What is your most productive shortcut with Vim?

Asked 12 years, 1 month ago Active 4 years, 1 month ago Viewed 858k times

1125 votes

5400

Locked. This question and its answers are locked because the question is off-topic but has historical significance. It is not currently accepting new answers or interactions.

I've heard a lot about Vim, both pros and cons. It really seems you should be (as a developer) faster with Vim than with any other editor. I'm using Vim to do some basic stuff and I'm at best 10 times *less productive* with Vim.

The only two things you should care about when you talk about speed (you may not care enough about them, but you should) are:

1. Using alternatively left and right hands is the **fastest** way to use the keyboard.
2. Never touching the mouse is the second way to be as fast as possible. It takes ages for you to move your hand, grab the mouse, move it, and bring it back to the keyboard (and you often have to look at the keyboard to be sure you returned your hand properly to the right place)

Here are two examples demonstrating why I'm far less productive with Vim.

Copy/Cut & paste. I do it all the time. With all the contemporary editors you press Shift with the left hand, and you move the cursor with your right hand to select text. Then Ctrl+C copies, you move the cursor and Ctrl+V pastes.

With Vim it's horrible:

- `yy` to copy one line (you almost never want the whole line!)
- `[number xx]yy` to copy `xx` lines into the buffer. But you never know exactly if you've selected what you wanted. I often have to do `[number xx]dd` then `u` to undo!

Another example? **Search & replace.**

- In PSPad: Ctrl+f then type what you want you search for, then press Enter.
- In Vim: `/`, then type what you want to search for, then if there are some special characters put `\` before *each* special character, then press Enter.

And everything with Vim is like that: it seems I don't know how to handle it the right way.

NB : **I've already read the Vim cheat sheet** :)

My question is:

What is the way you use Vim that makes you more productive than with a contemporary editor?

vim vi

Share edited Aug 16 '17 at 10:58 community wiki
12 revs, 9 users 36%
Olivier Pons

Comments disabled on deleted / locked posts / reviews |

50 Answers

Active Oldest Votes

1 2 Next

0 votes

`nnoremap q; q:` in my .vimrc, keeps the flow when crafting a complicated search&replace.

Share edited Jan 8 '13 at 20:20 community wiki
2 revs
sjas

48 votes

gi

Go to last edited location (very useful if you performed some searching and than want go back to edit)

^P and ^N

Complete previous (^P) or next (^N) text.

^O and ^I

Go to previous (`^O - "O"` for old) location or to the next (`^I - "I"` just near to `"O"`). When you perform searches, edit files etc., you can navigate through these "jumps" forward and back.

Share edited Dec 24 '12 at 14:50 community wiki
3 revs, 3 users 87%
dimba

4 @Kungi `. will take you to the last edit `` will take you back to the position you were in before the last 'jump' - which /might/ also be the position of the last edit. – Grant McLean Aug 23 '11 at 8:21

207
votes

Some productivity tips:

Smart movements

- `*` and `#` search for the word under the cursor forward/backward.
- `w` to the next word
- `W` to the next space-separated word
- `b` / `e` to the begin/end of the current word. (`B` / `E` for space separated only)
- `gg` / `G` jump to the begin/end of the file.
- `%` jump to the matching { .. } or (..), etc..
- `{` / `}` jump to next paragraph.
- `'.` jump back to last edited line.
- `g;` jump back to last edited position.

Quick editing commands

- `I` insert at the begin.
- `A` append to end.
- `o` / `O` open a new line after/before the current.
- `v` / `V` / `Ctrl+V` visual mode (to select text!)
- `Shift+R` replace text
- `C` change remaining part of line.

Combining commands

Most commands accept a amount and direction, for example:

- `cW` = change till end of word
- `3cW` = change 3 words
- `BcW` = to begin of full word, change full word
- `ciW` = change inner word.
- `ci"` = change inner between ".."
- `ci(` = change text between (..)
- `ci<` = change text between < .. > (needs `set matchpairs+=<:>` in vimrc)

- `4dd` = delete 4 lines
- `3x` = delete 3 characters.
- `3s` = substitute 3 characters.

Useful programmer commands

- `r` replace one character (e.g. `rd` replaces the current char with `d`).
- `~` changes case.
- `J` joins two lines
- Ctrl+A / Ctrl+X increments/decrements a number.
- `.` repeat last command (a simple macro)
- `==` fix line indent
- `>` indent block (in visual mode)
- `<` unindent block (in visual mode)

Macro recording

- Press `q[ key ]` to start recording.
- Then hit `q` to stop recording.
- The macro can be played with `@[ key ]`.

By using very specific commands and movements, VIM can replay those exact actions for the next lines. (e.g. A for append-to-end, `b` / `e` to move the cursor to the begin or end of a word respectively)

Example of well built settings

```
# reset to vim-defaults
if &compatible          # only if not set before:
  set nocompatible      # use vim-defaults instead of vi-defaults (easier,
more user friendly)
endif

# display settings
set background=dark     # enable for dark terminals
set nowrap              # dont wrap lines
set scrolloff=2         # 2 lines above/below cursor when scrolling
set number              # show line numbers
set showmatch           # show matching bracket (briefly jump)
set showmode            # show mode in status bar (insert/replace/...)
set showcmd             # show typed command in status bar
set ruler               # show cursor position in status bar
set title               # show file in titlebar
set wildmenu            # completion with menu
set wildignore=*.o,*.obj,*.bak,*.exe,*.py[co],*.swp,*~,*.pyc,.svn
set laststatus=2        # use 2 lines for the status bar
set matchtime=2         # show matching bracket for 0.2 seconds
set matchpairs+=<:>     # specially for html

# editor settings
set esckeys             # map missed escape sequences (enables keypad keys)
set ignorecase          # case insensitive searching
set smartcase           # but become case sensitive if you type uppercase
characters
set smartindent         # smart auto indenting
set smarttab            # smart tab handling for indenting
set magic               # change the way backslashes are used in search
patterns
set bs=indent,eol,start # Allow backspacing over everything in insert mode

set tabstop=4           # number of spaces a tab counts for
set shiftwidth=4        # spaces for autoindents
#set expandtab           # turn a tabs into spaces

set fileformat=unix     # file mode is unix
#set fileformats=unix,dos    # only detect unix file format, displays that ^M
```

The settings can be stored in `~/.vimrc`, or system-wide in `/etc/vimrc.local` and then by read from the `/etc/vimrc` file using:

```
source /etc/vimrc.local
```

(you'll have to replace the `#` comment character with `"` to make it work in VIM, I wanted to give proper syntax highlighting here).

