Vim Start to End

Projit Bandyopadhyay

April 14, 2020

1 Why Vim is Awesome

No one knows how to start

So the sad part is that most online tutorials on vim focus only on the absolute basics of vim, and not why vim is awesome. To a beginner it looks scary as

$$:wq$$

and

$$dd$$

are not exactly the most intuitive things in vim. If you're reading this document, I assume you probably know those already, and are actually looking to figure out why people rave about vim, and what the true power of this "text editor" is.

The philosophy behind vim is quite awesome as well. The idea is that we spend more time *editing* code rather than acutally writing it. So vim is configured and geared towards making that process easy. Moving things around, making quick changes, and basically everything else under the sun (of editing), is what vim is made for.

Vim has an insane amount of funtionality in-built without any plugins whatsoever. Hopefully reading this will show you that, and help you appreciate how useless most plugins are xD, and how awesome vim is out-of-the-box.

2 Basic Basics

This section has been added to include all commonly use commands. You don't need to read this part too well, but it's included for the sake of com-

pletion. Skim through it maybe, to see useful shortcuts here and there.

2.1 Modes

3 Modes in Vim: Insert, Normal, Visual

When you first open Vim (vim <somedocument>) you are in Normal Mode. This is the mode that you will stay in to edit.

Normal to Insert	i
Insert to Normal	Esc
Normal to Visual	v, Ctrl v, V
Visual to Normal	Esc

Basically if you're confused and don't know what's going on press *Esc* repeatedly till it goes back to normal :)

Little more on Visual:

Command	Description
v	enter visual mode, then can move around with navigation.
V	enter visual line mode, then can move around up and down
Ctrl V	enter visual block mode, can move around with navigation

2.2 Navigation

You never want to be navigating around in insert mode, always use normal mode to move around (you'll see why later)

In Normal Mode:

Command	Description
j	down one line
k	up one line
h	left one character
l	right one character
0	start of the line
$	end of the line
w	move to beginning of next (w)ord
W	move to beginning of next (w)ord after a whitespace
b	move to (b)ackward to beginning of word
B	move (b)ackward by word to next whitespace
e	move to (e)nd of word
E	move to (e)nd of word til whitespace
G	move to end of the file
gg	move to start of the file
Ctrl u	move half screen (u)p
Ctrl d	move half screen (d)own
Ctrl e	scroll downwards without moving cursor
Ctrl y	scroll upwards without moving cursor
zt	move current line (t)op of screen
zb	move current line (b)ottom of screen
zz	move current line center of screen
^	move to first non whitespace character in line (useful while coding)
‘.	move to last edit point
f<char>	move (f)orward to first character in that line which matches that character
F<char>	move backward to first character in that line which matches that character
t<char>	move forward (t)il first character in that line which matches that character, but stop one character before
T<char>	move backward (t)il first character in that line which matches that character, but stop one character in front
/text	searches forward for text in doc. n and N to navigate
?text	searches backward for text in doc. n and N to navigate

2.3 Editing

Some commands to be used while editing in vim, these will make more sense after the next section, but for the sake of some basic editing skills. **These will be greatly increased with the next section**

Note: All of these take a little bit of practice to get use to their full functionality

These all only work in Normal mode:

Command	Description
i	enter insert mode at current position
I	enter insert mode at start of line
a	enter insert mode one character in front of cursor
A	enter insert mode at the end of the line
dd	delete current line
d$	delete current point til end of line
yy	yank current line
p	paste from current point
P	paste before this point
x	delete character under cursor
rA	replace character under cursor with A (change A)
u	undo last change
Ctrl r	redo
.	repeat last change made
:w	save file
:q	quite file
:q!	quite with force (discard changes)
:wq	save and quit

When using vim from a terminal, a common workflow is to type :w and then Ctrl z.

After executing whatever needs to be done on the terminal, use fg to come back into vim. This retains your position and other buffers / splits you may have created

2.4 Fancy

You see other "cool" people using fancy splits and stuff. here are the basics on how to use those.

