark throne
        In the Land of Mordor where the Shadows lie.
        One Ring to rule them all, One Ring to find them,
        One Ring to bring them all and in the darkness bind them
        In the Land of Mordor where the Shadows lie.

    Three is Company

        The Road goes ever on and on
        Down from the door where it began.
        Now far ahead the Road has gone,
        And I must follow, if I can,
        Pursuing it with weary feet,
        Until it joins some larger way,
        Where many paths and errands meet.
        And whither then? I cannot say.
```

NOTE: This text was taken from WikiQuote.

After writing this text, run `:set foldmethod=indent` , position your cursor on the text you want to indent, press `zc` and see how the text folds up. Press `zo` to open the fold.

Basic commands are `zo` and `zc` where we can open and close the fold respectively. You can use `za` to 'a'lternate between opening and closing a fold respectively. You can make it even easier by using the spacebar (in normal mode) to open and close a fold:

```
:nmap <space> za
```

Folding is a huge topic on its own with more ways of folding (manual, marker, expression) and various ways of opening and closing hierarchies of folds, and so on. See `:help folding` for details.

Multiple Buffers

Suppose you wanted to edit more than one file at a time using the same Vim, what do you do?

Remember that files are loaded into buffers in Vim. Vim can also load multiple buffers at the same time. So, you can have multiple files open at the same time and you can keep switching between them.

Let's say you have two files, `part1.txt` and `part2.txt` :

part1.txt

> I have coined a phrase for myself - 'CUT to the G':
>
> 1. Concentrate
> 2. Understand
> 3. Think
> 4. Get Things Done

part2.txt

> Step 4 is eventually what gets you moving, but Steps 2 and 3 are equally important. As Abraham Lincoln once said "If I had eight hours to chop down a tree, I'd spend six hours sharpening my axe." And to get to this stage, you need to do Step 1 which boils down to one thing - It's all in the mind. That's why it's so hard.

Now, run `:e part1.txt` and then run `:e part2.txt` . Observe that you have the second file now open for editing. How do you switch back to the first file? In this particular case, you can just run `:b 1` to switch to 'b'uffer number '1'. You can also run `:e part1.txt` to open the existing buffer into view.

You can see what buffers has been loaded and correspondingly, which files are being edited by running `:buffers` or a shorter form, `:ls` which stands for 'l'i's't buffers.

Buffers will be automatically removed when you close Vim, so you don't have to do anything special other than making sure you save your files. However, if you really want to remove a buffer, for example in order to use less memory, then you can use `:bd 1` to 'd'elete the 'b'uffer numbered '1', etc.

See `:help buffer-list` on the numerous things you can do with buffers.

Multiple Windows

We have seen how to edit multiple files at the same time, but what if we wanted to view two different files simultaneously. For example, you want to have two different chapters of your book open so that you can write the second chapter consistent with the wordings/description given in the first chapter. Or you want to copy/paste some stuff from the first file to the second file.

In the last section, we used the same "view" to edit multiple buffers. Vim calls these "views" as windows. This term "window" should *not* be confused with your desktop application window which usually means the entire application itself. Just think of 'windows' as 'views' on different files.

Let's take the same `part1.txt` and `part2.txt` sample files used in the previous section.

First, load the part1.txt using `:e part1.txt`. Now, let's open a new buffer by splitting the window to create a new window - run `:new`. You should now be able to do any normal editing in the new buffer in the new window, except that you can't save the text because you haven't associated a file name with the buffer. For that, you can use `:w test.txt` to save the buffer.
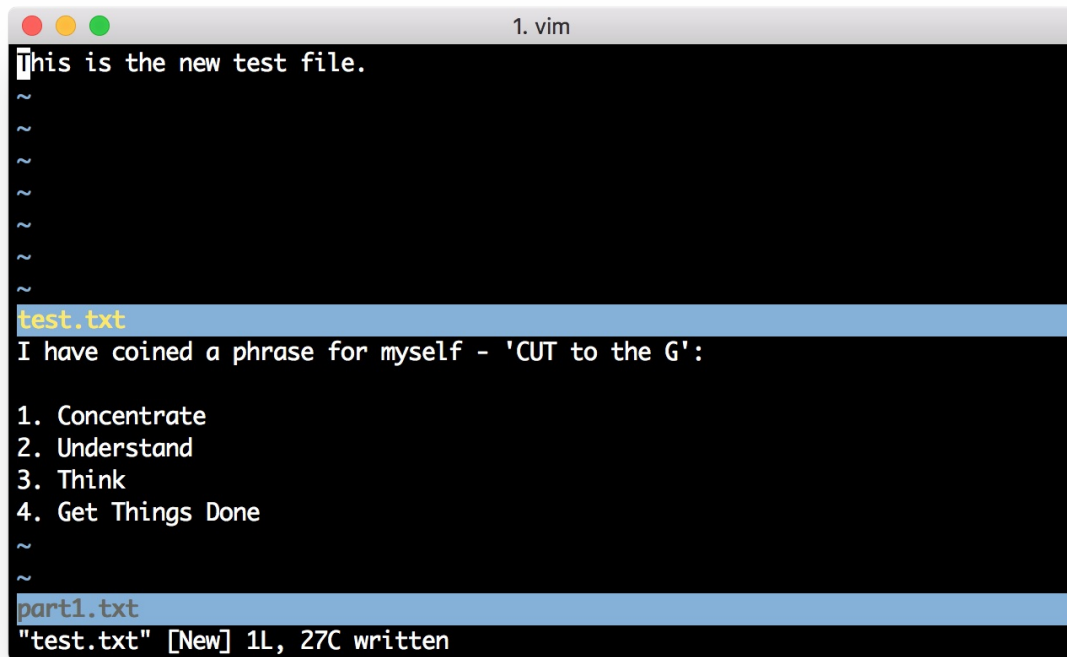
How do you switch between these two windows? Just use `ctrl-w <motion key>` to switch between the windows. Motion keys can be one of `h` , `j` , `k` , `l` or even any of the arrow keys (in this example, only up and down keys make sense). Remember that `ctrl-w` operations work on 'w'indows.

As a quick shortcut, you can press `ctrl-w` twice i.e. `ctrl-w ctrl-w` to cycle between all the open windows.

A particular situation where multiple windows are useful is when you want to view two different parts of the same file simultaneously. Just run `:sp` to create a 'sp'lit window and then you can scroll each window to a different position and continue your editing. Since they both are "windows" to the same buffer, changes in one window will be immediately reflected in the other window. You can also use `ctrl-w s` instead of `:sp` .

To create a vertical split, use `:vsp` or `ctrl-w v` . To close a window, just run `:q` .

Now that we have seen how to open and use multiple windows, let's see how to further play around with the display.

- Suppose you have two split windows but want to reverse the windows, so that you can focus your eyes on the bottom part or top part of your computer screen, as per your preference? Press `ctrl-w r` to 'r'otate the windows.
- Want to move the current window to the topmost position? Press `ctrl-w K` .
- Want to resize a window to make it smaller or larger? Run `:resize 10` to make it 10 lines long display, etc.

- Want to make the current window as big as possible so that you can concentrate on it? Press `ctrl-w _` . Think of the underscore as an indication that the other windows should be as small as possible.
- Want to make the windows 'equal' in height again? Press `ctrl-w =` .

See `:help windows` on more details on what you can do with windows.

Multiple Tabs

In web browsers (such as Firefox, Google Chrome or Safari), you may have used the tabs feature which allows you to open multiple websites in a single window so that you can switch between them without having the headache of switching between multiple windows. Well, tabs work exactly the same way in Vim also. Except that they are called "tab pages."

Run `:tabnew` to open a new tab with a new buffer (analogous to `:new`). How do you switch between the tabs? By pressing `gt` to 'g'o to the next 't'ab and `gT` to 'g'o in the opposite direction i.e. the previous 't'ab.
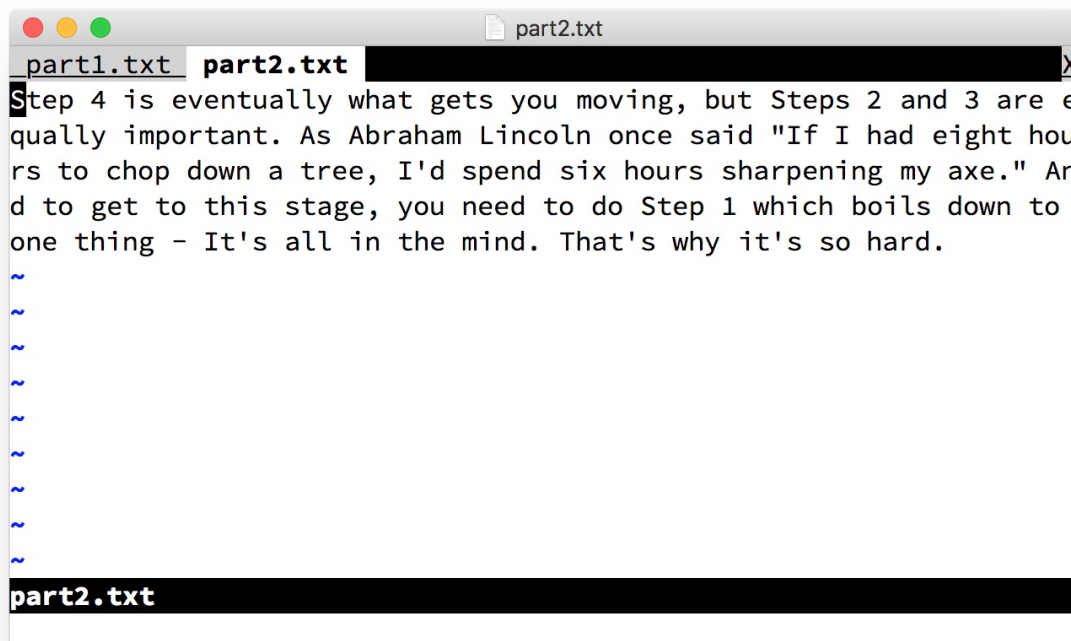
I personally prefer to use the keys `alt-j` and `alt-k` for the same operations analogous to how the `j` and `k` keys work for characters and how `ctrl-w j` and `ctrl-w k` work for (horizontally split) windows. To enable this, add the following lines to your vimrc file:

