**https://developer.ibm.com/technologies/linux/tutorials/l-vim-script-3/**

**A common mistake with list procedures**

Note that all list-related procedures also return the list they’ve just modified, so you



let sorted\_list = reverse(sort(unsorted\_list))

Doing so would almost always be a serious mistake, however, because even when their retur values are used in this way, list-related functions still modify their original argument. previous example, the listunsorted\_list would also be sorted and reversed. Moreover, unsorted\_list and sorted\_list would now be aliased to the same sorted-and-reversed list (as described under [List" assignments and](#page2) aliasing").

This is highly counterintuitive for most programmers, who typically expectsortfunctionsand like reverse to return modified copies of the original data, without changing the original its

Vimscript lists simply don’t work that way, One such habit is to onlysort(),reverse(), and the like, as pure functions, and to always pass a copy of the data to be modified. You can use the copy() function for this purpose:



let sorted\_list = reverse(sort(copy(unsorted\_list)))

**Filtering and transforming lists**

Two particularly useful procedural list functionsfilter()areand map(). Thefilter() function takes a list and removes those elements that fail to meet some specified criterion:



let filtered\_list = filter(copy(list), criterion\_as\_str)

The call filter() converts the string that is passed as its second argument to a piece of which it then applies to each element of the list that is passed as its first argument. I it repeatedly performseval() on its second argument. For each evaluation, it passes the n element of its first argument to the code, via the special*v:val*. variableIftheresult of the evaluate code is zero (that is, false), the corresponding element is removed from the list.

For example, to remove any negative numbers from a list, type:



let positive\_only = filter(copy(list\_of\_numbers), 'v:val >= 0')

To remove any names from a list that contain the/.\*nix/, type:



let non\_starnix = filter(copy(list\_of\_systems), 'v:val !~ ".\*nix"')

**The map() function**

The map() function is similarfilter(), except that instead of removing some elements, it replaces every element with a user-specified transformation of its original value. The sy



let transformed\_list = map(copy(list), transformation\_as\_str)

Likefilter(),map() evaluates the string passed as its second argument, passing each list element in turn, v:val. But, unlikefilter() , amap() always keeps every element of a list, replacing each value with the result of evaluating the code on that value.

For example, to increase every number in a list by 10, type:



let increased\_numbers = map(copy(list\_of\_numbers), 'v:val + 10')

Or to capitalize each word in a list: type:



let LIST\_OF\_WORDS = map(copy(list\_of\_words), 'toupper(v:val)')

Once again, remember thatfilter() and map() modify their first argument in-place. A very common error when using them is to write something like:



let squared\_values = map(values, 'v:val \* v:val')

instead of:



let squared\_values = map(copy(values), 'v:val \* v:val')

**List concatenation**

You can concatenate lists with*+* theand *+=* operators, like so:

**Listing 15. Concatenating lists**



let activities = ['sleep', 'eat'] + ['game', 'drink']

let activities += ['code']

Remember that both sides must be lists. Don’t *+=*thinkas append"; you can’t use it to add a single value directly to the end of a list:

**Listing 16. Concatenation needs two lists**



let activities += 'code'

" Error: Wrong variable type for +=

**Sublists**

You can extract part of a list by specifying a colon-separated range in the square bracke indexing operation. The limits of the range can be constants, variables with numeric valu numeric expression:

**Listing 17. Extracting parts of a list**



let week = ['Sun','Mon','Tue','Wed','Thu','Fri','Sat'] let weekdays = week[1:5]

let freedays = week[firstfree : lastfree-2]

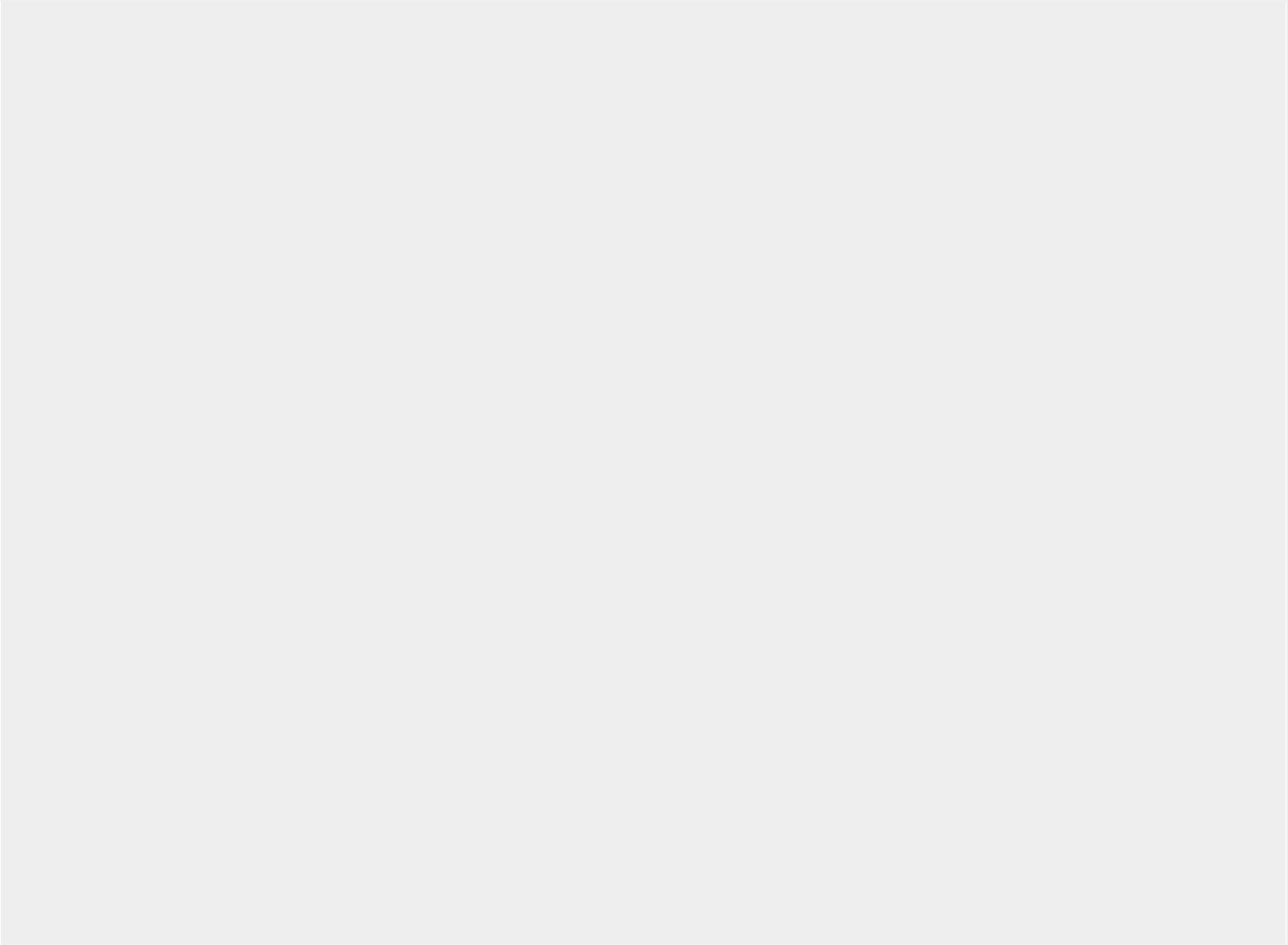
If you omit the starting index, the sublist automatically starts at zero; if you omit the the sublist finishes at the last element. For example, to split a list into two (near-)eq type:

**Listing 18. Splitting a list into two sublists**

|  |  |  |  |
| --- | --- | --- | --- |
| let middle = len(data)/2 | | |  |
| let | first\_half | = data[: middle-1] | " same as: data[0 : middle-1] |
| let | second\_half | = data[middle :] | " same as: data[middle : len(data)-1] |

**Example 1: Revisiting autoalignments**

**Listing 19. The original AlignAssignments() function**



function AlignAssignments ()

|  |  |  |  |
| --- | --- | --- | --- |
| " | Patterns needed | | to locate assignment operators... |
| let | | ASSIGN\_OP = | '[-+\*/%|&]\?=\@<!=[=~]\@!' |
| let | | ASSIGN\_LINE = | '^\(.\{-}\)\s\*\(' . ASSIGN\_OP . '\)' |
| " | Locate block of | | code to be considered (same indentation, no blanks) |

|  |  |  |
| --- | --- | --- |
| let indent\_pat | = | '^' . matchstr(getline('.'), '^\s\*') . '\S' |
| let firstline | = | search('^\%('. indent\_pat . '\)\@!','bnW') + 1 |
| let lastline | = | search('^\%('. indent\_pat . '\)\@!', 'nW') - 1 |
| if lastline < 0 |  |  |
| let lastline | | = line('$') |
| endif |  |  |

" Find the column at which the operators should be aligned...

let max\_align\_col = 0 let max\_op\_width = 0

for linetext in getline(firstline, lastline) " Does this line have an assignment in it?

let left\_width = match(linetext, '\s\*' . ASSIGN\_OP)

" If so, track the maximal assignment column and operator width...

if left\_width >= 0

let max\_align\_col = max([max\_align\_col, left\_width])

let op\_width = strlen(matchstr(linetext, ASSIGN\_OP))

let max\_op\_width = max([max\_op\_width, op\_width+1])

endif

endfor

" Code needed to reformat lines so as to align operators...

let FORMATTER = '\=printf("%-\*s%\*s", max\_align\_col, submatch(1),

\ max\_op\_width, submatch(2))'

" Reformat lines with operators aligned in the appropriate column...

for linenum in range(firstline, lastline) let oldline = getline(linenum)

let newline = substitute(oldline, ASSIGN\_LINE, FORMATTER, "") call setline(linenum, newline)