Command	Description
:vs <file?>	(v)ertical (s)plit current window. Add file to open that file as well
:sp <file?>	horizontal (sp)lit current window. Add file to open that file as well
<ctrl w>	use j,k,l,h along with this to move in that direction, to navigate splits
<ctrl w> <ctrl w>	cycle through windows
:q	close split you're in right now
:only	close all other splits but current
:set scrollbind	do this in each of the windows if you want them to scroll together (useful when comparing code)
<ctrl w> [N] -	Decrease current window height by N (default 1)
<ctrl w> [N] +	Increase current window height by N (default 1)
<ctrl w> [N] <	Decrease current window width by N (default 1)
<ctrl w> [N] >	Increase current window width by N (default 1)
<ctrl w>	maximize current window vertically
<ctrl w> |	maximize current window horizontally
<ctrl w> =	make all equal size

Buffers in vim are awesome though learning to use them can be a bit unintuitive. People are always like where did that file disappear to when I opened this new file with :e or something.

What's interesting is that internally, splits and even tabs are essentially buffers. All that changes is how you view them.

Command	Description
:ls	list all buffers
:e filename	can tab-complete the filename, but opens the file in current buffer making your other file "disappear". :ls to see that it's still there.
:q	quits out of vim, AND all buffers. Can be used to close current split
:bn	navigate to (n)ext buffer
:bp	navigate to (p)revious buffer
:b<num>	navigate to buffer number <num>
:b<partial>	navigate to buffer with name where partial is a substring
:bd	delete current buffer; fails if unsaved. add ! to override

Let's say you've come from sublime or something, and just can't live without tabs. Well that's there too... (I don't use tabs very much so credits to `https://vim.fandom.com/wiki/Using_tab_pages` for the info, and sorry if anything important is missing)

Command	Description
:tabs	list all tabs including their displayed windows
:tabm 0	move current tab to first
:tabm	move current tab to last
:tabm i	move current tab to position i+1
:tabn	go to next tab
:tabp	go to previous tab
:tabfirst	go to first tab
:tablast	go to last tab
gt	go to next tab
gT	go to previous tab
igt	go to tab in position i
:tab split	copy the current window to a new tab of its own
:tabedit file	edit specified file in a new tab
:tabfind file	open a new tab with filename given, searching the 'path' to find it
:tabclose	close current tab
:tabclose i	close i-th tab
:tabonly	close all other tabs (show only the current tab)
:tab split	copy the current window to a new tab of its own

Congrats! You now know enough vim to impress your friends and think you're proficient at it. The beauty in a tool isn't learning how to use the basics of it though. Hopefully you read on :)

3 Vim as a Composable Tool

What are Commands?

Everything from inserting text, to the command "commands" that we execute are considered commands. How can you tell?

Try typing something in insert mode and then press Esc. If you type a . now, you'll see that it repeats

The . operator repeats your last command.

Therefore the undo and redo actions just go through the commands issued...no magic in why "random" chunks of text disappear when you press undo.

To vim, entering insert mode and then typing is one command on the whole....which begs the question...what are commands?

3.1 Commands, the heart of Vim

Simply put (and this is my own take, so please don't kill me if it isn't exact :)), a command comprises a *noun* and a *verb*.

Ex. in *dw* the verb is [d] while the noun is [w].

Thus to create any command in vim, all you need to know is to learn what the nouns are, and what the verbs are. And this is the crux of the beauty of the design of vim. If you know 5 nouns and 5 verbs..you can create 25 commands. And the fun thing about vim is that the nouns and verbs are extremely intuitive.

Thus the "right" way to use vim is to first think about the action you want to do, and then formulate the command for it based on your need.

So when you type in a verb you'll notice that vim kind of "waits", for you to input the noun. It then takes the verb and applies it onto that noun.

In proper vim jargon, these commands are created by:

$$\{operator\}\{motion\}$$

3.2 Operators

Popular Operators	Description
d	(d)elete
c	(c)hange (delete and go to insert mode)
>	right shift (indent)
<	left shift (un-indent)
v	(v)isually select
y	(y)ank
p	(p)ut (paste) after cursor. Not an operator in the conventional sense but still added here .
g~	toggle case (tildy looks like a wave going up and down)
gu	lower case
gU	(U)pper case
=	format code

Some comments on the operators:

- As a regular vim user, you may actually be aware of all of these operators, having seen them as part of some of the basic commands. There exist more, but these are the commonly used ones.
- doubling an operator makes it operate on a whole line (ex. dd, cc, yy, gUU, guu). This isn't applicable to all the operators though
- Capitalizing an operator makes it do a "bolder" version of what it does now. A bit unpredictable, but not necessary once we se the motions.
- i (insert), a (append), I (insert start of line), A (append at end of line) are also operators. Though we normall only use them in going to insert mode from normal mode, they can be applied on text objects(try using visual box and move down a few lines, then press I, type something and press Esc.)