```
" Shortcuts for moving between tabs.
" Alt-j to move to the tab to the left
noremap  gT
" Alt-k to move to the tab to the right
noremap  gt
```

To 'c'lose a 'tab', run `:tabc` or `:q` .

You can even open text that opens in a new window to open in a new tab instead. For example, `:help tabpage` opens the help in a horizontally split window. To view it in a new tab instead, use `:tab help tabpage` .

If you want to reorder the tabs, use `:tabmove` . For example, to move the current tab to the first position, use `:tabmove 0` and so on.

See `:help tabpage` for more details on tab pages and the other operations that you can run, such as `:tabdo` to operate on each of the tab pages which are open, and customizing the title of the tab pages (`:help setting-guitablabel`), etc.

Summary

Vim provides a number of ways to edit multiple files at the same time - buffers, windows and tabs. Using these features depends on your personal habit. For example, using multiple tabs may obviate the usage of multiple windows. It's important to use the one which is most convenient and comfortable.

Personal Information Management

A chapter on 'personal information management' (PIM) in a book on an editor software seems strange, doesn't it? Well, there are lots of "professional software" that claim to do personal information management, so let us explore why can't we use a plain text editor like Vim for this purpose?

Personal information management is about organizing all your "information" - such as your todo lists, diary entries, your reference material (such as important phone numbers), scratchpad and so on. Putting all of this in one convenient location can be extremely handy, and we will explore this using Vim and a few plugins.

I tend to think of a PIM system is best organized as a wiki. A wiki is a quick way to link together various documents which are inter-related but are independent in their own right. Unsurprisingly, the word 'wiki' means 'quick' in the Hawaiian language. Think of a website - there is a home page, and there are related pages to which you see links, and each page will have its own content but can also inter-link to other pages. Isn't this an easy way of organizing websites? What if you could do the same for your own personal information? See this LifeHack article 'Wikify Your Life: How to Organize Everything' on some great examples on what you can do.

But does this really require a specialized Wiki software? What if you could do the same in just plain text files using Vim? Let's dive in.

Installing Viki

> NOTE: The `$vimfiles` directory corresponds to `~/.vim` on Linux/Mac, `C:/Documents and Settings/<your-user-name>/vimfiles` on Windows and `C:/Users/<your-user-name>/vimfiles` on Windows Vista. See `:help vimfiles` for specific details.

We're going to install Viki and its related plugins:

1. Download multvals.vim and store as `$vimfiles/plugin/multvals.vim` .
2. Download genutils.zip and unzip this file to `$vimfiles` .
3. Download Viki.zip and unzip this file to `$vimfiles` (make sure all the folders and files under the 'Viki' folder name are stored directly in the `$vimfiles` folder)

Get Started

1. Open the GUI version of Vim
2. `:e test.txt`
3. `:set filetype=viki`
4. Type the following text:

```
[[http://deplate.sourceforge.net/Markup.html][Viki syntax]]
```

5. `:w`

6. Position your cursor on the above text and press `ctrl+enter`, or alternatively press `\vf`.
7. You should see a web browser open up with the above website page open.

Similarly, you can write down any file name (with a valid path) - whether it is a `.doc` file or a `.pdf` file and then you can `ctrl+enter` to open the file in the corresponding Word or Acrobat Reader programs!

The idea is that you can use plain text files to *hold* all your thinking together and you can `ctrl+enter` your way into everything else.

Now, notice that we had to type the square brackets in pairs above to identify the target of the link and the words that describe the link. This is basically the syntax of the markup language which we will explore next.

Markup language

The Viki syntax page (that you just opened in your web browser) explains how to write the text to allow Viki to syntax highlight portions of your text as well as how to do the linking between 'wiki pages' and even write Viki-specific comments.

Learning the basics of the syntax highlighting is useful because you can visually see the parts of your text file. For example, use `* List of things to do` to make it a header, and then use dashes to create a list:

```
 * List of things to do

  - Finish the blog post on Brahmagiri trek
  - Fix footer bug on IONLAB website
  - Buy some blank CDs
  - Get motorbike serviced
```

Disabling CamelCase

Writing `CamelCase` can create a wiki link in Viki, but I personally dislike this. I prefer that only explicit links like `[[CamelCase]]` be allowed to avoid situations where I have genuinely used a name which uses camel case but I don't want it to be a link (for example, the word "JavaScript"). To disable camel-case syntax, put `let g:vikiNameTypes = "sSeuix"` in your `~/.vimrc` file.

Getting Things Done

One of the major reasons for creating this 'viki' for myself is to maintain a 'Getting Things Done' system.

Getting Things Done ("GTD") is a system devised by David Allen to help manage your 'stuff' - which could mean anything from your career plans to the list of chores you have to do today.

From David Allen's book:

> "Get everything out of your head. Make decisions about actions required on stuff when it shows up - not when it blows up. Organize reminders of your projects and the next actions on them in appropriate categories. Keep your system current, complete, and reviewed sufficiently to trust your intuitive choices about what you're doing (and not doing) at any time."

The GTD system basically consists of organizing your information into certain pages/folders:

1. Collection Basket
2. Projects List
3. Next Actions
4. Calendar
5. Someday/Maybe
6. Reference Material
7. Waiting For

I created a viki to match this system by using the following method:

1. First, create a `StartPage` which is literally the start page to your personal organization system (hereby referred to as simply "your viki").
2. Then, create a list of main sections of your viki:

```
* Getting Things Done

1. [[Collect][In Basket]]
2. [[Project][Projects List]]
3. [[NextActions][Next Actions]]
4. [[Calendar]]
5. [[SomedayMaybe][Someday/Maybe]]
6. [[Reference][Reference Material]]
7. [[Waiting][Waiting For]]
```

3. Similarly, go to as much depth as you want, for example creating a `[[Reference.Career]]` to jot down your career plans, and `[[Project.TopSecret]]` to gather thoughts on your next project, and so on.

4. Every time you want to jot down something, use the `[[Collect]]` page and then process, organize, review and finally actually do your next-physical-actions.

5. It takes a while to get accustomed to using this system, but once you are comfortable, you can achieve clarity of mind, confidence that you're taking care of all the factors in your life, and most importantly, a sense of direction in knowing what are the important things in your life.

Notice how we are managing an entire system using just plain text!

Summary

We have just explored how Vim can help you in creating a personal information management system for yourself. It's fascinating how we don't need a complicated software for such a system, just plain text files and Vim will do.

See Abhijit Nadgouda's article on using Vim as a personal wiki for an alternative way of achieving the same using built-in Vim functionality.

Scripting

If you want to customize any software, most likely you will change the various settings in the software to suit your taste and needs. What if you wanted to do more than that? For example, to check for conditions such as `if GUI version, then use this colorscheme else use this colorscheme` ? This is where "scripting" comes in. Scripting basically means using a language where you can specify conditions and actions put together into 'scripts'.

There are two approaches to scripting in Vim - using the built-in Vim scripting language, or using a full-fledged programming language such as Python or Perl which have access to the Vim internals via modules (provided that Vim has been compiled with these options enabled).

This chapter requires some knowledge of programming. If you have no prior programming experience, you will still be able to understand although it might seem terse. If you wish to learn programming, please refer my other free book A Byte of Python.

There are two ways of creating reusable functionality in Vim - using macros and writing scripts.

Macros

Using macros, we can record sequences of commands and then replay it in different contexts.

For example, suppose you had some text like this:

```
tansen is the singer
daswant is the painter
todarmal is the financial wizard
abul fazl is the historian
birbal is the wazir
```

There are many things to correct here:

1. Change the first letter of the sentence to upper case.
2. Change 'is' to 'was'.
3. Change 'the' to 'a'.
4. End the sentence with "in Akbar's court."

One way would be to use a series of substitute commands, such as `:s/^\\w/\\u\\0/` but this would require 4 substitution commands and it might not be simple if the substitute command changes parts of the text which we do not want to be changed.

A better way would be to use macros.

1. Position your cursor on the first letter of the first line: `tansen is the singer`
2. Type `qa` in normal mode to start recording the macro named as `a`.
3. Type `gU1` to switch the first letter to upper case.
4. Type `w` to move to the next word.
5. Type `cw` to change the word.
6. Type `was`.
7. Press `<Esc>`.
8. Type `w` to move to the next word.
9. Type `cw` to change the word.
10. Type `a`.
11. Press `<Esc>`.
12. Type `A` to insert text at the end of the line.
13. Type `in Akbar's court`.
14. Press `<Esc>`.
15. Type `q` to stop recording the macro.

That looks like a long procedure, but sometimes, this is much easier to do than cook up some complicated substitute commands!

At the end of the procedure, the line should look like this:

```
Tansen was a singer in Akbar's court.
```

Great. Now, let's move on to apply this to the other lines. Just move to the first character of the second line and press `@a`. Voila, the line should change to:

```
Daswant was a painter in Akbar's court.
```

This demonstrates that macros can record complicated operations and can be easily replayed. This helps the user to replay complicated editing in multiple places. This is one type of reusing the manipulations you can do to the text. Next, we will see more formal ways of manipulating the text.

> NOTE: If you want to simply repeat the last action and not a sequence of actions, you do not have to use macros, just press `.` (dot).

Basics of Scripting

Vim has a built-in scripting language using which you can write your own scripts to take decisions, "do" stuff, and manipulate the text.

Actions

How do you change the theme i.e. colors used by Vim? Just run:

```
:colorscheme desert
```

Here, I am using the 'desert' color scheme, which happens to be my favorite. You can view the other schemes available by typing `:colorscheme` and then pressing `<tab>` key to cycle through the available schemes.

What if you wanted to know how many characters are in the current line?