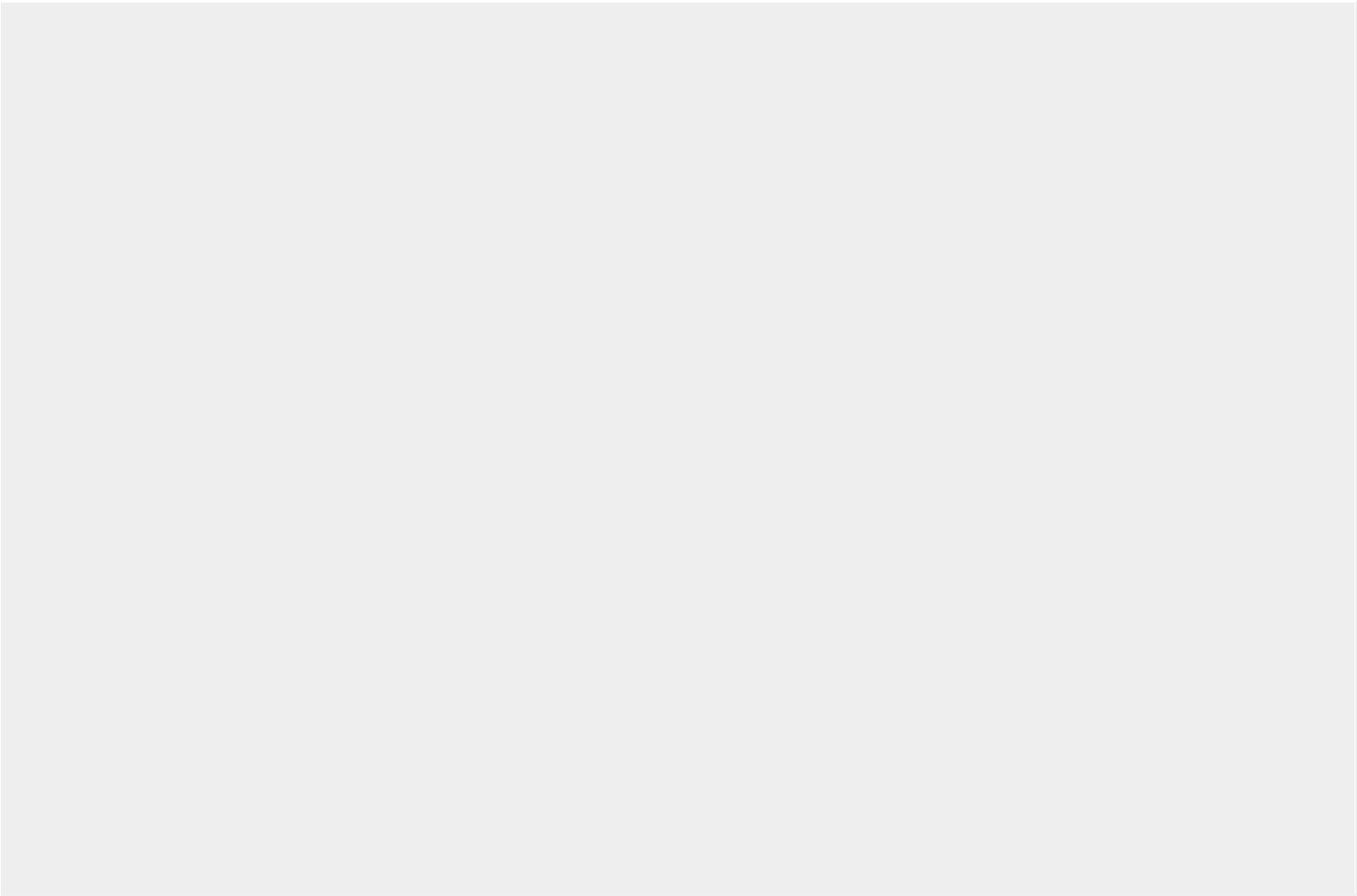
endfor endfunction

One deficiency of this function is that it has to grab each line being processed twice: o firstfor loop) to gather information on the paragraph’s existing structure, and a second t finalfor loop) to adjust each line to fit the new structure.

This duplicated effort is clearly suboptimal. It would be better to store the lines in so data structure and reuse them directly. Knowing what you do about lists, it is indeed pos rewriteAlignAssignments() more efficiently and more cleanly. Listing 20 shows a new versio

the function that takes advantage of several list data structures and the various list-ma functions described earlier.

**Listing 20. An updated AlignAssignments() function**



|  |  |  |  |
| --- | --- | --- | --- |
| function! AlignAssignments () | | |  |
| " Patterns needed to locate assignment operators... | | |  |
| let ASSIGN\_OP |  | = '[-+\*/%|&]\?=\@<!=[=~]\@!' |  |
| let ASSIGN\_LINE = '^\(.\{-}\)\s\*\(' . ASSIGN\_OP . '\)\(.\*\)$' | | | |
| " Locate block | | of code to be considered (same indentation, no blanks) | |
| let indent\_pat | | = '^' . matchstr(getline('.'), '^\s\*') . '\S' |  |
| let firstline |  | = search('^\%('. indent\_pat . '\)\@!','bnW') + 1 | |
| let lastline |  | = search('^\%('. indent\_pat . '\)\@!', 'nW') - 1 | |
| if lastline < 0 | |  |  |
| let lastline = line('$') | | |  |
| endif |  |  |  |
| " Decompose lines at assignment operators... | | |  |
| let lines = [] | |  |  |
| for linetext in getline(firstline, lastline) | | |  |
| let fields | | = matchlist(linetext, ASSIGN\_LINE) |  |
| call add(lines, fields[1:3]) | | |  |
| endfor |  |  |  |
| " Determine maximal lengths of lvalue and operator... | | |  |
| let op\_lines = | | filter(copy(lines),'!empty(v:val)') |  |
| let max\_lval | = | max( map(copy(op\_lines), 'strlen(v:val[0])') | ) + 1 |
| let max\_op | = | max( map(copy(op\_lines), 'strlen(v:val[1])' | ) ) |

" Recompose lines with operators at the maximum length...

let linenum = firstline for line in lines

if !empty(line) let newline

\ = printf("%-\*s%\*s%s", max\_lval, line[0], max\_op, line[1], line[2]) call setline(linenum, newline)

endif

let linenum += 1 endfor

endfunction

Note that the first two code blocks within the new function are almost identical to those original. As before, they locate the range of lines whose assignments are to be aligned, the current indentation of the text.

The changes begin in the third code block, which uses the two-argument form of the built-getline() function to return a list of all the lines in the range to be realigned.

The for loop then iterates through each line, matching it against the regular expression

ASSIGN\_LINE using the builtmatchlist() function:



let fields = matchlist(linetext, ASSIGN\_LINE)

The call matchlist() returns a list of all the fields captured by the regex (that is, any matched by those parts of the pattern \(...\) delimiters). In this example, if the match succeeds, the resulting fields are a decomposition that separateslvalueout, operator,the and rvalue of any assignment line.

