What is your most productive shortcut with Vim? Grok Vi!!

Community Wiki and Jim Dennis

question

What is your most productive shortcut with Vim?

I've heard a lot about http://www.vim.org/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

Copy/Cut & paste. I do it all the time. With all the contemporary ors you press Shift with the left hand, and you move the cursor with 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 http://en.wikipedia.org/wiki/PSPadPSPad: 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 `\\00` 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 http://www.viemu.com/vi-vim-cheat-sheet.gifcheat http://www.fprintf.net/vimCheatSheet.htmlsheet** :)

My question is:

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

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 `'y00a` 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 `''00a` (single quote) command. Thus `''00a` moves to the beginning of the line containing the '*a*' mark. You can move to the precise location of any mark using the ``` ``00a ``` (backquote) command. Thus ``` ``00z ``` 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 `` `d00a `` to cut or `` `y00a `` 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 ``` ``00z ``` 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 foo00ENTER001G:1,00a g/foo00/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/^0  /-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 `\\\\\2/g:% s/00(.*foo.*00)bar00(.*00)/` 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: `'':00a,00bg/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 `~~00/.exinitrc` file (and **Vim** usually does this on `~~00/.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 +00so mymacro.ex00 ./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: `''/known_hostsvi +00/fmo/d20wq!0` 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.

- https://stackoverflow.blog?blb=1Blog
- https://www.facebook.com/officialstackoverflow/Facebook
- https://twitter.com/stackoverflowTwitter
- https://linkedin.com/company/stack-overflowLinkedIn

Jim Dennis