```
:echo strlen(getline("."))
```

Notice the names 'strlen' and 'getline'. These are "functions". *Functions* are pieces of scripts already written and have been given a name so that we can use them again and again. For example, the getline function fetches a line and we are indicating which line by the `.` (dot) which means the current line. We are passing the result returned by the `getline` function to the strlen function which counts the number of characters in the text and then we are passing the result returned by the `strlen` function to the `:echo` command which simply displays the result. Notice how the information flows in this command.

The `strlen(getline("."))` is called an expression. We can store the results of such expressions by using variables. Variables do what the name suggests - they are names pointing to values and the value can be anything i.e. it can vary. For example, we can store the length as:

```
:let len = strlen(getline("."))
:echo "We have" len "characters in this line."
```

When you run this line on the second line above in this text, you will get the following output:

```
We have 46 characters in this line.
```

Notice how we can use variables in other 'expressions'. The possibilities of you can achieve with the help of these variables, expressions and commands are endless.

Vim has many types of variables available via prefixes such as `$` for environment variables, `&` for options, and `@` for registers:

```
:echo $HOME
:echo &filetype
:echo @a
```

See `:help function-list` for a huge list of functions available.

You can create your own functions as well:

```
:function CurrentLineLength()
: let len = strlen(getline("."))
: return len
:endfunction
```

Now position your cursor on any line and run the following command:

```
:echo CurrentLineLength()
```

You should see a number printed.

Function names have to start with an upper case. This is to differentiate that built-in functions start with a lower case and user-defined functions start with an upper case.

f you want to simply "call" a function to run but not display the contents, you can use `:call CurrentLineLength()`

Decisions

Suppose you want to display a different color schemes based on whether Vim is running in the terminal or is running as a GUI i.e. you need the script to take decisions. Then, you can use:

```
:if has("gui_running")
: colorscheme desert
:else
: colorscheme darkblue
:endif
```

How It Works:

- `has()` is a function which is used to determine if a specified feature is supported in Vim

installed on the current computer. See `:help feature-list` to see what kind of features are available in Vim.

- The `if` statement checks the given condition. If the condition is satisfied, we take certain actions. "Else", we take the alternative action.
- Note that an `if` statement should have a matching `endif`.
- There is `elseif` also available, if you want to chain together multiple conditions and actions.

The looping statements 'for' and 'while' are also available in Vim:

```
:let i = 0
:while i < 5
: echo i
: let i += 1
:endwhile
```

Output:

```
0
1
2
3
4
```

Using Vim's built-in functions, the same can also be written as:

```
:for i in range(5)
:    echo i
:endfor
```

- `range()` is a built-in function used to generate a range of numbers. See `:help range()` for details.
- The `continue` and `break` statements are also available.

Data Structures

Vim scripting also has support for lists and dictionaries. Using these, you can build up complicated data structures and programs.

```
:let fruits = ['apple', 'mango', 'coconut']

:echo fruits[0]
" apple

:echo len(fruits)
" 3

:call remove(fruits, 0)
:echo fruits
" ['mango', 'coconut']

:call sort(fruits)
:echo fruits
" ['coconut', 'mango']

:for fruit in fruits
: echo "I like" fruit
:endfor
" I like coconut
" I like mango
```

There are many functions available - see 'List manipulation' and 'Dictionary manipulation' sections in `:help function-list` .

Writing a Vim script

We will now write a Vim script that can be loaded into Vim and then we can call its functionality whenever required. This is different from writing the script inline and running immediately as we have done all along.

Let us tackle a simple problem - how about capitalizing the first letter of each word in a selected range of lines? The use case is simple - When I write headings in a text document, they look better if they are capitalized, but I'm too lazy to do it myself. So, I can write the text in lower case, and then simply call the function to capitalize.

We will start with the basic template script. Save the following script as the file capitalize.vim:

```
" Vim global plugin for capitalizing first letter of each word
"        in the current line.
" Last Change: 2008-11-21 Fri 08:23 AM IST
" Maintainer: www.swaroopch.com/contact/
" License: www.opensource.org/licenses/bsd-license.php


if exists("loaded_capitalize")
  finish
endif
let loaded_capitalize = 1


" TODO : The real functionality goes in here.
```

How It Works:

- The first line of the file should be a comment explaining what the file is about.
- There are 2-3 standard headers mentioned regarding the file such as 'Last Changed:' which explains how old the script is, the 'Maintainer:' info so that users of the script can contact the maintainer of the script regarding any problems or maybe even a note of thanks.
- The 'License:' header is optional, but highly recommended. A Vim script or plugin that you write may be useful for many other people as well, so you can specify a license for the script. Consequently, other people can improve your work and that it will in turn benefit you as well.
- A script may be loaded multiple times. For example, if you open two different files in the same Vim instance and both of them are `.html` files, then Vim opens the HTML syntax highlighting script for both of the files. To avoid running the same script twice and redefining things twice, we use a safeguard by checking for existence of the name 'loaded_capitalize' and closing if the script has been already loaded.

Now, let us proceed to write the actual functionality.

We can define a function to perform the transformation - capitalize the first letter of each word, so we can call the function as `Capitalize()`. Since the function is going to work on a range, we can specify that the function works on a range of lines.