Specifically, a successful matchlist() will return a list with the following elements:

* The full line (becausematchlist()*always* returns the entire match as its first element)
* Everything to the left of the assignment operator
* The assignment operator itself
* Everything to the right of the assignment operator

In that case, the calladd()to adds a sublist of the final three fields to the lines list. If failed (that is, the line didn’t contain an assignment),matchlist()thenwill return an empty list, so the sublist add() appends fields[1:3] below) will also be empty. This will be used to ind a line of no further interest to the reformatter:



call add(lines, fields[1:3])

The fourth code block deploysfilter() and map() functions to analyze the structure of each line containing an assignment. It firstfilter() ato winnow the list of lines, keeping only those that were successfully decomposed into multiple components by the previous code blo



let op\_lines = filter(copy(lines), '!empty(v:val)')

Next the function determines the length of each assignment’slvalue,by mapping thestrlen() function over a copy of the filtered lines:



map(copy(op\_lines), 'strlen(v:val[0])')

The resulting listlvalue lengths is then passed to the max()-infunction to determine the longestlvalue in any assignment. The maximal length determines the column at which all th assignment operators will need to be aligned (that is, one column beyondlvalue):widest



let max\_lval = max( map(copy(op\_lines),'strlen(v:val[0])') ) + 1

In the same way, the final line of the fourth code block determines the maximal number of required to accommodate the various assignment operators that were found, by mapping and maximizing their individual string lengths:



let max\_op = max( map(copy(op\_lines),'strlen(v:val[1])' ) )

The final code block then reformats the assignment lines, by iterating through the origin line numberslinenum) and through each line in the lines list, in parallel:



let linenum = firstline

for line in lines

Each iteration of the loop checks whether a particular line needs to be reformatted (that it was decomposed successfully around an assignment operation). If so, the function creat new version of the line, usingprintf() to reformat the line’s components:

ng the Vim editor, Part 3: Built-in lists Page 9 of19



if !empty(line)

let newline = printf("%-\*s%\*s%s", max\_lval, line[0], max\_op, line[1], line[2])

That new line is then written back to the editor buffersetline()calling,and the line tracking is updated for the next iteration:



call setline(linenum, newline) endif

let linenum += 1

Once all the lines have been processed, the buffer will have been completely updated and the relevant assignment operators aligned to a suitable column. Because it can take advan of Vimscript's excellent support for lists and list operations, the code for this second AlignAssignments() is about 15 percent shorter than that of the previous version. Far mor importantly, however, the function does only one-third as many buffer accesses, and the c much clearer and more maintainable.

**Example 2: Enhancing Vim’s completion facilities**

For example, if you had the preceding two paragraphs in a buffer, and then—in insertion m you typed:



My use of Vim is increasingly so<CTRL-N>

Vim would search the text and determine that the only word beginning*"so..."***w**asith *sophisticated*, and would complete that word immediately:



My use of Vim is increasingly sophisticated\_

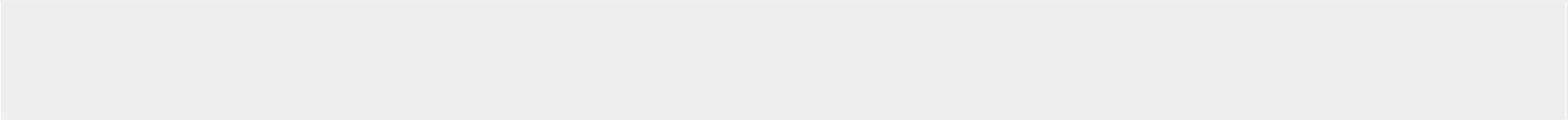
On the other hand, if you typed:



My repertoire of editing skills is bu<CTRL-N>

Vim would detect three possible completions:*built*,*buffer*, and*buffers*. By default, it would show a menu of alternatives:

**Listing 21. Text completion with alternatives**



My repertoire of editing skills is bu\_ built buffer buffers

and you could then use a sequence**CTRL**of**-N** and **CTRL-P** (or the up- and down-arrows) to step through the menu and select the word you wanted.

To cancel a completion at any time, you can**CTRL**type**-E** ; to accept and insert the currently selected alternative, you can**CTRL**type**-Y** . Typing anything else (typically, a space or newline also accepts and inserts the currently selected word, as well as whatever extra character typed.

**Designing smarter completions**

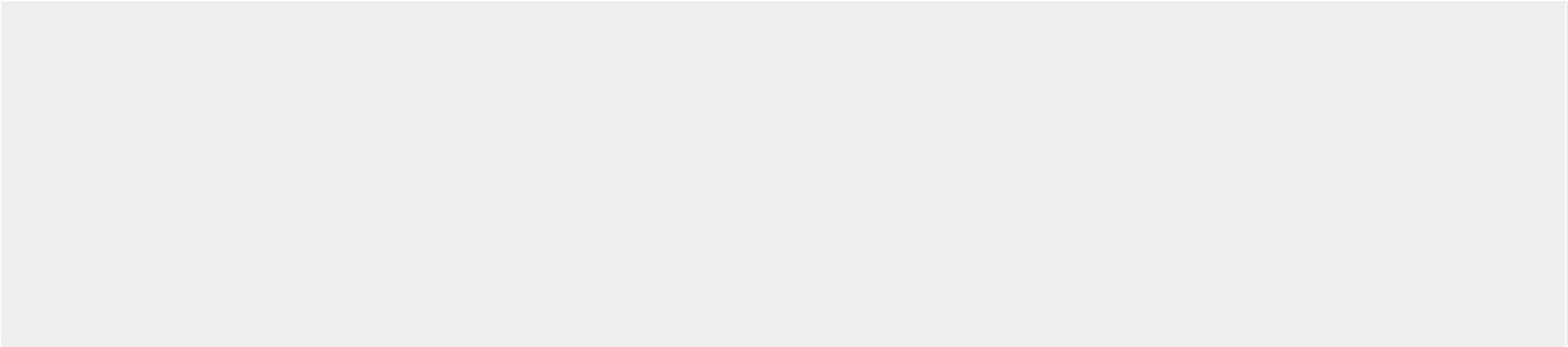
By default, it matches only sequences of "keyword" characters (alphanumerics and underscore), and it has no deep sense of context beyond matching what’s immediately to th of the cursor.

Happily, with Vimscript, we can easily remedy those deficiencies. Let’s**TAB**redefinekeyin the insertion mode so that it can be taught to recognize patterns in the text on either side and select an appropriate completion for that context. We’ll also arrange it so that, if mechanism doesn’t recognize the current insertion context, it will fall back**CTRL**to**-**Vim’s bui **N** completion mechanism. Oh, and while we’re at it, we should probably make sure we can sthe **TAB** key to type tab characters, where that’s appropriate.

**Specifying smarter completions**

To build this smarter completion mechanism, we’ll need to store a series of "contextual r to a completion request. So we’ll need a list. Or rather, a list of lists, given each con response will itself consist of four elements. Listing 22 shows how to set up that data s

**Listing 22. Setting up a look-up table in Vimscript**



* Table of completion specifications (a list of lists)...

let s:completions = []

* Function to add user-defined completions...

function! AddCompletion (left, right, completion, restore)

call insert(s:completions, [a:left, a:right, a:completion, a:restore])

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| endfunction |  |  |  |  |  |
| let s:NONE = "" |  |  |  |  |  |
| " Table of completions... | |  |  |  |  |
| " | Left | Right | Complete with... | Restore |  |
| " | ===== | ======= | ==================== | ======= |  |
| call AddCompletion( | '{', | s:NONE, | "}", | 1 | ) |
| call AddCompletion( | '{', | '}', | "\<CR>\<C-D>\<ESC>O", | 0 | ) |

Scripting the Vim editor, Part 3: Built-in lists Page 11 of19

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| developerWorks® |  |  |  |  | ibm.com/developerWorks/ | |
|  |  |  |  |  |  |  |
| call AddCompletion( | '\[', | s:NONE, | "]", | 1 | ) |  |
| call AddCompletion( | '\[', | '\]', | "\<CR>\<ESC>O\<TAB>", | 0 | ) |  |
| call AddCompletion( | '(', | s:NONE, | ")", | 1 | ) |  |
| call AddCompletion( | '(', | ')', | "\<CR>\<ESC>O\<TAB>", | 0 | ) |  |
| call AddCompletion( | '<', | s:NONE, | ">", | 1 | ) |  |
| call AddCompletion( | '<', | '>', | "\<CR>\<ESC>O\<TAB>", | 0 | ) |  |
| call AddCompletion( | '"', | s:NONE, | '"', | 1 | ) |  |
| call AddCompletion( | '"', | '"', | "\\n", | 1 | ) |  |
| call AddCompletion( | "'", | s:NONE, | "'", | 1 | ) |  |
| call AddCompletion( | "'", | "'", | s:NONE, | 0 | ) |  |

The list-of-lists we create will act as a table of contextual response specifications, an stored in the list s:completions. Each entry in the list will itself be a list, with fo

* A string specifying a regular expression to match what’s to the left of the cursor
* A string specifying a regular expression to match what’s to the right of the cursor
* A string to be inserted when both contexts are detected
* A flag indicating whether to automatically restore the cursor to its pre-completion after the completion text has been inserted

To populate the table, we create a small AddCompletion(). This function expects four arguments: the left and right contexts, and the replacement text,restore cursor" flag. The series of arguments are simply collected into a single list:



[a:left, a:right, a:completion, a:restore]

and that list is then prepended as a single element at thes:completionsthevariable using the built insert() function:



call insert(s:completions, [a:left, a:right, a:completion, a:restore])

Repeated calls AddCompletion() therefore build up a list of lists, each of which specifie completion. The code in Listing 22 does the work.

The first callAddCompletion():

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| " | Left | Right | Complete with... | Restore |  |
| " | ===== | ======= | ==================== | ======= |  |
| call AddCompletion( '{', | | s:NONE, | '}', | 1 | ) |

specifies that, when the new mechanism encounters a curly brace to the left of the cursor nothing to the right, it should insert a closing curly brace and then restore the cursor completion position. That is, when completing:



while (1) {\_

(where the \_ represents the cursor), the mechanism will now produce:



while (1) {\_}

leaving the cursor conveniently in the middle of the newly closed block.

Scripting the Vim editor, Part 3: Built-in lists Page 12 of19

ibm.com/developerWorks/ developerWorks®

The second call AddCompletion():

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| " | Left | Right | Complete with... | Restore |  |
| " | ===== | ======= | ==================== | ======= |  |
| call AddCompletion( | '{', | '}', | "\<CR>\<C-D>\<ESC>O", | 0 | ) |

then proceeds to make the completion mechanism smarter still. It specifies that, when the mechanism encounters an opening curly brace to the left of the cursor and a closing brace right of the cursor, it should insert a newline, outdent the closing**CTRL-D**curly),then(viaescape from insertion mode**ESC**( ) and open a new line above the closing Ocurly). (

Assuming the smartindent" option is enabled, the net effect of the sequence is that, when press **TAB** in the following context



while (1) {\_}

the mechanism will produce:



while (1) {

\_

}

In other words, because of the first two additions to the completion**TAB**-completiontable, after an opening brace closes it on the same line, and then immediately doing**TAB**a -secondcompletion "stretches" the block across several lines (with correct indenting).

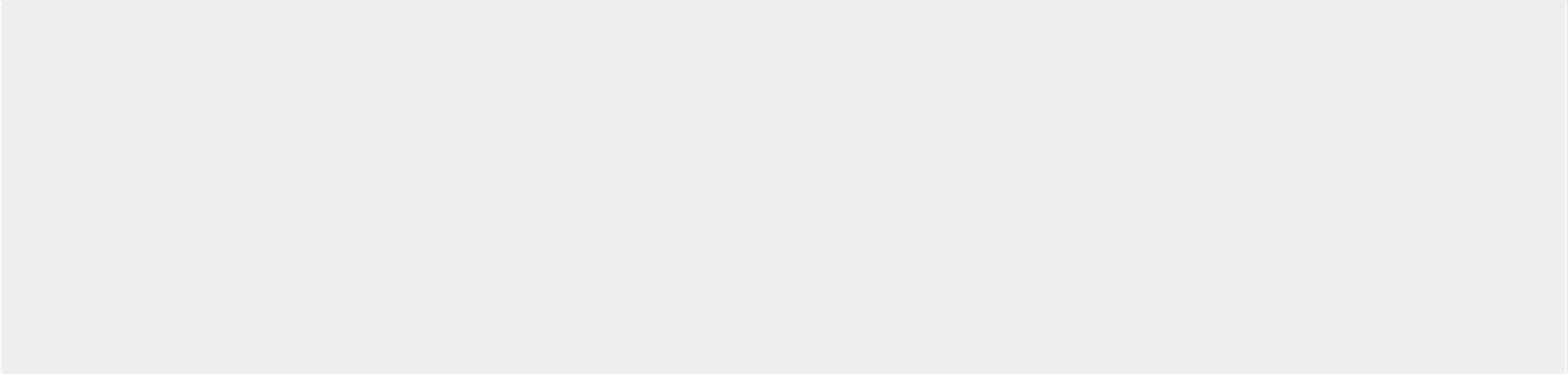
The remaining callsAddCompletion() replicate this arrangement for the three other kinds o brackets (square, round, and angle) and also provide special completion semantics for sin and double-quotes. Completing after a double-quote appends the matching double-quote, whi completing between two double quotes appends\n a(newline) metacharacter. Completing after

a single quote appends the matching single quote, and then a second completion attempt do nothing.

**Implementing smarter completions**

Once the list of completion-specifications has been set up, all that remains is to implem function to select the appropriate completion from the table, and then bind that**TAB**function key. Listing 23 shows that code.

**Listing 23. A smarter completion function**



" Implement smart completion magic...

function! SmartComplete ()

* Remember where we parked...

let cursorpos = getpos('.') let cursorcol = cursorpos[2] let curr\_line = getline('.')

* Special subpattern to match only at cursor position...

let curr\_pos\_pat = '\%' . cursorcol . 'c'

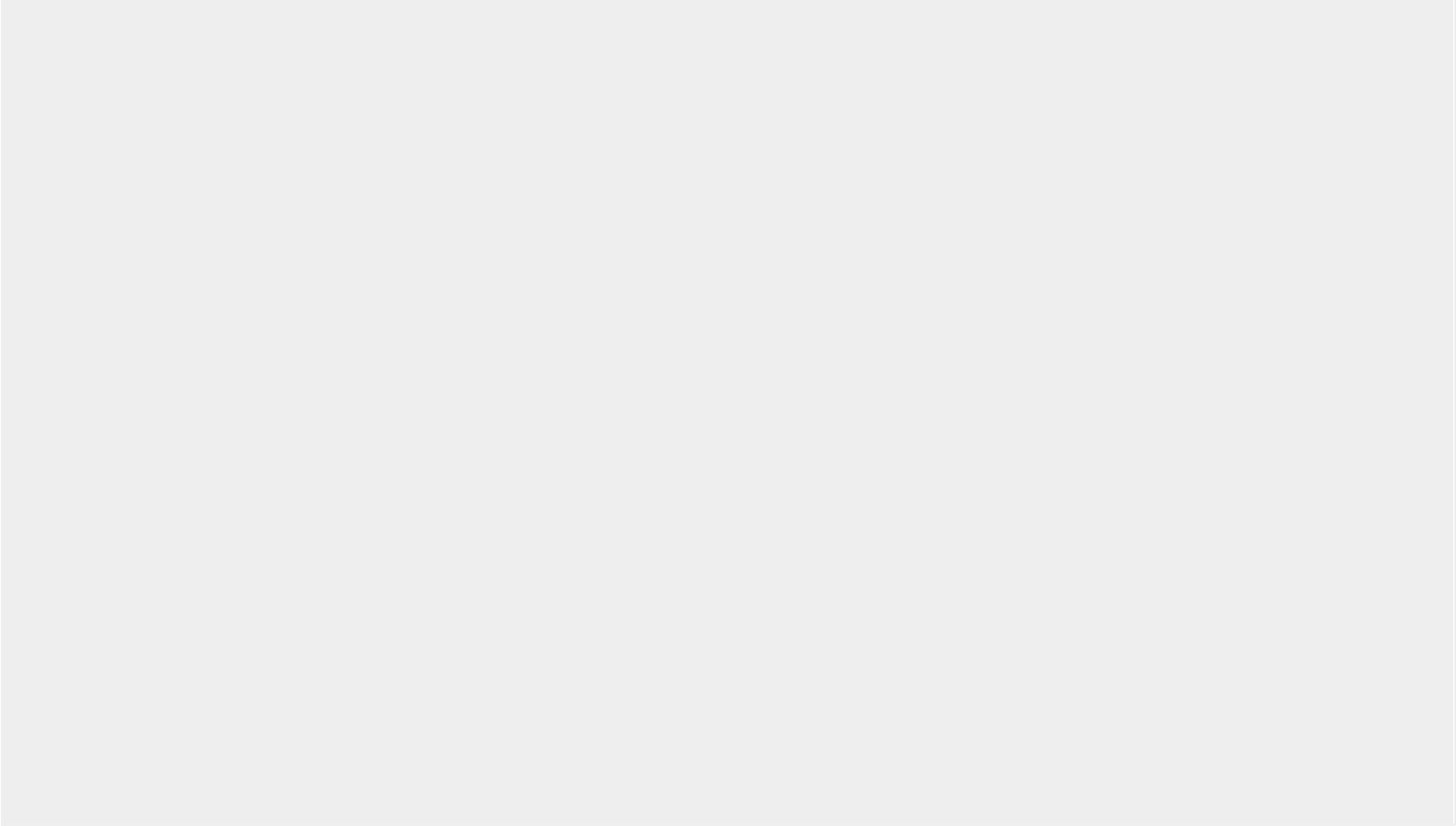
* Tab as usual at the left margin...

if curr\_line =~ '^\s\*' . curr\_pos\_pat return "\<TAB>"

Scripting the Vim editor, Part 3: Built-in lists Page 13 of19

developerWorks® ibm.com/developerWorks/

endif



" How to restore the cursor position...

let cursor\_back = "\<C-O>:call setpos('.'," . string(cursorpos) . ")\<CR>"

" If a matching smart completion has been specified, use that...

for [left, right, completion, restore] in s:completions let pattern = left . curr\_pos\_pat . right

if curr\_line =~ pattern

" Code around bug in setpos() when used at EOL...

if cursorcol == strlen(curr\_line)+1 && strlen(completion)==1 let cursor\_back = "\<LEFT>"

endif

" Return the completion...

return completion . (restore ? cursor\_back : "") endif

endfor

" If no contextual match and after an identifier, do keyword completion...

if curr\_line =~ '\k' . curr\_pos\_pat return "\<C-N>"

" Otherwise, just be a <TAB>...

else

return "\<TAB>" endif

endfunction

" Remap <TAB> for smart completion on various characters...

inoremap <silent> <TAB> <C-R>=SmartComplete()<CR>

The SmartComplete() function first locates the cursor, using getpos()builtfunction-in with a

'.' argument (that is, "get position of cursor"). That call returns a list of four elemen number (usually zero), the row and column numbers (both indexed from 1), and a special "v offset" (which is also usually zero, and not relevant here). We’re primarily interested i two values, as they indicate the location of the cursor. SmartComplete() needs the

column number, which is extracted by indexing into thegetpos()thatreturned, like so:



let cursorcol = cursorpos[2]

The function also needs to know the text on the current line, which can be retrieved usin getline(), and is storedcurr\_line.

SmartComplete() is going to convert each entrys:completions table into a pattern to be matched against the current line. In order to correctly match left and right contexts aro cursor, it needs to ensure the pattern matches only at the cursor’s column. Vim has a spe subpattern for that:\%Nc (whereN is the column number required). So, the function creates th subpattern by interpolating the cursor’s column position found earlier:



let curr\_pos\_pat = '\%' . cursorcol . 'c'

Because we’re eventually going to bind this function**TAB** tokey,thewe’d like the function to stil insert **TAB**a whenever possible, and especially at the start ofSmartComplete() first

checks if there is only whitespace to the left of the cursor position, in which case it r tabspace:

Scripting the Vim editor, Part 3: Built-in lists Page 14 of19

ibm.com/developerWorks/ developerWorks®



if curr\_line =~ '^\s\*' . curr\_pos\_pat return "\<TAB>"

endif

If the cursor isn’t at the start of SmartComplete() needs to check all the entries in the completion table and determine which, if any, apply. Some of those entries will specify t cursor should be returned to its previous position after completion, which will require e a custom command from within insertion mode. That command is simply a call to the built-i setpos() function, passing the value the original information from thegetpos(). call to

To execute that function call from within insertion mode**CTRL**requires**-O**escape (see:help i\_CTRL-O in any Vim session).SmartComplete() prebuilds the necessary**CTRL-O** command as a string and storescursor\_back:



let cursor\_back = "\<C-O>:call setpos('.'," . string(cursorpos) . ")\<CR>"

**A more-sophisticated for loop**

To walk through the completions table, the function uses a special versionforstatementofthe.

The standardfor loop in Vimscript walks through a one-dimensional list, one element at a

**Listing 24. A standard for loop**

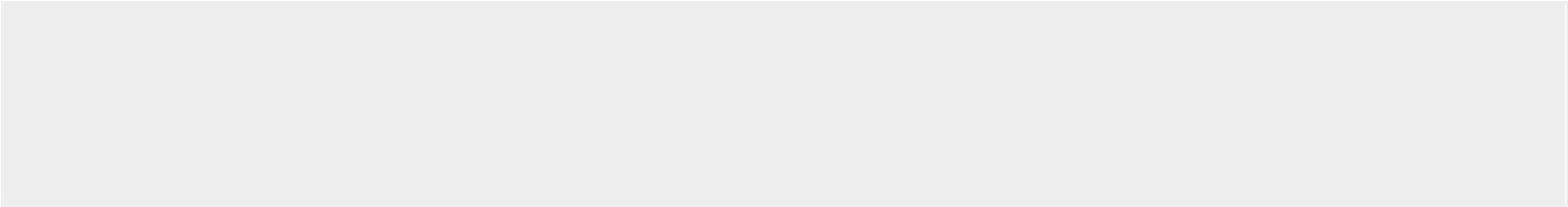


for name in list echo name

endfor

However, if the list is two-dimensional (that is, each element is itself a list), then yo "unpack" the contents of each nested list as it is iterated. You could do that like so:

**Listing 25. Iterating over nested lists**



for nested\_list in list\_of\_lists

|  |  |  |  |
| --- | --- | --- | --- |
| let name | | = nested\_list[0] | |
| let | rank | = | nested\_list[1] |
| let | serial = | | nested\_list[2] |

echo rank . ' ' . name . '(' . serial . ')' endfor

but Vimscript has a much cleaner shorthand for it:

**Listing 26. A cleaner shorthand for iterating over nested lists**



for [name, rank, serial] in list\_of\_lists

echo rank . ' ' . name . '(' . serial . ')' endfor

On each iteration, the loop takes the next nestedlist\_of\_lists and assigns the first element of that nested listnameto, the second nested elementrank , and the thirdserial.

Using this special formforofloop makes it easy SmartComplete() to walk through the table of completions and give a logical name to each component of each completion:



for [left, right, completion, restore] in s:completions

Scripting the Vim editor, Part 3: Built-in lists Page 15 of19

developerWorks® ibm.com/developerWorks/

**Recognizing a completion context**

Within the loop,SmartComplete() constructs a regular expression by placing the left and righ context patterns around the special subpattern that matches the cursor position:



let pattern = left . curr\_pos\_pat . right

If the current line matches the resulting regex, then the function has found the correct (the text of which is already in completion) and can return it immediately. Of course, it to append the cursor restoration command it built earlier, if the selected completion has it (that restore is true).

Unfortunately, setpos() -based cursor restoration command has a problem. In Vim versions 7.2 or earlier, there’s an obscure idiosyncrasysetpos()in: it doesn’t correctly reposition the cursor in insertion mode if the cursor was previously at the end of a line and the comple to be inserted is only one character long. In that special case, the restoration command changed to a single left-arrow, which moves the cursor back over the one newly inserted c

So, before the selected completion is returned, the following code makes that change:

**Listing 27. Restoring the cursor after a one-character insertion at end-of**



if cursorcol == strlen(curr\_line)+1 && strlen(completion)==1 let cursor\_back = "\<LEFT>"

endif

|  |  |  |
| --- | --- | --- |
| All that remains is to | return the selected completion, | cursor\_backthe command if |
| cursor restoration was | requested: | |



return completion . (restore ? cursor\_back : "")

If none of the entries from the completion table match the currentSmartComplete()

will eventually fall outfor loopthe and will then try two final alternatives. If the charact

immediately before the cursor was a "keyword" character, it invokes a normal keyword-comp by returning **CTRL**a**-N** :

**Listing 28. Falling back to CTRL-N behavior**

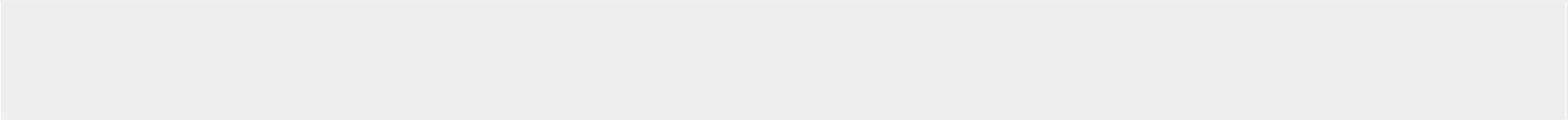


" If no contextual match and after an identifier, do keyword completion...

if curr\_line =~ '\k' . curr\_pos\_pat return "\<C-N>"

Otherwise, no completion was possible, so it falls back to acting **TAB**likekey,anormalby returning a literal tab character:

**Listing 29. Falling back to normal TAB key behavior**



" Otherwise, just be a <TAB>...

else

return "\<TAB>" endif

**Deploying the new mechanism**

Now we just have to make the**TAB** key callSmartComplete() in order to work out what it should insert. That’s done withinoremap, like so:



inoremap <silent> <TAB> <C-R>=SmartComplete()<CR>

The key-mapping converts any insert-mode**TAB** to a**CTRL-R=** , callingSmartComplete() and inserting the completion string it returns:help i\_CTRL-R or the [first article in](http://www.ibm.com/developerworks/linux/library/l-vim-script-1/index.html) thisforseries details of this mechanism).

The inoremap form ofimap is used here because some of the completion strings that SmartComplete() returns also contain**TAB**a character. If a regularimapwere used, inserting

that returned**TAB** would immediately cause this same key-mapping to be re-invoked, calling SmartComplete() again, which might return another**TAB**, and so on.

With theinoremap in place, we now have**TAB**a key that can:

* Recognize special user-defined insertion contexts and complete them appropriately
* Fall back to regular**CTRL-N** completion after an identifier
* Still act like**TAB** a everywhere else

In addition, with the code from Listings 22 and 23 placed in your .vimrc file, you will b add new contextual completions simply by extending the completion table with extra calls AddCompletion(). For example, you could make it easier to start new Vimscript functions w



call AddCompletion( 'function!\?', "", "\<CR>endfunction", 1 )

|  |  |  |  |
| --- | --- | --- | --- |
| so that | tabbing immediately after a function keyword appends the |  |  |
| endfunction |  |
| keyword | on the next line. |  |  |

Or, you could autocomplete C/C++ comments intelligently (assumingcindenttheoption is also set) with:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| call | AddCompletion( | '/\\*', | "", | '\*/', | 1 ) |
| call | AddCompletion( | '/\\*', | '\\*/', | "\<CR>\* \<CR>\<ESC>\<UP>A", | 0 ) |

So that:



/\*\_<TAB>

appends a closing comment delimiter after the cursor:



/\*\_\*/

and a second**TAB** at that point inserts an elegant multiline comment and positions the cur the middle of it:

/\*

\* \_

\*/