D	deletes til end of line
C	changes til end of line
Y	copies line
P	Pastes before cursor

3.3 Motions and More

The "noun" part of the command construct can be quite nuanced. There are of course easy nouns, and a little more complicated nouns.

Motion	Description
h,j,k,l	left, down, up, right
w,W	to start of next word or WORD
b,B	to start of previous word or WORD
e,E	to end of word or WORD
$	to end of line
^	to first non-empty character in line
G	end of document
gg	start of document
f<char>	til (and including) <char>
t<char>	til <char>
/text	til text forwards
?text	til text backwards

Each of the operators can also take a count before them, to increase their effectiveness.

$$\{count\}\{motion\}$$

While count can't be applied to all motions, it can to many.

Examples:

- `d2j`

  - delete 2 lines (current and below)

- `c2w`

  - change 2 words

- `3k`

  - move up 3 lines

- `7l`

  - move right 7 characters

So these motions can be used by itself to navigate the document while in normal mode. But coupled with an operator you can do cool things.

- `df"`

  - delete til and including "

- `c/goal<CR>`

  - This would change everything til the word goal. <CR> is carriage return, which is just press enter.

- d2w - delete 2 words

3.4 Text Objects

Here comes the holy grail of this class of things. **Text Objects**

For the sake of simplicity of understanding, let's say text objects can be the count + motion or this other class that we're about to describe.

A little more intuitively however, motions can basically move you around as well, while text objects are just defined without any particular movement as well

Take a look at two main modifiers:

- `i`

  - inner

- `a`

  - around

These are sometimes called modifiers, which are added to other motions/objects.

Plugins can be used to create new objects on which these could be applied...check it out :)

For now, look at the awesome objects that you have access to with these

Text Object	Description
iw	inner word
it	inner tag (between <tag>text</tag>)
i"	inner quotes
i'	inner quotes
ip	inner paragraph
aw	around word
as	around sentence
a'	around quotes (including the quotes essentially)

You can essentially do inner <anything> and around <anything> try:

- `i\{`
- `i<`
- `i>  (same results as previous)`

You've probably understood that i and a can be used on pretty much all the same things. i usually implies inside and not anything else. a usually implies that the object's surroundings or the object specified itself is included.

Try them out to actually get the hang of them...but they're truly awesome.

So now for some truly magical stuff try combining

$$\{operator\}\{textobject\}$$

Examples:

- `ciw`
- `ci"`
- `di)`

The possiblities are endless :)

As mentioned before, research online on how to add in more text objects as well

3.5 Summary Thus Far

We've now seen that Vim is ridiculously composable. Wanna delete and replace a word? ciw. Your cursor could be anywhere on the object and it works!

You know what's even better? try ciw and then type something. Then go to another word and press .

It'll do the same operation?! Cause change puts us into insert mode, it kind of chains the command til the completion of that.

We've seen that any command can be created from just operators (verbs) and objects / motions (nouns).

Learning a few operators and a few objects allows you to compose them to do tons of stuff. Hope you've started to appreciate the beauty that is vim. Now what's the fun if you can't customize it?

4 How to Customize

4.1 The Horror that is Vimscript

So, vimscript sucks. Sure you could use other versions of vim that don't require this, so skip this part if you want.

But inevitably you may be forced to an unconfigured vim at some point, so the knowledge isn't half bad.

For better of for worse, vimscript is actually a full blown programming language. But for most of your daily vim needs, you probably don't need to know the whole of it.

So think of this as a minimal introduction to vimscript, with some barebones so that many operations don't look foreign, especially when editing your .vimrc (a file which can store settings and configurations)

:help is your best friend. Anything and everything you don't understand just do :help <query> and boom. It's surprisingly well written documentation.

So basically that bottom line where all the : commands are written, you're inserting vimscript there.

Try random stuff like:

- `:echo "hello world"`
- `:echom "hello world second time"`

- `To see all messages execute :messages`
- `" is considered to be a comment`

4.1.1 Boolean Options

Lot of random settings are boolean. If you read through everything you would've seen something called :set scrollbind.

This was setting a boolean value within vim.

Booleans in vim are weird. Generic form of accessing with some examples:

- `:set <name>`

  - Makes boolean value true

- `:set no<name>`

  - Makes boolean value false

- try

  ```
  :set number
  ```

  and

  ```
  :set nonumber
  ```

- `:set <name>!`

  - Toggles boolean value

- `:set <name>?`

  - Checks current state of boolean value

- `:set number numberwidth=4`

  - Can set multiple things at once

While executing things in vim we use the :

While adding these settings to your .vimrc file, the : can be omitted

Recommendation: Add *set relativenumber* to your vimrc. Let's you use motions much more easily. You'll be able to see where a motion like 3j or something would take you to.