```
" Vim global plugin for capitalizing first letter of each word
"       in the current line
" Last Change: 2008-11-21 Fri 08:23 AM IST
" Maintainer: www.swaroopch.com/contact/
" License: www.opensource.org/licenses/bsd-license.php

" Make sure we run only once
if exists("loaded_capitalize") finish
endif
let loaded_capitalize = 1

" Capitalize the first letter of each word
function Capitalize() range
  for line_number in range(a:firstline, a:lastline)
    let line_content = getline(line_number)
    " Luckily, the Vim manual had the solution already!
    " Refer ":help s%" and see 'Examples' section
    let line_content = substitute(line_content, "\\w\\+", "\\u\\0", "g")
    call setline(line_number, line_content)
  endfor
endfunction
```

How It Works:

- The `a:firstline` and `a:lastline` represent the arguments to the function with correspond to the start and end of the range of lines respectively.
- We use a 'for' loop to process each line (fetched using `getline()`) in the range.
- We use the `substitute()` function to perform a regular expression search-and-replace on the string. Here we are specifying the function to look for words which is indicated by `\\w\\+` which means a word (i.e. a continuous set of characters that are part of words). Once such words are found, they are to be converted using `\\u\\0` - the `\\u` indicates that the first character following this sequence should be converted to upper case. The `\\0` indicates the match found by the `substitute()` function which corresponds to the words. In effect, we are converting the first letter of each word to upper case.
- We call the `setline()` function to replace the line in Vim with the modified string.

To run this command:

1. Open Vim and enter some random text such as 'this is a test'.
2. Run `:source capitalize.vim` - this 'sources' the file as if the commands were run in Vim inline as we have done before.
3. Run `:call Capitalize()` .
4. The line should now read 'This Is A Test'.

Running `:call Capitalize()` every time appears to be tedious, so we can assign a keyboard shortcut using leaders:

```
" Vim global plugin for capitalizing first letter of each word
" in the current line
" Last Change: 2008-11-21 Fri 08:23 AM IST
" Maintainer: www.swaroopch.com/contact/
" License: www.opensource.org/licenses/bsd-license.php

" Make sure we run only once
if exists("loaded_capitalize")
  finish
endif
let loaded_capitalize = 1

" Refer ':help using-'
if !hasmapto('Capitalize')
  map  c Capitalize
endif
noremap
```

- We have changed the name of the function from simply `Capitalize` to `s:Capitalize` - this is to indicate that the function is local to the script that it is defined in, and it shouldn't be available globally because we want to avoid interfering with other scripts.
- We use the `map` command to define a shortcut.
- The `<Leader>` key is usually backslash, `\` .
- We are now mapping `<Leader>c` (i.e. the leader key followed by the 'c' key) to some functionality.
- We are using `<Plug>Capitalize` to indicate the `Capitalize()` function described within a plugin i.e. our own script.
- Every script has an ID, which is indicated by `<SID>` . So, we can map the command `<SID>Capitalize` to a call of the local function `Capitalize()` .

So, now repeat the same steps mentioned previously to test this script, but you can now run `\c` to capitalize the line(s) instead of running `:call Capitalize()` .

This last set of steps may seem complicated, but it just goes to show that there's a wealth of possibilities in creating Vim scripts and they can do complex stuff.

If something does go wrong in your scripting, you can use `v:errmsg` to see the last error message which may give you clues to figure out what went wrong.

Note that you can use `:help` to find help on everything we have discussed above, from `:help \w` to `:help setline()` .

Using external programming languages

Many people would not like to spend the time in learning Vim's scripting language and may prefer to use a programming language they already know and write plugins for Vim in that language. This is possible because Vim supports writing plugins in Python, Perl, Ruby and many other languages.

In this chapter, we will look at a simple plugin using the Python programming language, but we can easily use any other supported language as well.

As mentioned earlier, if you are interested in learning the Python language, you might be interested in my other free book A Byte of Python.

First, we have to test if the support for the Python programming language is present.

```
:echo has("python")
```

If this returns `1`, then we are good to go, otherwise you might want to install Python on your machine and try again.

Suppose you are writing a blog post. A blogger usually wants to get as many people to read his/her blog as possible. One of the ways people find such blog posts is by querying a search engine. So, if you're going to write on a topic (say 'C V Raman', the famous Indian physicist who has won a Nobel Prize for his work on the scattering of light), you might want to use important phrases that helps more people find your blog when they search for this topic. For example, if people are searching for 'c v raman', they might also search for the 'raman effect', so you may want to mention that in your blog post or article.

How do we find such related phrases? It turns out that the solution is quite simple, thanks to Yahoo! Search.

First, let us figure out how to use Python to access the current text, which we will use to generate the related phrases.

