Manual

Tips and Tricks

Pressing tells Vim you will enter a carriage return character next. Example: ^V ==

Key Mappings

The most basic form of Vim scripting is creating key mappings to do small but powerful combinations of keystrokes to accomplish a desired behaviour / task.

Mapping and Modes

Here is an overview of map commands and in which mode they work:

```
:map        " Normal, Visual and Operator-pending
:vmap       " Visual
:nmap       " Normal
:omap       " Operator-pending
:map!       " Insert and Command-line
:imap       " Insert
:cmap       " Command-line
```

Operator-pending mode is when you typed an operator character, such as “d” or “y”, and you are expected to type the motion command or a text object. Thus when you type “dw”, the “w” is entered in operator-pending mode.

Use “:map” to see currently defined mappings.

Remapping

There is a similar command for every mode:

```
:noremap     " Normal, Visual and Operator-pending
:vnoremap    " Visual
:nnoremap    " Normal
:onoremap    " Operator-pending
:noremap!    " Insert and Command-line
:inoremap    " Insert
:cnoremap    " Command-line
```

Delete a mapping

To remove a mapping use the “:unmap” command. Again, the mode the unmapping applies to depends on the command used:

```
:unmap      " Normal, Visual and Operator-pending
:vunmap     " Visual
:nunmap     " Normal
```

```
:ounmap     " Operator-pending
:unmap!     " Insert and Command-line
:iunmap     " Insert
:cunmap     " Command-line
```

To remove all mappings, use ":mapclear".

Mappings and abbreviations

Abbreviations are a lot like Insert mode mappings. The arguments are handled in the same way. The main difference is the way they are triggered. An abbreviation is triggered by typing a non-word character after the word. A mapping is triggered when typing the last character.

Mappings miscellaneous options

- The
  keyword can be used to make a mapping local to a script.
- The keyword can be used to make a mapping local to a specific buffer.
- The keyword can be used to make defining a new mapping fail when it already exists. Otherwise a new mapping simply overwrites the old one.
- To make a key do nothing, map it to .

Command-line commands

The Vim editor enables you to define your own commands. You execute these commands just like any other Command-line mode command. To define a command, use the ":command" command, as follows:

```
:command DeleteFirst 1delete
```

Now when you execute the command ":DeleteFirst" Vim executes ":1delete", which deletes the first line.

```
Note:
User-defined commands must start with a capital letter.  You cannot
use ":X", ":Next" and ":Print".  The underscore cannot be used!  You
can use digits, but this is discouraged.
```

To list the user-defined commands, execute the following command:

```
:command
```

Just like with the builtin commands, the user defined commands can be abbreviated. You need to type just enough to distinguish the command from another. Command line completion can be used to get the full name.

Arguments

Number of arguments can be specified by the *-nargs* option. The values of -nargs are as follows:

```
-nargs=0    " No arguments
-nargs=1    " One argument
-nargs=*    " Any number of arguments
-nargs=?    " Zero or one argument
-nargs=+    " One or more arguments
```

Inside the command definition, the arguments are represented by the keyword. For example:

```
:command -nargs=+ Say :echo "<args>"
```

To get special characters turned into a string, properly escaped to use as an expression, use :

```
:command -nargs=+ Say :echo <q-args>
```

The keyword contains the same information as the keyword, except in a format suitable for use as function call arguments. For example:

```
:command -nargs=* DoIt :call AFunction(<f-args>)
:DoIt a b c
```

Executes the following command:

```
:call AFunction("a", "b", "c")
```

Line range

Some commands take a range as argument, you can tell Vim that you are defining such a command you must use the *-range* option. The values for this option are as follows:

```
-range          " Range is allowed; default is the current line.
-range=%        " Range is allowed; default is the whole file.
-range=`count`  " Range is allowed; the last number in it is used as a
                " single number whose default is `count`.
```

Other options

Some of the other options and keywords are as follows:

```
-count=`number`     " The command can take a count whose default is
                    " `number`.  The resulting count can be used
                    " through the <count> keyword.
-bang               " You can use a !.  If present, using <bang> will
                    " result in a !.
-register           " You can specify a register.  (The default is
```

```
                    " the unnamed register.)
                    " The register specification is available as
                    " <reg> (a.k.a. <register>).
-complete=`type`    " Type of command-line completion used.  See
                    " |:command-completion| for the list of possible
                    " values.
-bar                " The command can be followed by | and another
                    " command, or " and a comment.
-buffer             " The command is only available for the current
                    " buffer.
```

Finally, you have the keyword. It stands for the character <. Use this to escape the special meaning of the <> items mentioned.

Autocommands

Commands that are executed in response of some event. You can call functions or key strokes in response of this event. For example, when you save a file you can call a key stroke to align your current line to the center of the buffer (*zz* keystroke):

```
:autocmd BufWritePre *  zz
```

Learn Vimscript the Hard Way

Programmers shape ideas into text. - Steve Losh

We can print stuff into the bottom of the window in Vim using the commands `echo` and `echom`. The main difference between this is the persistence in time of the outputted message. This can be seen in the next example:

```
:echo "echo test"
:echom "echom test"
```

This will output *echo test* and *echom test* in the bottom of the window of your current buffer. The difference is that when we run the command `messages` we only see the output of the last command run.

```
:messages
```

We can use this behaviour for debugging Vimscript down the line and to see if specific checkpoints or flags of the script have been executed.

Vim's help describes `echo` and `echom` the following way:

echo Echoes each {expr1}, with a space in between. The first {expr1} starts on a new line. Use "\n" to start a new line. Use "\r" to move the cursor to the first column.

echom Echo the expression(s) as a true message, saving the message in the *message-history*.