4.2 Life Hacks

4.2.1 Mapping Keys

Mapping keys in vim is ridiculously easy.

$$: map - x$$

This would map hyphen to delete a character (the command x).

Mappings persist only in a vim session. To make them permanent you must add them to your .vimrc file (remove the : then)

You can map special keys as well. Ex. <c-d> would be [ctrl d], or <space> corresponds to the spacebar (the angular brackets must be present)

Given vim has multiple modes, you can create mode-specific mappings with :imap, :nmap, and :vmap ... corresponding to insert, normal, and visual modes. The usage is the same.

*map functions may recurse, which is why they're not used. Can this recurse?

```
:nmap dd 0<esc>jddk
```

To handle this issue we use noremap instead of map. Correspondingly, we have inoremap, nnoremap, and vnoremap. Noremap means no recursion map which prevents recursion from happening.

While mapping stuff, you usually dont have to many keys. SO you prefix your mappings with something, so that you can use it everywhere. This prefix we call a leader

```
Set by  :let mapleader = ','
Use by  :nnoremap <leader>d dd
```

There is also another leader in vim called localleader. works the same way as leader, just replace with localleader It's mainly used to mappings that take effect in only some types of files. Basically leader takes precedence over localleader, which is why localleader is often used while writing plugins.

Life hacks to quickly edit your vimrc from within vim:

```
Open up vimrc -
:nnoremap <leader>ev :vsplit \$MYVIMRC<cr>

Source vimrc -
:nnoremap <leader>sv :source \$MYVIMRC<cr>
```

Note: <cr> is used as carriage return. It's basically the equivalent of your enter/return key, in command form

To learn a new mapping that you've created you can noremap your old mapping to <nop> forcing you to use the new mapping

4.2.2 Autocommands

These are things which wait for an event and then execute something automatically.

The following would indent before reading or before writing to a html file:

```
:autocmd BufWritePre,BufRead *.html :normal gg=G
```

Filetypes can also be events:

```
:autocmd FileType python nnoremap <buffer> <localleader>c I\#<esc>
```

This complicated piece of code...all it does is that if the file type is python, it creates a mapping to <localleader>c which goes to the start of the line by I, types #, and then Esc back to normal mode...essentially a oneliner to create a commenting plugin. Will bring back this example when we see how useless most plugins are

To prevent repeated execution of your autocmds, you should insert it into your vimrc like this:

```
augroup filetype_html
    autocmd!
    autocmd FileType html nnoremap <buffer> <localleader>f Vatzf
augroup END
```

4.2.3 New Operators

The naming of these differs slightly from what we've used to far, but bear with me. Operators are essentially our text objects

The command :onoremap is the way to define operators Try:

```
:onoremap p i\{
:onoremap b /subsection<cr>
:onoremap in( :<c-u>normal! f(vi(<cr>
```

The first one makes an operator with key p. Try cp to delete things that correspond to inner {

The second defines b as the forward search for subsection

Try figuring out what the last does. The normal! is a way of executing normal mode commands from vimscript

If we wanted to create a way to select markdown headings, a new object for that..use:

```
:onoremap ih :<c-u>execute "normal! ?^==\\+\$\r:nohlsearch\rkvg_"<cr>
```

:normal takes a set of commands and performs action as if typed in normal mode

we use :execute for strings wihch contain <cr> and other special characters which a command like normal! would not recognize

Bit of more advanced vimscript..don't really worry too much about it. Just basically know that creating these text objects isn't very hard..just a combination of regex and motions

4.3 Plugin-less Random Suggestions

4.3.1 ctags

Install ctags on your shell

Add this to your .vimrc file:

```
command! MakeTags !ctags -R .
```

If you want support in python add this to ~/.ctags (This will get you started, but can put this in local directories as well for more nuanced support)

```
--python-kinds=-i
```

Now when you open a project directory in vim, just execute MakeTags in vim.

After that you can use:

Command	Description
ctrl]	Navigate to definition of function across files
ctrl t	go back along tag stack
g ctrl]	if ambgious tag that you need to jump to

This will save you tremendous amounts of time. Ctags support is inbuilt with vim, just may not be so with your shell, thus the installation.

4.3.2 Autocomplete

You'll find a ton of autocomplete options and plugins. Just know that vim does come with an out of the box autocomplete as well, though it is not a semantic completion engine. (Maybe use YCM or something for that?).

However don't both installing anything that is not semantic autocomplete as you could just remap the default commands to your preferred keys and have it work just out of the box.

Command	Description
ctrl x ctrl n	autocomplete within current file (do it while typing)
ctrl x ctrl f	file name autocomplete
ctrl n, ctrl p	go up or down through suggestions
ctrl x ctrl]	ctags based autocomplete across files

4.3.3 Snippets

Another cool hack is the :abbrev command, or iabbrev

```
:iabbrev waht what
```

Type something like waht and then press tab (or it may work directly sometimes...occasionally there are plugin clashes with this feature), and it'll replace it. Intuitively a collection of these could easily be used to create code snippets without any plugins

Another way to go about creating snippets is something like:

```
nnoremap <Leader>html :-1read $HOME/.vim_templates/html_template<CR>3jwf>a
```

All this does is when I type my leader, followed by html in rapid succession, it'll go and read from that file linked, after which it will go 3 lines down, one word over, go til the end of the angular bracket, and go to insert mode at that location.

It's that simple. Add any file and manage your own snippets easily like this. Create some kind of directory of your own templates, and you have full configuration capabilities on top of this. Easy huh?

4.3.4 File Exploring

Vim's native file browser can be invoked by

```
:Ex
:Sex
```

The second one splits the screen and opens the file explorer while the first opens up in the current buffer.

Plugins like nerdtree and others are quite useless given that they are essentially wrappers around the default explorer, and just a few minor settings can get it to look almost the same (go look online for the config).

Here's a clean config getting rid of some banners and all:

```
let g:netrew_banner=0
let g:netrew_browse_split=4
let g:netrew_altv=1
let g:netrew_lifestyle=3
let g:netrew_list_hide=netrw_gitignore#Hide()
let g:netrew_list_hide=',\(^|\s\s\)\zs\.\S\+'
```

Files can be really easily opened in vim. If you have some valid path written ex. /test, try putting your cursor above it and typing gf.

Remember buffers and :ls to get yourself back out.

4.3.5 Plugins

Try to reduce your use of plugins as much as possible. Hopefully throughout this document you've seen that it's quite trivial to create many of the "popular" plugins that are used.

Still, there are many plugins which do add quite a bit of value so don't discard all of them.

Check out:

- repeat.vim(`https://github.com/tpope/vim-repeat`) which adds the repeat command to many larger commands.
- surround vim `https://github.com/tpope/vim-surround`. This lets you do stuff like delete surrounding quotes with ds", change type of quotes with cs"', add single quotes around a particular word with ysiw", change surrounding tag in html with csit...etc.

- Some commenting plugins are nice, look for the most lightweight ones. Native way to comment would be to visual block, I, comment, Esc.
- look for good plugin managers as well like Vundle, Pathogen..etc.

Plugins, inevitably, people will figure out. So won't be adding too much to this.

4.3.6 Random

Find and Replace over the whole file

```
%s/target/new/
```

You can use any visual mode to select some text and then execute find and replace only within that by

```
s/target/new/
```

This will replace the first occurence of the target on each line. There's more nuanced ways which you can look up, but if you want to replace all occurences just change it to

```
s/target/new/g
```

As seen in the snippet section, you can read directly from another file into your current file. Try:

```
:read filename
```

I very commonly like to see what I'm selecting, so often before I select some lines, I'll go into visual line mode (V), then I use motions like 3j, 5k or something to select the lines I want to, and then use d or something to delete or apply some other operation.

Can find pressing Esc to be quite tiresome so I suggest a remap to jk, an uncommon set of keys typed, essentially

```
inoremap jk <c-c>`^
inoremap kj <c-c>`^
```

5 Thanks

Thanks for reading! Anyway this concludes the document. Hope you learned something useful :)

Will have this up on github, so please feel free to contribute and add more details. Aim of this was to keep it simple but explore the power of vim.

6 Acknowledgements

Lot of the material here is a result of going through a bunch of blogs, youtube videos, and even some books. Will try to credit as many as I can, and update this list as much as possible.

- Learn Vimscript the Hard Way
- `https://medium.com/actualize-network/how-to-learn-vim-a-four-week-plan-cd8b3`
- `https://www.youtube.com/watch?time_continue=1&v=wlR5gYd6um0&feature=emb_logo`
- `https://vim.fandom.com/wiki/Vim_Tips_Wiki`