```
" Vim plugin for looking up popular search queries related
"        to the current line
" Last Updated: 2008-11-21 Fri 08:36 AM IST
" Maintainer: www.swaroopch.com/contact/
" License: www.opensource.org/licenses/bsd-license.php


" Make sure we run only once
if exists("loaded_related")
  finish
endif
let loaded_related = 1


" Look up Yahoo Search and show results to the user
function Related()
python <
```

The main approach to writing plugins in interfaced languages is same as that for the built-in scripting language.

The key difference is that we have to pass on the code that we have written in the Python language to the Python interpreter. This is achieved by using the EOF as shown above - all the text from the `python <<EOF` command to the subsequent `EOF` is passed to the Python interpreter.

You can test this program, by opening Vim again separately and running `:source related.vim`, and then run `:call Related()`. This should display something like `Length of the current line is 54`.

Now, let us get down the actual functionality of the program. Yahoo! Search has something called a RelatedSuggestion query which we can access using a web service. This web service can be accessed by using a Python API provided by Yahoo! Search pYsearch:

```
" Vim plugin for looking up popular search queries related
" to the current line
" Last Updated: 2008-11-21 Fri 08:36 AM IST
" Maintainer: www.swaroopch.com/contact/
" License: www.opensource.org/licenses/bsd-license.php


" Make sure we run only once
if exists("loaded_related")
  finish
endif
let loaded_related = 1


" Look up Yahoo Search and show results to the user
function Related()
python <
```

Notice that we use the current line in Vim as the current text we are interested in, you can change this to any text that you want, such as the current word, etc.

We use the `yahoo.search.web.RelatedSuggestion` class to query Yahoo! Search for phrases related to the query that we specify. We get back the results by calling `parse_results()` on the result object. We then loop over the results and display it to the user.

1. Run `:source related.vim`
2. Type the text `c v raman` .
3. Run `:call Related()`
4. The output should look something like this:

```
Related popular searches are:
1. raman effect
2. c v raman india
3. raman research institute
4. chandrasekhara venkata raman
```

Summary

We have explored scripting using the Vim's own scripting language as well as using external scripting/programming languages. This is important because the functionality of Vim can be extended in infinite ways.

See `:help eval` , `:help python-commands` , `:help perl-using` and `:help ruby-commands` for details.

To dive deep into this topic, see Learn VimScript The Hard Way by Steve Losh.

Plugins

As we have seen in the previous chapter, we can write scripts to extend the existing functionality of Vim to do more stuff. We call these scripts which extend or add functionality as "plugins."

There are various kinds of plugins that can be written:

- vimrc
- global plugin
- filetype plugin
- syntax highlighting plugin
- compiler plugin

Not only can you write your own plugins but also download and use plugins written by others.

Customization using vimrc

When I install a new Linux distribution or reinstall Windows, the first thing I do after installing Vim is fetch my latest `vimrc` file from my backups, and then start using Vim. Why is this important? Because the `vimrc` file contains various customizations/settings I like which makes Vim more useful and comfortable for me.

There are two files you can create to customize Vim to your taste:

1. vimrc - for general customizations
2. gvimrc - for GUI specific customizations

These are stored as:

- `%HOME%/_vimrc` and `%HOME%/_gvimrc` on Windows
- `$HOME/.vimrc` and `$HOME/.gvimrc` on Linux/BSD/Mac OS X

See `:help vimrc` on the exact location on your system.

These vimrc and gvimrc files can contain any Vim commands. The convention followed is to use only simple settings in the vimrc files, and advanced stuff are sourced from plugins.

For example, here's a portion of my vimrc file:

```
" My Vimrc file
" Maintainer: www.swaroopch.com/contact/
```

```
" Reference: Initially based on http://dev.gentoo.org/~ciaranm/docs/vim-guide/
" License: www.opensource.org/licenses/bsd-license.php

" Use Vim settings, rather then Vi settings (much better!).
" This must be first, because it changes other options as a side effect.
set nocompatible


" Enable syntax highlighting.
syntax on

" Automatically indent when adding a curly bracket, etc.
set smartindent

" Tabs should be converted to a group of 4 spaces.
" This is the official Python convention
" http://www.python.org/dev/peps/pep-0008/
" I didn't find a good reason to not use it everywhere.
set shiftwidth=4
set tabstop=4
set expandtab
set smarttab

" Minimal number of screen lines to keep above and below the cursor.
set scrolloff=999

" Use UTF-8.
set encoding=utf-8

" Set color scheme that I like.
if has("gui_running")
  colorscheme desert
else
  colorscheme darkblue
endif

" Status line
set laststatus=2
set statusline=
set statusline+=%-3.3n\                    " buffer number
set statusline+=%f\                        " filename
set statusline+=%h%m%r%w                   " status flags
set statusline+=\[%{strlen(&ft)?&ft:'none'}] " file type
set statusline+=%=                         " right align remainder
set statusline+=0x%-8B                      " character value
set statusline+=%-14(%l,%c%V%)             " line, character
set statusline+=%<%P "="" file="" position="" Show="" line="" number,="" cursor="" positi
imap  =strftime("%Y-%m-%d %a %I:%M %p")

" To save, press ctrl-s.
nmap  :w
imap  :wa
```

```
" Search as you type.
set incsearch

" Ignore case when searching.
set ignorecase

" Show autocomplete menus.
set wildmenu

" Show editing mode
set showmode

" Error bells are displayed visually.
set visualbell
```

Notice that these commands are not prefixed by colon. The colon is optional when writing scripts in files because it assumes they are normal mode commands.

If you want to learn detailed usage of each setting mentioned above, refer `:help`.

A portion of my gvimrc file is:

```
" Size of GVim window
set lines=35 columns=99

" Don't display the menu or toolbar. Just the screen.
set guioptions-=m
set guioptions-=T

" Font.
if has('win32') || has('win64')
  " set guifont=Monaco:h16
  " http://jeffmilner.com/index.php/2005/07/30/windows-vista-fonts-now-available/
  set guifont=Consolas:h12:cANSI
elseif has('unix')
  let &guifont="Monospace 10"
endif
```

There are various example vimrc files out there that you should definitely take a look at and learn the various kinds of customizations that can be done, then pick the ones you like and put it in your own vimrc.

A few good ones that I have found in the past are:

- Steve Francia's vim distribution
- Steve Losh's vim config
- bling's dotvim
- Janus

It is a known fact that a person's vimrc usually reflects how long that person has been using Vim.

Global plugin

Global plugins can be used to provide global/generic functionality.

Global plugins can be stored in two places:

1. `$VIMRUNTIME/plugin/` for standard plugins supplied by Vim during its installation
2. To install your own plugins or plugins that you have download from somewhere, you can use your own plugin directory:

    - `$HOME/.vim/plugin/` on Linux/BSD/Mac OS X
    - `%HOME%/vimfiles/plugin/` on Windows
    - See `:help runtimepath` for details on your plugin directories.

Let's see how to use a plugin.

A useful plugin to have is `highlight_current_line.vim` by Ansuman Mohanty which does exactly as the name suggests. Download the latest version of the file `highlight_current_line.vim` and put it in your plugin directory (as mentioned above). Now, restart Vim and open any file. Notice how the current line is highlighted compared to the other lines in the file.

But what if you don't like it? Just delete the `highlight_current_line.vim` file and restart Vim.

Similarly, we can install our own `related.vim` or `capitalize.vim` from the Scripting chapter into our plugin directory, and this avoids us the trouble of having to use :source every time. Ultimately, any kind of Vim plugin that you write should end up somewhere in your `.vim/vimfiles` plugin directory.

Filetype plugin

Filetype plugins are written to work on certain kinds of files. For example, programs written in the C programming language can have their own style of indentation, folding, syntax highlighting and even error display. These plugins are not general purpose, they work for these specific filetypes.

Using a filetype plugin

Let's try a filetype plugin for XML. XML is a declarative language that uses tags to describe the structure of the document itself. For example, if you have a text like this:

> Iron Gods
> ---------
>
> Ashok Banker's next book immediately following the Ramayana is said to be a novel tentatively titled "Iron Gods" scheduled to be published in 2007. A contemporary novel, it is an epic hard science fiction story about a war between the gods of different faiths. Weary of the constant infighting between religious sects and their deities, God (aka Allah, Yahweh, brahman, or whatever one chooses to call the Supreme Deity) wishes to destroy creation altogether.
>
> A representation of prophets and holy warriors led by Ganesa, the elephant-headed Hindu deity, randomly picks a sample of mortals, five of whom are the main protagonists of the book--an American Catholic, an Indian Hindu, a Pakistani Muslim, a Japanese Buddhist, and a Japanese Shinto follower. The mortal sampling, called a 'Palimpsest' is ferried aboard a vast Dyson's Sphere artifact termed The Jewel, which is built around the sun itself, contains retransplanted cities and landscapes brought from multiple parallel Earths and is the size of 12,000 Earths. It is also a spaceship travelling to the end of creation, where the Palimpsest is to present itself before God to plead clemency for all creation.
>
> Meanwhile, it is upto the five protagonists, aided by Ganesa and a few concerned individuals, including Lucifer Morningstar, Ali Abu Tarab, King David and his son Solomon, and others, to bring about peace among the myriad warring faiths. The question is whether or not they can do so before the audience with God, and if they can do so peacefully--for pressure is mounting to wage one final War of Wars to end all war itself.

(Excerpt taken from Wikipedia under the GNU Free Documentation License)

It can be written in XML form (specifically, in 'DocBook XML' format) as: