# commentary.vim

Comment stuff out. Use gcc to comment out a line (takes a count), gc to comment out the target of a motion (for example, gcap to comment out a paragraph), gc in visual mode to comment out the selection, and gc in operator pending mode to target a comment. You can also use it as a command, either with a range like :7,17Commentary, or as part of a :global invocation like with :g/TODO/Commentary. That's it.

I wrote this because 5 years after Vim added support for mapping an operator, I still couldn't find a commenting plugin that leveraged that feature (I overlooked tcomment.vim). Striving for minimalism, it weighs in at under 100 lines of code.

Oh, and it uncomments, too. The above maps actually toggle, and gcgc uncomments a set of adjacent commented lines.

**\*commentary.txt\* Comment stuff out**

Author: Tim Pope <http://tpo.pe/>

License: Same terms as Vim itself (see |license|)

Comment stuff out. Then uncomment it later. Relies on 'commentstring' to be

correctly set, or uses b:commentary\_format if it is set.

\*gc\*

gc{motion} Comment or uncomment lines that {motion} moves over.

\*gcc\*

gcc Comment or uncomment [count] lines.

\*v\_gc\*

{Visual}gc Comment or uncomment the highlighted lines.

\*o\_gc\*

gc Text object for a comment (operator pending mode only.)

\*gcgc\* \*gcu\*

gcgc Uncomment the current and adjacent commented lines.

gcu

\*:Commentary\*

:[range]Commentary Comment or uncomment [range] lines

The |User| CommentaryPost autocommand fires after a successful operation and

can be used for advanced customization.

vim:tw=78:et:ft=help:norl:

$

pend [pend] vi. 悬而未决

adjacent [əˈdʒesənt] adj. 相邻; 邻近的

operator-pending mode 操作待决模式

comment out 注释掉

$

# vim的操作待决模式

普通、插入及可视模式很容易识别，但是vim还有另外的一些容易被忽视的模式，Operator-Pending模式就是一个例子。我们经常使用到它，但通常它只持续不到一秒。

它用来接受命令的状态，这个状态在我们调用操作符（Operator）时被激活，然后什么也不做，直到提供一个motion，完成整个操作。当操作符带决模式被激活时，我们可以像平常一样按<Esc>终止该操作，返回到普通模式。

例子：执行命令dw删除单词，这一模式就在d及w键之间短暂时间间隔内存在。

很多命令都由两个或更多的按键调用，像g，例如命令gU将小写转换成大写，头一个按键只是第二个按键的前缀。这些命令不会激活操作符带决模式，相反，可以把她们当成命名空间，用来扩充命令的数目。只有操作符（Operator）才会激活操作符带决模式。

意义：它允许我们创建自定义的操作符及动作命令，从而让我们可以扩充Vim的词汇。

# surround.vim

\*surround.txt\* Plugin for deleting, changing, and adding "surroundings"

Author: Tim Pope <http://tpo.pe/>

License: Same terms as Vim itself (see |license|)

This plugin is only available if 'compatible' is not set.

**INTRODUCTION** \*surround\*

This plugin is a tool for dealing with pairs of "surroundings." Examples of surroundings include parentheses, quotes, and HTML tags. They are closely related to what Vim refers to as |text-objects|. Provided are mappings to allow for removing, changing, and adding surroundings.

Details follow on the exact semantics, but first, consider the following examples. An asterisk (\*) is used to denote the cursor position.

Old text Command New text ~

"Hello \*world!" ds" Hello world!

[123+4\*56]/2 cs]) (123+456)/2

"Look ma, I'm \*HTML!" cs"<q> <q>Look ma, I'm HTML!</q>

if \*x>3 { ysW( if ( x>3 ) {

my $str = \*whee!; vllllS' my $str = 'whee!';

While a few features of this plugin will work in older versions of Vim, Vim 7 is recommended for full functionality.

**MAPPINGS** \*surround-mappings\*

Delete surroundings is \*ds\* . The next character given determines the target to delete. The exact nature of the target is explained in |surround-targets| but essentially it is the last character of a |text-object|. This mapping deletes the difference between the "i"nner object and "a"n object. This is

easiest to understand with some examples:

Old text Command New text ~

"Hello \*world!" ds" Hello world!

(123+4\*56)/2 ds) 123+456/2

<div>Yo!\*</div> dst Yo!

Change surroundings is \*cs\* . It takes two arguments, a target like with |ds|, and a replacement. \*cS\* changes surroundings, placing the surrounded text on its own line(s) like |yS|. Details about the second argument can be found below in |surround-replacements|. Once again, examples are in order.

Old text Command New text ~

"Hello \*world!" cs"' 'Hello world!'

"Hello \*world!" cs"<q> <q>Hello world!</q>

(123+4\*56)/2 cs)] [123+456]/2

(123+4\*56)/2 cs)[ [ 123+456 ]/2

<div>Yo!\*</div> cst<p> <p>Yo!</p>

\*ys\* takes a valid Vim motion or text object as the first object, and wraps it using the second argument as with |cs|. (It's a stretch, but a good mnemonic for "ys" is "you surround".)

Old text Command New text ~

Hello w\*orld! ysiw) Hello (world)!

As a special case, \*yss\* operates on the current line, ignoring leading whitespace.

Old text Command New text ~

Hello w\*orld! yssB {Hello world!}

There is also \*yS\* and \*ySS\* which indent the surrounded text and place it on a line of its own.

In visual mode, a simple "S" with an argument wraps the selection. This is referred to as the \*vS\* mapping, although ordinarily there will be additional keystrokes between the v and S. In linewise visual mode, the surroundings are placed on separate lines and indented. In blockwise visual

mode, each line is surrounded.

A "gS" in visual mode, known as \*vgS\* , behaves similarly. In linewise visual mode, the automatic indenting is suppressed. In blockwise visual mode, this enables surrounding past the end of the line with 'virtualedit' set (there seems to be no way in Vim Script to differentiate between a jagged end of line selection and a virtual block selected past the end of the line, so two maps were needed).

\*i\_CTRL-G\_s\* \*i\_CTRL-G\_S\*

Finally, there is an experimental insert mode mapping on <C-G>s and <C-S>. Beware that the latter won't work on terminals with flow control (if you accidentally freeze your terminal, use <C-Q> to unfreeze it). The mapping inserts the specified surroundings and puts the cursor between them. If, immediately after the mapping and before the replacement, a second <C-S> or carriage return is pressed, the prefix, cursor, and suffix will be placed on three separate lines. <C-G>S (not <C-G>s) also exhibits this behavior.

**TARGETS** \*surround-targets\*

The |ds| and |cs| commands both take a target as their first argument. The possible targets are based closely on the |text-objects| provided by Vim. All targets are currently just one character.

Eight punctuation marks, (, ), {, }, [, ], <, and >, represent themselves and their counterparts. If the opening mark is used, contained whitespace is also trimmed. The targets b, B, r, and a are aliases for ), }, ], and > (the first two mirror Vim; the second two are completely arbitrary and

subject to change).

Three quote marks, ', ", `, represent themselves, in pairs. They are only searched for on the current line.

A t is a pair of HTML or XML tags. See |tag-blocks| for details. Remember that you can specify a numerical argument if you want to get to a tag other than the innermost one.

The letters w, W, and s correspond to a |word|, a |WORD|, and a |sentence|, respectively. These are special in that they have nothing to delete, and used with |ds| they are a no-op. With |cs|, one could consider them a slight shortcut for ysi (cswb == ysiwb, more or less).

A p represents a |paragraph|. This behaves similarly to w, W, and s above; however, newlines are sometimes added and/or removed.

**REPLACEMENTS** \*surround-replacements\*

A replacement argument is a single character, and is required by |cs|, |ys|, and |vS|. Undefined replacement characters (with the exception of alphabetic characters) default to placing themselves at the beginning and end of the destination, which can be useful for characters like / and |.

If either ), }, ], or > is used, the text is wrapped in the appropriate pair of characters. Similar behavior can be found with (, {, and [ (but not <), which append an additional space to the inside. Like with the targets above, b, B, r, and a are aliases for ), }, ], and >. To fulfill the common need for code blocks in C-style languages, <C-}> (which is really <C-]>) adds braces on lines separate from the content.

If t or < is used, Vim prompts for an HTML/XML tag to insert. You may specify attributes here and they will be stripped from the closing tag. If replacing a tag, its attributes are kept in the new tag. End your input with > to discard the those attributes. If <C-T> is used, the tags will appear on lines by

themselves.

If s is used, a leading but not trailing space is added. This is useful for removing parentheses from a function call with csbs.

**CUSTOMIZING** \*surround-customizing\*

The following adds a potential replacement on "-" (ASCII 45) in PHP files. (To determine the ASCII code to use, :echo char2nr("-")). The carriage return will be replaced by the original text.

>

autocmd FileType php let b:surround\_45 = "<?php \r ?>"

<

This can be used in a PHP file as in the following example.

Old text Command New text ~

print "Hello \*world!" yss- <?php print "Hello world!" ?>

Additionally, one can use a global variable for globally available replacements.

>

let g:surround\_45 = "<% \r %>"

let g:surround\_61 = "<%= \r %>"

<

Advanced, experimental, and subject to change: One can also prompt for replacement text. The syntax for this is to surround the replacement in pairs of low numbered control characters. If this sounds confusing, that's because it is (but it makes the parsing easy). Consider the following example for a LaTeX environment on the "l" replacement.

>

let g:surround\_108 = "\\begin{\1environment: \1}\r\\end{\1\1}"

<

When this replacement is used, the user is prompted with an "environment: " prompt for input. This input is inserted between each set of \1's. Additional inputs up to \7 can be used.

Furthermore, one can specify a regular expression substitution to apply.

>

let g:surround\_108 = "\\begin{\1environment: \1}\r\\end{\1\r}.\*\r\1}"

<

This will remove anything after the first } in the input when the text is placed within the \end{} slot. The first \r marks where the pattern begins, and the second where the replacement text begins.

Here's a second example for creating an HTML <div>. The substitution cleverly prompts for an id, but only adds id="" if it is non-blank. You may have to read this one a few times slowly before you understand it.

>

let g:surround\_{char2nr("d")} = "<div\1id: \r..\*\r id=\"&\"\1>\r</div>"

<

Inputting text replacements is a proof of concept at this point. The ugly, unintuitive interface and the brevity of the documentation reflect this.

Finally, It is possible to always append a string to surroundings in insert mode (and only insert mode). This is useful with certain plugins and mappings that allow you to jump to such markings.

>

let g:surround\_insert\_tail = "<++>"

<

ISSUES \*surround-issues\*

Vim could potentially get confused when deleting/changing occurs at the very end of the line. Please report any repeatable instances of this.

Do we need to use |inputsave()|/|inputrestore()| with the tag replacement?

Indenting is handled haphazardly. Need to decide the most appropriate behavior and implement it. Right now one can do :let b:surround\_indent = 1 (or the global equivalent) to enable automatic re-indenting by Vim via |=|; should this be the default?

vim:tw=78:ts=8:ft=help:norl: