Vim: Customization-Mapping keys

This is part 7 of a series of tutorial to Vim. You can read Part 6 here.

My Vim

In the last part, we changed a lot of options, like setting up the line numbers, autoindent etc. Let's briefly consider 3 operations on the flags:

- All of these can be turned off by prepending 'no'. So, `set nonumber` will turn off line numbers. This command can be executed using `:` [1] in normal mode to turn off features temporarily.
- Another way of **toggling** an option is by appending `!`. So whatever is the present value of the flag, it will get toggled. `set cursorline!` will toggle the highlighting of cursor line.
- Similarly you can **query** status of a flag by appending `?`. If we execute `set cursorline?`, we can expect reply `cursorline` or `nocursorline` depending on the status of the flag.

Mapping

You can add or even edit the key mappings to personalize your Vim. You may want to, for example, map `<ctrl>+t` to `di<` (delete everything in a <..> region), if you work on a lot of HTML files. The way to do this would be:

```
imap <c-t> <esc>di<
```

- The way we represent control keys in VimL is little different than what is used here till now. `<ctrl>` is represented simply as `<c>`. The keys which are to be pressed simultaneously are enclosed in `<`. Thus `<c-s>` corresponds to `<ctrl>+s`.
- The `i` in `imap` stands for mapping in insert mode. Thus pressing `<c-t>` will delete everything in `<` only in insert mode. You need to use `nmap`, `imap` and `vmap` to map keys in *normal* mode, *insert* mode and, a new mode which we are going to see very soon, *visual* mode.
- So we are saying that, map `<c-t>` in insert mode to sequence of keys `<esc>`, `d`, `i`, `<`. Since we are in insert mode, we need to get to normal mode before we can `di<` and that's why we need `<esc>`.
    - Other control characters are represented in similar way: `<space>`, `<cr>`, `<enter>` [2].

Recursive Mapping (is evil)

Say you want to change the way yanking a line works, instead of yanking just one line, you want it to yank and paste two lines. One way to do this would be `nmap yy yypkkyyp` [3]: yank a line, paste, go up twice, yank previous line and paste. Try this and execute your newly mapped key `yy` from the last line, you will see a beautiful animation of your mapping duplicating everything in your buffer.

So what really happened is, since we mapped `yy` and used the same combination while defining it, the mapping got called recursively. What we wanted was, to use the default mapping for `yy` while we use it in definition of mapping. The way to do this is to use `noremap` (no recursive map). This combines with `i`, `n` and `v` to form `noreimap`, `norenmap` and `norevmap` respectively.

One last word on recursive mapping, one maybe tempted to think he/she will be careful to not use mapping recursively and get away with simple `map`. This doesn't work always, since there might be other mapping by plugins which might get called unintentionally.

- Lesson 8

> Always use non recursive mapping. (`noremap`)

Leader Key

You might have preference for certain key combinations which might be already in use by Vim or other plugin you installed. If you map that combination, which conflicts with other pre defined mapping, your mapping will simply shadow the previous binding. You may not want to have this.

The way to get around this is having a prefix key, a namespace. Remember the days when you were just starting to program, and you would name your function `my_sort` to get around the standard function `sort`, using a leader key is the exact same thing.

You can map `<leader>yy` to map to your personal way of yanking a line, or two, and leave `yy` for the default one. Leader key is a variable, you can map the leader key itself to some other key. The key `\` acts as leader key in default case. Thus, in above example, you would press `\yy` to invoke *your yank*.

You can change your leader key, say to `,` , a popular option, by executing `let mapleader = ","`

VimL

VimL or Vim Script, (both are actually the same) is a fully featured turing complete[4] language. It has support for functions, if else condition, exception handling etc. VimL is used to customize Vim as well to develop plugins which extends the features of Vim.

Details of VimL is well beyond this tutorial. But if you are interested in reading more about it and all the things we talked about in this essay and much more, please have a look at Learn Vim Script the Hard Way.

One more thing

Let's look at couple of more modes:

1. **command-line**: We've seen this mode before, but haven't talked about it formally. When we hit `:` while in normal mode, we enter command-line mode. All the stuff which we can do in `.vimrc` can be executed from command line mode. But the changes are not permanent, things will get back to what is defined in `.vimrc` next time Vim is opened.
2. **Replace mode**: Hitting `R` switches Vim to a altogether different mode, in which characters are replaced by entered characters. This can be very helpful while editing configuration files.
    - `<esc>` puts back to normal mode.
    - `r`: replace, is like switching momentarily to replace mode. It replaces *single* character under cursor with new character.
        - `r-<enter>`: Executing `r-<enter>` will over a space character will replace space with a newline, effectively breaking line into two lines.
        - `<shift>-j`: works as complementary to above action. It joins lines. It takes the line below the current line and joins it to current line, by replacing newline with a space.

Summary

Command	Comment

`imap`	Maps key combination in **insert** mode
`nmap`	Maps key combination in **normal** mode
`vmap`	Maps key combination in **visual** mode
`noremap`	Maps keys in **non recursive** way, you **always** want to have this.
`<leader>`	A prefix key to avoid shadowing already defined key combinations.
`:`	switches to command line mode. VimL statements can be executed in this mode directly
`R`	Replace Mode: replaces character under cursor with entered characters
`r`	replaces single character
`<shift>-j`	joins lines

Click here for part 8

Footnotes:

[1] Pressing `:` in normal mode, gets us into command-line mode. After execution of command, we get back to normal mode.

[2] Longer list of control characters here.

[3] this is inefficient way of yanking two lines, better way would be `ykp`

[4] In layman terms, turing complete languages are those which can do everything the most powerful programming language can do.

Date: 2017-06-8
Author: Anurag Peshne
Emacs 25.1.1 (Org mode 9.0.8)
Validate