The commands I've listed here are pretty basic, and the main ones I use so far. They already make me quite more productive, without having to know all the fancy stuff.

Share

edited Oct 29 '12 at 18:51

community wiki
12 revs, 3 users 99%
vdboor

4

Better than `'.` is `g;`, which jumps back through the `changelist`. Goes to the last edited position, instead of last edited line – naught101 Apr 28 '12 at 2:09

32
votes

Last week at work our project inherited a lot of Python code from another project. Unfortunately the code did not fit into our existing architecture - it was all done with global variables and functions, which would not work in a multi-threaded environment.

We had ~80 files that needed to be reworked to be object oriented - all the functions moved into classes, parameters changed, import statements added, etc. We had a list of about 20 types of fix that needed to be done to each file. I would estimate that doing it by hand one person could do maybe 2-4 per day.

So I did the first one by hand and then wrote a vim script to automate the changes. Most of it was a list of vim commands e.g.

```
" delete an un-needed function "
g/someFunction(/ d

" add wibble parameter to function foo "
%s/foo(/foo( wibble,/

" convert all function calls bar(thing) into method calls thing.bar() "
g/bar(/ normal nmaf(ldi(`aPa.
```

The last one deserves a bit of explanation:

```
g/bar(/  executes the following command on every line that contains "bar("
normal   execute the following text as if it was typed in in normal mode
n        goes to the next match of "bar(" (since the :g command leaves the
cursor position at the start of the line)
ma       saves the cursor position in mark a
f(       moves forward to the next opening bracket
l        moves right one character, so the cursor is now inside the brackets
di(      delete all the text inside the brackets
`a       go back to the position saved as mark a (i.e. the first character of
"bar")
P        paste the deleted text before the current cursor position
a.       go into insert mode and add a "."
```

For a couple of more complex transformations such as generating all the import statements I embedded some python into the vim script.

After a few hours of working on it I had a script that will do at least 95% of the conversion. I just open a file in vim then run `:source fixit.vim` and the file is transformed in a blink of the eye.

We still have the work of changing the remaining 5% that was not worth automating and of testing the results, but by spending a day writing this script I estimate we have saved weeks of work.

Of course it would have been possible to automate this with a scripting language like Python or Ruby, but it would have taken far longer to write and would be less flexible - the last example would have been difficult since regex alone would not be able to handle nested brackets, e.g. to convert `bar(foo(xxx))` to `foo(xxx).bar()`. Vim was perfect for the task.

Share | edited Aug 18 '12 at 21:44 | community wiki
8 revs, 2 users 98%
Dave Kirby

1 Thanks a lot for sharing that's really nice to learn from "useful & not classical" macros. – Olivier Pons Feb 28 '10 at 14:41

1 Of if you don't like vim-style escapes, use \v to turn on Very Magic: `%s/\v(bar)\((.+)\)/\2.\1()/` – Ipsquiggle Mar 23 '10 at 16:56

2 votes

Quick Cut and Overwrite portion of a line:

A very common task when you are editing a line is to cut from the current cursor location till a certain place and overwrite the new content.

You can use the following commands:

`ct<identifier>` for forward cutting.

`cT<identifier>` for backward cutting.

Where is the character in the line till which you want to cut.

Example: Lets say this the line you want to edit and your cursor is at `I`.

```
Hi There. I am a Coder and I code in : Python and R.
```

You want to cut till `:` and overwrite with `I am a programmer`, you type: `ct:` then type `I am a programmer`. This will result in: `Hi There. I am a programmer: Python and R`.

Quick Delete portion of a line:

Just like above the following commands delete the content from the current cursor location till the 'identifier'

`dt<identifier>` for forward delete

`dT<identifier>` for backward delete

Hope this is useful to you too.

Share   edited Jul 20 '12 at 17:47   community wiki
2 revs
Aby

1
vote

clever tab completion ^^

http://vim.wikia.com/wiki/Smart_mapping_for_tab_completion

Share   answered Jul 18 '12 at 7:47   community wiki
David Lam

9
votes

After mapping the below to a simple key combo, the following are very useful for me:

Jump into a file while over its path

```
gf
```

get full path name of existing file

```
:r!echo %:p
```

get directory of existing file

```
:r!echo %:p:h
```

run code:

```
:!ruby %:p
```

ruby abbreviations:

```
ab if_do if end<esc>bi<cr><esc>xhxO
ab if_else if end<esc>bi<cr><esc>xhxO else<esc>bhxA<cr> <esc>k$O
ab meth def method<cr>end<esc>k<esc>:s/method/
ab klas class KlassName<cr>end<esc>k<esc>:s/KlassName/
ab mod module ModName<cr>end<esc>k<esc>:s/ModName/
```

run current program:

```
   map ,rby :w!<cr>:!ruby %:p<cr>
```

check syntax of current program:

```
    map ,c :w!<cr>:!ruby -c %:p<cr>
```

run all specs for current spec program:

```
    map ,s :w!<cr>:!rspec %:p<cr>
```

crack it open irb:

```
    map ,i :w!<cr>:!irb<cr>
```

rspec abreviations:

```
    ab shared_examples shared_examples_for "behavior here" do<cr>end
    ab shared_behavior describe "description here" do<cr>  before :each
 do<cr>end<cr>it_should_behave_like "behavior here"<cr><bs>end<cr>
    ab describe_do describe "description here" do<cr>end
    ab context_do describe "description here" do<cr>end
    ab it_do it "description here" do<cr>end
    ab before_each before :each do<cr>end<cr>
```

rails abbreviations:

user authentication:

```
      ab userc <esc>:r $VIMRUNTIME/Templates/Ruby/c-users.rb<cr>
      ab userv <esc>:r $VIMRUNTIME/Templates/Ruby/v-users.erb<cr>
      ab userm <esc>:r $VIMRUNTIME/Templates/Ruby/m-users.rb<cr>
```

open visually selected url in firefox:

```
 "function
    function open_url_in_firefox:(copy_text)
      let g:open_url_in_firefox="silent !open -a \"firefox\"
 \"".a:copy_text."\""
      exe g:open_url_in_firefox
    endfunction

 "abbreviations
    map ,d :call open_url_in_firefox:(expand("%:p"))<cr>
    map go y:call open_url_in_firefox:(@0)<cr>
```

rspec: run spec containing current line:

```
"function
   function run_single_rspec_test:(the_test)
     let g:rake_spec="!rspec ".a:the_test.":".line(".")
     exe g:rake_spec
   endfunction

"abbreviations
   map ,s :call run_single_rspec_test:(expand("%:p"))<cr>
```

rspec-rails: run spec containing current line:

```
"function
   function run_single_rails_rspec_test:(the_test)
     let g:rake_spec="!rake spec SPEC=\"".a:the_test.":".line(".")."\""
     exe g:rake_spec
   endfunction

"abbreviations
   map ,r :call run_single_rails_rspec_test:(expand("%:p"))<cr>
```

rspec-rails: run spec containing current line with debugging:

```
"function
   function run_spec_containing_current_line_with_debugging:(the_test)
     let g:rake_spec="!rake spec SPEC=\"".a:the_test.":".line(".")." -d\""
     exe g:rake_spec
   endfunction

"abbreviations
   map ,p :call run_spec_containing_current_line_with_debugging:(expand("%:p"))
<cr>
```

html

```
"abbreviations

  "ab htm <html><cr><tab><head><cr></head><cr><body><cr></body><cr><bs><bs>
</html>
   ab template_html <script type = 'text/template' id = 'templateIdHere'>
</script>
   ab script_i <script src=''></script>
   ab script_m <script><cr></script>
   ab Tpage <esc>:r ~/.vim/templates/pageContainer.html<cr>
   ab Ttable <esc>:r ~/.vim/templates/listTable.html<cr>

"function to render common html template

   function html:()
     call feedkeys( "i", 't' )
     call feedkeys("<html>\<cr>  <head>\<cr></head>\<cr><body>\<cr> ", 't')
     call feedkeys( "\<esc>", 't' )
     call feedkeys( "i", 't' )
     call include_js:()
     call feedkeys("\<bs>\<bs></body>\<cr> \<esc>hxhxi</html>", 't')
   endfunction
```

javascript

```
"jasmine.js
  "abbreviations
   ab describe_js describe('description here', function(){<cr>});
   ab context_js context('context here', function(){<cr>});
   ab it_js it('expectation here', function(){<cr>});
   ab expect_js expect().toEqual();
   ab before_js beforeEach(function(){<cr>});
   ab after_js afterEach(function(){<cr>});

"function abbreviations

   ab fun1 function(){}<esc>i<cr><esc>ko
   ab fun2 x=function(){};<esc>hi<cr>
   ab fun3 var x=function(){<cr>};

"method for rendering inclusion of common js files

   function include_js:()
     let includes_0  = "  <link   type = 'text\/css' rel = 'stylesheet' href
= '\/Users\/johnjimenez\/common\/stylesheets\/jasmine-1.1.0\/jasmine.css'\/>"
     let includes_1  = "  <link   type = 'text\/css' rel = 'stylesheet' href
= '\/Users\/johnjimenez\/common\/stylesheets\/screen.css'\/>"
     let includes_2  = "<script type = 'text\/javascript' src =
'\/Users\/johnjimenez\/common\/javascripts\/jquery-1.7.2\/jquery-1.7.2.js'>
<\/script>"
     let includes_3  = "<script type = 'text\/javascript' src =
'\/Users\/johnjimenez\/common\/javascripts\/underscore\/underscore.js'>
<\/script>"
     let includes_4  = "<script type = 'text\/javascript' src =
'\/Users\/johnjimenez\/common\/javascripts\/backbone-0.9.2\/backbone.js'>
<\/script>"
     let includes_5  = "<script type = 'text\/javascript' src =
'\/Users\/johnjimenez\/common\/javascripts\/jasmine-1.1.0\/jasmine.js'>
<\/script>"
     let includes_6  = "<script type = 'text\/javascript' src =
'\/Users\/johnjimenez\/common\/javascripts\/jasmine-1.1.0\/jasmine-html.js'>
<\/script>"
     let includes_7  = "<script>"
     let includes_8  = "  describe('default page', function(){ "
```

Share

edited Jul 16 '12 at 17:32

community wiki
4 revs
kikuchiyo

1 Ah, and if you want to practice basic movement commands in a game environment, I started this game while I was in-between jobs in December: kikuchiyo.org – kikuchiyo Mar 24 '12 at 3:34

5
votes

How about further shortcutting shortcuts?

Put into your .vimrc:

nnoremap ; :

This way entering commmandmode ist way easier: `;q` or `;w` work, instead just `:q` or `:w`.

Two keystrokes instead of three, and you will need this very often.

Bad for sysadmins, since they need the same functionality to be given out of the box to be the same on every box everywhere.

But a ***HUGE** improvement for programmers* using vi.

Share

edited Jul 10 '12 at 13:06

community wiki
2 revs
sjas

3
votes

The series of vim commands `ggVGg?` applies a Rot13 cipher to the text in your current document.

Gung vf zl zbfg cebqhpgvir fubegphg fvapr V nyjnlf glcr va Ebg13.

Share

answered Jul 7 '12 at 2:15

community wiki
cytinus

1 That is my most productive shortcut since I always type in Rot13. ;) Well, actually ROT-13 is just `g?`, the rest is just selection shortcutting. This is just for those who dont know, not a critique of the post! :D – sjas Jul 10 '12 at 13:11

1 Ununun Avpr. Did you use the vim shortcut? – cytinus Jul 10 '12 at 13:15

4
votes

You can search the content of a register.

Suppose that your register `x` contains

```
string to search
```

To search this string, you have to type in normal mode
`/` `CTRL-r` `x` `ENTER`

It'll paste the content of `x` register.

Share

answered Jun 28 '12 at 14:16

community wiki
Luc M

16 votes

Odd nobody's mentioned ctags. Download "exuberant ctags" and put it ahead of the crappy preinstalled version you already have in your search path. Cd to the root of whatever you're working on; for example the Android kernel distribution. Type "ctags -R ." to build an index of source files anywhere beneath that dir in a file named "tags". This contains all tags, nomatter the language nor where in the dir, in one file, so cross-language work is easy.

Then open vim in that folder and read :help ctags for some commands. A few I use often:

- Put cursor on a method call and type CTRL-] to go to the method definition.
- Type :ta name to go to the definition of name.

Share

edited May 6 '12 at 20:50

community wiki
3 revs
Bradjcox

9 votes

Here is another **site** that I found helpful in learning Vim. It's fun too! :)

> **VIM Adventures** is an online game based on VIM's keyboard shortcuts (commands, motions and operators). It's the "Zelda meets text editing" game. It's a puzzle game for practicing and memorizing VIM commands (good old VI is also covered, of course). It's an easy way to learn VIM without a steep learning curve.

Share

edited Apr 25 '12 at 22:22

community wiki
2 revs
Alex Lockwood

1 Very nice site, a bit slow under my (big configuration) Linux, but nice ;) – Olivier Pons Apr 25 '12 at 5:36

98 votes

The `Control` + `R` mechanism is very useful :-) In either **insert mode** or **command mode** (i.e. on the `:` line when typing commands), continue with a numbered or named register:

- `a` - `z` the named registers
- `"` the unnamed register, containing the text of the last delete or yank
- `%` the current file name

- `#` the alternate file name
- `*` the clipboard contents (X11: primary selection)
- `+` the clipboard contents
- `/` the last search pattern
- `:` the last command-line
- `.` the last inserted text
- `-` the last small (less than a line) delete
- `=5*5` insert 25 into text (mini-calculator)

See `:help i_CTRL-R` and `:help c_CTRL-R` for more details, and snoop around nearby for more CTRL-R goodness.

Share edited Apr 12 '12 at 7:46 community wiki 5 revs, 4 users 53% kev

10 FYI, this refers to Ctrl+R in *insert mode*. In normal mode, Ctrl+R is redo. – vdboor Jun 3 '10 at 9:08

2 +1 for current/alternate file name. `Control-A` also works in insert mode for last inserted text, and `Control-@` to both insert last inserted text and immediately switch to normal mode. – Aryeh Leib Taurog Feb 26 '12 at 19:06

---

23 votes

I recently discovered `q:`. It opens the "command window" and shows your most recent ex-mode (command-mode) commands. You can move as usual within the window, and pressing `<CR>` executes the command. You can edit, etc. too. Priceless when you're messing around with some complex command or regex and you don't want to retype the whole thing, or if the complex thing you want to do was 3 commands back. It's almost like bash's `set -o vi`, but for vim itself (heh!).

See `:help q:` for more interesting bits for going back and forth.

Share answered Apr 2 '12 at 7:56 community wiki David Pope

2858 votes

Your problem with Vim is that you don't grok vi.

You mention cutting with `yy` and complain that you almost never want to cut whole lines. In fact programmers, editing source code, very often want to work on whole lines, ranges of lines and blocks of code. However, `yy` is only one of many way to yank text into the anonymous copy buffer (or "register" as it's called in **vi**).

The "Zen" of **vi** is that you're speaking a language. The initial `y` is a verb. The statement `yy` is a synonym for `y_`. The `y` is doubled up to make it easier to type, since it is such a common operation.

This can also be expressed as `dd P` (delete the current line and paste a copy back into place; leaving a copy in the anonymous register as a side effect). The `y` and `d` "verbs" take any movement as their "subject." Thus `yW` is "yank from here (the cursor) to the end of the current/next (big) word" and `y'a` is "yank from here to the line containing the mark named '*a*'."

If you only understand basic up, down, left, and right cursor movements then **vi** will be no more productive than a copy of "notepad" for you. (Okay, you'll still have syntax highlighting and the ability to handle files larger than a piddling ~45KB or so; but work with me here).

vi has 26 "marks" and 26 "registers." A mark is set to any cursor location using the `m` command. Each mark is designated by a single lower case letter. Thus `ma` sets the '*a*' mark to the current location, and `mz` sets the '*z*' mark. You can move to the line containing a mark using the `'` (single quote) command. Thus `'a` moves to the beginning of the line containing the '*a*' mark. You can move to the precise location of any mark using the `` ` `` (backquote) command. Thus `` `z `` will move directly to the exact location of the '*z*' mark.

Because these are "movements" they can also be used as subjects for other "statements."

So, one way to cut an arbitrary selection of text would be to drop a mark (I usually use '*a*' as my "first" mark, '*z*' as my next mark, '*b*' as another, and '*e*' as yet another (I don't recall ever having interactively used more than four marks in 15 years of using **vi**; one creates one's own conventions regarding how marks and registers are used by macros that don't disturb one's interactive context). Then we go to the other end of our desired text; we can start at either end, it doesn't matter. Then we can simply use `` d`a `` to cut or `` y`a `` to copy. Thus the whole process has a 5 keystrokes overhead (six if we started in "insert" mode and needed to `Esc` out command mode). Once we've cut or copied then pasting in a copy is a single keystroke: `p`.

I say that this is one way to cut or copy text. However, it is only one of many. Frequently we can more succinctly describe the range of text without moving our cursor around and dropping a mark. For example if I'm in a paragraph of text I can use `{` and `}` movements to the beginning or end of the paragraph respectively. So, to move a paragraph of text I cut it using `{ d}` (3 keystrokes). (If I happen to already be on the first or last line of the paragraph I can then simply use `d}` or `d{` respectively.

The notion of "paragraph" defaults to something which is usually intuitively reasonable. Thus it often works for code as well as prose.

Frequently we know some pattern (regular expression) that marks one end or the other of the text in which we're interested. Searching forwards or backwards are movements in **vi**. Thus they can also be used as "subjects" in our "statements." So I can use `d/foo` to cut from the current line to the next line containing the string "foo" and `y?bar` to copy from the current line to the most recent (previous) line containing "bar." If I don't want whole lines I can still use the search movements (as statements of their own), drop my mark(s) and use the `` `x `` commands as described previously.

In addition to "verbs" and "subjects" **vi** also has "objects" (in the grammatical sense of the term). So far I've only described the use of the anonymous register. However, I can use any of the 26 "named" registers by *prefixing* the "object" reference with `"` (the double quote modifier). Thus if I use `"add` I'm cutting the current line into the '*a*' register and if I use `"by/foo` then I'm yanking a copy of the text from here to the next line containing "foo" into the '*b*' register. To paste from a register I simply prefix the paste with the same modifier sequence: `"ap` pastes a copy of the '*a*' register's contents into the text after the cursor and `"bP` pastes a copy from '*b*' to before the current line.

This notion of "prefixes" also adds the analogs of grammatical "adjectives" and "adverbs' to our text manipulation "language." Most commands (verbs) and movement (verbs or objects, depending on context) can also take numeric prefixes. Thus `3J` means "join the next three lines" and `d5}` means "delete from the current line through the end of the fifth paragraph down from here."

This is all intermediate level **vi**. None of it is **Vim** specific and there are far more advanced tricks in **vi** if you're ready to learn them. If you were to master just these intermediate concepts then you'd probably find that you rarely need to write any macros because the text manipulation language is sufficiently concise and expressive to do most things easily enough using the editor's "native" language.

A sampling of more advanced tricks:

There are a number of `:` commands, most notably the `:% s/foo/bar/g` global substitution technique. (That's not advanced but other `:` commands can be). The whole `:` set of commands was historically inherited by **vi**'s previous incarnations as the **ed** (line editor) and later the **ex** (extended line editor) utilities. In fact **vi** is so named because it's the visual interface to **ex**.

`:` commands normally operate over lines of text. **ed** and **ex** were written in an era when terminal screens were uncommon and many terminals were "teletype" (TTY) devices. So it was common to work from printed copies of the text, using commands through an extremely terse interface (common connection speeds were 110 baud, or, roughly, 11 characters per second -- which is slower than a fast typist; lags were common on multi-user interactive sessions; additionally there was often some motivation to conserve paper).

So the syntax of most `:` commands includes an address or range of addresses (line number) followed by a command. Naturally one could use literal line numbers: `:127,215 s/foo/bar` to change the first occurrence of "foo" into "bar" on each line between 127 and 215. One could also use some abbreviations such as `.` or `$` for current and last lines respectively. One could also use relative prefixes `+` and `-` to refer to offsets after or before the curent line, respectively. Thus: `:.,$j` meaning "from the current line to the last line, join them all into one line". `:%` is synonymous with `:1,$` (all the lines).

The `:... g` and `:... v` commands bear some explanation as they are incredibly powerful. `:... g` is a prefix for "globally" applying a subsequent command to all lines which match a pattern (regular expression) while `:... v` applies such a command to all lines which do NOT match the given pattern ("v" from "conVerse"). As with other **ex** commands these can be prefixed by addressing/range references. Thus `:.,+21g/foo/d` means "delete any lines containing the string "foo" from the current one through the next 21 lines" while `:.,$v/bar/d` means "from here to the end of the file, delete any lines which DON'T contain the string "bar."

It's interesting that the common Unix command **grep** was actually inspired by this **ex** command (and is named after the way in which it was documented). The **ex** command `:g/re/p` (grep) was the way they documented how to "globally" "print" lines containing a "regular expression" (re). When **ed** and **ex** were used, the `:p` command was one of the first that anyone learned and often the first one used when editing any file. It was how you printed the current contents (usually just one page full at a time using `:.,+25p` or some such).

Note that `:% g/.../d` or (its reVerse/conVerse counterpart: `:% v/.../d` are the most common usage patterns. However there are couple of other `ex` commands which are worth remembering:

We can use `m` to move lines around, and `j` to join lines. For example if you have a list and you want to separate all the stuff matching (or conversely NOT matching some pattern) without deleting them, then you can use something like: `:% g/foo/m$` ... and all the "foo" lines will have been moved to the end of the file. (Note the other tip about using the end of your file as a scratch space). This will have preserved the relative order of all the "foo" lines while having extracted them from the rest of the list. (This would be equivalent to doing something like: `1G!GGmap!Ggrep foo<ENTER>1G:1,'a g/foo'/d` (copy the file to its own tail, filter the tail through `grep`, and delete all the stuff from the head).

To join lines usually I can find a pattern for all the lines which need to be joined to their predecessor (all the lines which start with "^ " rather than "^ * " in some bullet list, for example). For that case I'd use: `:% g/^    /-1j` (for every matching line, go up one line and join them). (BTW: for bullet lists trying to search for the bullet lines and join to the next doesn't work for a couple reasons ... it can join one bullet line to another, and it won't join any bullet line to *all* of its continuations; it'll only work pairwise on the matches).

Almost needless to mention you can use our old friend `s` (substitute) with the `g` and `v` (global/converse-global) commands. Usually you don't need to do so. However, consider some case where you want to perform a substitution only on lines matching some other pattern. Often you can use a complicated pattern with captures and use back references to preserve the portions of the lines that you DON'T want to change. However, it will often be easier to separate the match from the substitution: `:% g/foo/s/bar/zzz/g` -- for every line containing "foo" substitute all "bar" with "zzz." (Something like `:% s/\(.*foo.*\)bar\(.*\)/\1zzz\2/g` would only work for the cases those instances of "bar" which were PRECEDED by "foo" on the same line; it's ungainly enough already, and would have to be mangled further to catch all the cases where "bar" preceded "foo")

The point is that there are more than just `p`, `s`, and `d` lines in the `ex` command set.

The `:` addresses can also refer to marks. Thus you can use: `:'a,'bg/foo/j` to join any line containing the string foo to its subsequent line, if it lies between the lines between the '*a*' and '*b*' marks. (Yes, all of the preceding `ex` command examples can be limited to subsets of the file's lines by prefixing with these sorts of addressing expressions).

That's pretty obscure (I've only used something like that a few times in the last 15 years). However, I'll freely admit that I've often done things iteratively and interactively that could probably have been done more efficiently if I'd taken the time to think out the correct incantation.

Another very useful **vi** or **ex** command is `:r` to read in the contents of another file. Thus: `:r foo` inserts the contents of the file named "foo" at the current line.

More powerful is the `:r!` command. This reads the results of a command. It's the same as suspending the **vi** session, running a command, redirecting its output to a temporary file, resuming your **vi** session, and reading in the contents from the temp. file.

Even more powerful are the `!` (bang) and `:... !` (**ex** bang) commands. These also execute external commands and read the results into the current text. However, they also filter selections of our text through the command! This we can sort all the lines in our file using `1G!Gsort` (`G` is the **vi** "goto" command; it defaults to going to the last line of the file, but can be prefixed by a line number, such as 1, the first line). This is equivalent to the **ex** variant `:1,$!sort`. Writers often use `!` with the Unix **fmt** or **fold** utilities for reformating or "word wrapping" selections of text. A very common macro is

`{!}fmt` (reformat the current paragraph). Programmers sometimes use it to run their code, or just portions of it, through **indent** or other code reformatting tools.

Using the `:r!` and `!` commands means that any external utility or filter can be treated as an extension of our editor. I have occasionally used these with scripts that pulled data from a database, or with **wget** or **lynx** commands that pulled data off a website, or **ssh** commands that pulled data from remote systems.

Another useful **ex** command is `:so` (short for `:source`). This reads the contents of a file as a series of commands. When you start **vi** it normally, implicitly, performs a `:source` on `~/.exinitrc` file (and **Vim** usually does this on `~/.vimrc`, naturally enough). The use of this is that you can change your editor profile on the fly by simply sourcing in a new set of macros, abbreviations, and editor settings. If you're sneaky you can even use this as a trick for storing sequences of **ex** editing commands to apply to files on demand.

For example I have a seven line file (36 characters) which runs a file through **wc**, and inserts a C-style comment at the top of the file containing that word count data. I can apply that "macro" to a file by using a command like: `vim +'so mymacro.ex' ./mytarget`

(The `+` command line option to **vi** and **Vim** is normally used to start the editing session at a given line number. However it's a little known fact that one can follow the `+` by any valid **ex** command/expression, such as a "source" command as I've done here; for a simple example I have scripts which invoke: `vi +'/foo/d|wq!' ~/.ssh/known_hosts` to remove an entry from my SSH known hosts file non-interactively while I'm re-imaging a set of servers).

Usually it's far easier to write such "macros" using Perl, AWK, **sed** (which is, in fact, like **grep** a utility inspired by the **ed** command).

The `@` command is probably the most obscure **vi** command. In occasionally teaching advanced systems administration courses for close to a decade I've met very few people who've ever used it. `@` executes the contents of a register as if it were a **vi** or **ex** command.
Example: I often use: `:r!locate ...` to find some file on my system and read its name into my document. From there I delete any extraneous hits, leaving only the full path to the file I'm interested in. Rather than laboriously Tab-ing through each component of the path (or worse, if I happen to be stuck on a machine without Tab completion support in its copy of **vi**) I just use:

1. `0i:r` (to turn the current line into a valid **:r** command),
2. `"cdd` (to delete the line into the "c" register) and
3. `@c` execute that command.

That's only 10 keystrokes (and the expression `"cdd @c` is effectively a finger macro for me, so I can type it almost as quickly as any common six letter word).

A sobering thought

I've only scratched to surface of **vi**'s power and none of what I've described here is even part of the "improvements" for which **vim** is named! All of what I've described here should work on any old copy of **vi** from 20 or 30 years ago.

There are people who have used considerably more of **vi**'s power than I ever will.

Share

edited Jun 20 '20 at 9:12

community wiki
18 revs, 16 users 64%
Jim Dennis

440 Holy code monkeys,..that's one in-depth answer. What's great is that you probably wrote in in vim in about 10 keystrokes. – EightyOne Unite Nov 27 '09 at 15:36

48 @Wahnfieden -- grok is exactly what I meant: en.wikipedia.org/wiki/Grok (It's apparently even in the OED --- the closest we anglophones have to a canonical lexicon). To "grok" an editor is to find yourself using its commands fluently ... as if they were your natural language. – Jim Dennis Feb 12 '10 at 4:08

22 wow, a very well written answer! i couldn't agree more, although i use the `@` command a lot (in combination with `q` : record macro) – knittl Feb 27 '10 at 13:15

63 Superb answer that utterly redeems a really horrible question. I am going to upvote this question, that normally I would downvote, just so that this answer becomes easier to find. (And I'm an Emacs guy! But this way I'll have somewhere to point new folks who want a good explanation of what vi power users find fun about vi. Then I'll tell them about Emacs and they can decide.) – Brandon Rhodes Mar 29 '10 at 15:26

8 Can you make a website and put this tutorial there, so it doesn't get burried here on stackoverflow. I have yet to read better introduction to vi then this. – Marko Apr 1 '10 at 14:47

---

3 votes

There is loads of vim tricks but as of now, the one that I really enjoy is Ctrl+A as I happen to be dealing with some st**d code that hard-code array index.

Share

answered Oct 14 '11 at 12:43

community wiki
lang2

---

10 votes

You can use \= in an substitution string and is something I do every so often.

If you have what's essentially an unordered list in vim, say using # as a marker, you can convert it over to an ordered list.

```
# CSSLINT
# PHPCS
# Charlie
# Delta
```

If it starts on line one, you can do

```
:1,6s/#/\=line(".")/g
```

To convert it over to

```
1 CSSLINT
2 PHPCS
3 Charlie
4 Delta
```

If it doesn't start on line one, just do the maths:

```
:16,20s/#/\=line(".") - 15/g
```

More info at :help sub-replace-expression

Share answered Aug 27 '11 at 9:04 community wiki kguest

---

6 votes

For **Copy/Cut and Paste** specifically, using visual mode makes it much easier to adapt from other editors. So the way I normally cut and paste is:

- `Esc` - exit Insert mode (skip if you are already in Normal mode)
- `v` - turn on visual mode
- move about the file to select the text you want - visual mode will show you what characters are selected. For a few words `w`, `e` and `b` are useful (move to start of next word, end of next word and start of this/previous word respectively).
- `d` - cut the text (use `y` if you want to copy the text)
- move about to where you want the text to go
- `p` - paste (this pastes after the current character, `P` pastes before the current character.

Also useful is using `V` to go into Visual mode (line) which selects whole lines automatically, where ever the cursor is on the line.

Share

edited Jun 15 '11 at 13:53

community wiki
2 revs, 2 users 72%
Hamish Downer

4
votes

In addition to the great reply about grokking vi, it should be noted that vim *does* add some very vi-like features that make using vi commands nicer. The one that comes to mind first are text objects: instead of `{!}fmt` to reformat the current paragraph, `!apfmt` does the same. It works by first specifying that we want to select `a` text object, which is the current *paragraph*. Similar, to change the current string literal (`foo` to `bar` for an example), instead of `T"ct"bar` (move to just after the previous `"` , *change* until just before the next `"` , insert *bar*), you can say `ci"bar` : change inside (innermost) quotes, inserting *bar*.

Thinking in terms of text objects instead of movement commands is quite nice.

Share

edited Jun 15 '11 at 13:52

community wiki
2 revs, 2 users 80%
wolverian

16
votes

I was surprised to find no one mention the `t` movement. I frequently use it with parameter lists in the form of `dt,` or `yt,`

Share

edited Jun 15 '11 at 13:44

community wiki
2 revs, 2 users 67%
David Corbin

2 or dfx, dFx, dtx, ytx, etc where x is a char, +1 – hhh Jan 14 '11 at 17:09

16
votes

Use `\c` anywhere in a search to ignore case (overriding your ignorecase or smartcase settings). E.g. `/\cfoo` or `/foo\c` will match `foo` , `Foo` , `fOO` , `FOO` , etc.

Use `\C` anywhere in a search to force case matching. E.g. `/\Cfoo` or `/foo\C` will only match foo.

Share

edited Jun 15 '11 at 13:42

community wiki
2 revs, 2 users 67%
kev

25
votes

I am a member of the American Cryptogram Association. The bimonthly magazine includes over 100 cryptograms of various sorts. Roughly 15 of these are "cryptarithms" - various types of arithmetic problems with letters substituted for the digits. Two or three of these are sudokus, except with letters instead of numbers. When the grid is completed, the nine distinct letters will spell out a word or words, on some line, diagonal, spiral, etc., somewhere in the grid.

Rather than working with pencil, or typing the problems in by hand, I download the problems from the members area of their website.

When working with these sudokus, I use vi, simply because I'm using facilities that vi has that few other editors have. Mostly in converting the lettered grid into a numbered grid, because I find it easier to solve, and then the completed numbered grid back into the lettered grid to find the solution word or words.

The problem is formatted as nine groups of nine letters, with `-` s representing the blanks, written in two lines. The first step is to format these into nine lines of nine characters each. There's nothing special about this, just inserting eight linebreaks in the appropriate places.

The result will look like this:

```
T-O-----C
-E-----S-
--AT--N-L
---NASO--
---E-T---
--SPCL---
E-T--OS--
-A-----P-
S-----C-T
```

So, first step in converting this into numbers is to make a list of the distinct letters. First, I make a copy of the block. I position the cursor at the top of the block, then type `:y}}p`. `:` puts me in command mode, `y` yanks the next movement command. Since `}` is a move to the end of the next paragraph, `y}` yanks the paragraph. `}` then moves the cursor to the end of the paragraph, and `p` pastes what we had yanked just after the cursor. So `y}}p` creates a copy of the next paragraph, and ends up with the cursor between the two copies.

Next, I to turn one of those copies into a list of distinct letters. That command is a bit more complex:

```
:!}tr -cd A-Z | sed 's/\(.\)/\1\n/g' | sort -u | tr -d '\n'
```

`:` again puts me in command mode. `!` indicates that the content of the next yank should be piped through a command line. `}` yanks the next paragraph, and the command line then uses the `tr` command to strip out everything except for upper-case letters, the `sed` command to print each letter on a single line, and the `sort` command to sort those lines, removing duplicates, and then `tr` strips out the newlines, leaving the nine distinct letters in a single line, replacing the nine lines that had made up the paragraph originally. In this case, the letters are: `ACELNOPST` .

Next step is to make another copy of the grid. And then to use the letters I've just identified to replace each of those letters with a digit from 1 to 9. That's simple: `:!}tr ACELNOPST 0-9` . The result is:

```
8-5-----1
-2-----7-
--08--4-3
---4075--
---2-8---
--7613---
2-8--57--
-0-----6-
7-----1-8
```

This can then be solved in the usual way, or entered into any sudoku solver you might prefer. The completed solution can then be converted back into letters with `:!}tr 1-9 ACELNOPST` .

There is power in vi that is matched by very few others. The biggest problem is that only a very few of the vi tutorial books, websites, help-files, etc., do more than barely touch the surface of what is possible.

Share

edited Jun 15 '11 at 13:39

community wiki
2 revs, 2 users 92%
Jeff Dege

1 Doesn't vim check the name it was started with so that it can come up in the right 'mode'? – dash-tom-bang Aug 24 '11 at 0:45

3 I'm baffled by this repeated error: you say you need `:` to go into command mode, but then invariably you specify *normal mode* commands (like `y}}p`) which cannot possibly work from the command mode?! – sehe Mar 4 '12 at 20:47

7
votes

Inserting text to some bit in code:

```
ctrl + v, (selecting text on multiple lines), I, (type something I want), ESC
```

Recording a macro to edit text and running it N times:

```
q, a (or some other letter), (do the things I want to record), q, ESC,
(type N, as in the number of times I want to run the macro), @, a
```

Share edited May 28 '11 at 9:48 community wiki
2 revs
Vesa Nieminen

1 <Esc> when recording a macro does not stop recording! It simply records an <Esc> (i.e. return to normal mode). To stop you use `q`. Also, I tend to name my macros `q` :)
– R. Martinho Fernandes Apr 1 '10 at 3:01

504 votes

You are talking about text selecting and copying, I think that you should give a look to the Vim Visual Mode.

In the visual mode, you are able to select text using Vim commands, then you can do whatever you want with the selection.

Consider the following common scenarios:

You need to select to the next matching parenthesis.

You could do:

- `v%` if the cursor is on the starting/ending parenthesis
- `vib` if the cursor is inside the parenthesis block

You want to select text between quotes:

- **vi"** for double quotes
- **vi'** for single quotes

You want to select a curly brace block (very common on C-style languages):

- `viB`
- `vi{`

You want to select the entire file:

- `ggVG`

Visual block selection is another really useful feature, it allows you to select a rectangular area of text, you just have to press Ctrl - V to start it, and then select the text block you want and perform any type of operation such as yank, delete, paste, edit, etc. It's great to edit *column oriented* text.

Share

edited Feb 8 '17 at 14:14
Community
1 1

answered Aug 2 '09 at 8:27
Christian C. Salvadó
739k 174 902 828

2 Every editor has something like this, it's not specific to vim. – finnw Aug 2 '09 at 8:49

22 Yes, but it was a specific complaint of the poster. Visual mode is Vim's best method of direct text-selection and manipulation. And since vim's buffer traversal methods are superb, I find text selection in vim fairly pleasurable. – guns Aug 2 '09 at 9:54

9 I think it is also worth mentioning Ctrl-V to select a block - ie an arbitrary rectangle of text. When you need it it's a lifesaver. – Hamish Downer Mar 16 '10 at 13:34

6 @viksit: I'm using Camtasia, but there are plenty of alternatives: codinghorror.com/blog/2006/11/screencasting-for-windows.html – Christian C. Salvadó Apr 2 '10 at 2:07

3 Also, if you've got a visual selection and want to adjust it, `o` will hop to the other end. So you can move both the beginning and the end of the selection as much as you like. – Nathan Long Mar 1 '11 at 19:05

16 votes

> What is the way you use Vim that makes you more productive than with a contemporary editor?

Being able to execute complex, repetitive edits with very few keystrokes (often using macros). Take a look at VimGolf to witness the power of Vim!

After over ten years of almost daily usage, it's hard to imagine using any other editor.

Share

answered Jan 12 '11 at 22:52

community wiki
Johnsyweb

7 votes

```
Ctrl-w Ctrl-f ............ open file under cursor in new window
Ctrl-6 ................... alternate file
'0 ....................... open last file
:x ....................... close if save
```

Share

answered Nov 4 '10 at 22:51

community wiki
SergioAraujo

19 votes

Another useful vi "shortcut" I frequently use is 'xp'. This will swap the character under the cursor with the next character.

Share answered Sep 20 '10 at 10:34 community wiki Costyn

5 Around the time that Windows xp came out, I used to joke that this is the only good use for it. – kguest Aug 27 '11 at 8:21

10 votes

cit – empty the contents of the tag with the transition in insert mode

yit – copy the contents of the tag

Share answered Sep 10 '10 at 21:27 community wiki Aleksandr Zavatskiy

3 `:help text-objects` for more of these – Joe Holloway Jan 15 '11 at 18:30

34 votes

Session

a. save session

> :mks *sessionname*

b. force save session

> :mks! *sessionname*

c. load session

> gvim or vim -S *sessionname*

Adding and Subtracting

a. Adding and Subtracting

> CTRL-A ;Add [count] to the number or alphabetic character at or after the cursor. {not in Vi
>
> CTRL-X ;Subtract [count] from the number or alphabetic character at or after the cursor. {not in Vi}

b. Window key unmapping

> In window, Ctrl-A already mapped for whole file selection you need to unmap in rc file. mark mswin.vim CTRL-A mapping part as comment or add your rc file with unmap

c. With Macro

> The CTRL-A command is very useful in a macro. Example: Use the following steps to make a numbered list.
>
> 1. Create the first list entry, make sure it starts with a number.
> 2. qa - start recording into buffer 'a'
> 3. Y - yank the entry
> 4. p - put a copy of the entry below the first one
> 5. CTRL-A - increment the number
> 6. q - stop recording
> 7. @a - repeat the yank, put and increment times

Share

answered Aug 19 '10 at 8:08

community wiki
agfe2

1 Any idea what the shortcuts are in Windows? – Don Reba Aug 22 '10 at 5:22

---

14 votes

You asked about productive shortcuts, but I think your real question is: Is vim worth it? The answer to this stackoverflow question is -> "Yes"

You must have noticed two things. Vim is powerful, and vim is hard to learn. Much of it's power lies in it's expandability and endless combination of commands. Don't feel overwhelmed. Go slow. One command, one plugin at a time. Don't overdo it.

All that investment you put into vim will pay back a thousand fold. You're going to be inside a text editor for many, many hours before you die. Vim will be your companion.

Share answered Jul 24 '10 at 5:41 community wiki autodidakto