# Scripting the Vim editor, Part 3: Built-in lists

## Explore Vimscript's support for lists and arrays

Damian Conway, Dr. January 27, 2010

Vimscript provides excellent support for operating on collections of data, a cornerstone of programming. In this third article in the series, learn how to use Vimscript's built-in lists to ease everyday operations such as reformatting lists, filtering sequences of filenames, and sorting sets of line numbers. You'll also walk through examples that demonstrate the power of lists to extend and enhance two common uses of Vim: creating a user-defined function to align assignment operators, and improving the built-in text completions mechanism.

View more content in this series

The heart of all programming is the creation and manipulation of data structures. So far in this series, we've considered only Vimscript's scalar data types (strings, numbers, and booleans) and the scalar variables that store them. But the true power of programming Vim becomes apparent when its scripts can operate on entire collections of related data at once: reformatting lists of text lines, accessing multidimensional tables of configuration data, filtering sequences of filenames, and sorting sets of line numbers.

In this article, we'll explore Vimscript's excellent support for lists and the arrays that store them, as well as the language's many built-in functions that make using lists so easy, efficient, and maintainable.

## Lists in Vimscript

In Vimscript, a list is a sequence of scalar values: strings, numbers, references, or any mixture thereof.

Vimscript lists are arguably misnamed. In most languages, a "list" is a value (rather than a container), an immutable ordered sequence of simpler values. In contrast, lists in Vimscript are mutable and in many ways far more like (references to) anonymous-array data structures. A Vimscript variable that is storing a list is, for most purposes, an array.

You create a list by placing a comma-separated sequence of scalar values inside a pair of square brackets. List elements are indexed from zero, and are accessed and modified via the usual notation: postfix square brackets with the index inside them:

## Listing 1. Creating a list

```
let data = [1,2,3,4,5,6,"seven"]
echo data[0]                |" echoes: 1
let data[1] = 42            |" [1,42,3,4,5,6,"seven"]
let data[2] += 99           |" [1,42,102,4,5,6,"seven"]
let data[6] .= ' samurai'   |" [1,42,102,4,5,6,"seven samurai"]
```

You can also use indices less than zero, which then count backward from the end of the list. So the final statement of the previous example could also be written like so:

```
let data[-1] .=  ' samurai'
```

As in most other dynamic languages, Vimscript lists require no explicit memory management: they automatically grow or shrink to accommodate the elements they're asked to store, and they're automatically garbage-collected when the program no longer requires them.

### Nested lists

In addition to storing strings or numbers, a list can also store other lists. As in C, C++, or Perl, if a list contains other lists, it acts like a multidimensional array. For example:

## Listing 2. Creating a nested list

```
let pow = [
\   [ 1, 0, 0, 0  ],
\   [ 1, 1, 1, 1  ],
\   [ 1, 2, 4, 8  ],
\   [ 1, 3, 9, 27 ],
\]
" and later...
echo pow[x][y]
```

Here, the first indexing operation (`pow[x]`) returns one of the elements of the list in `pow`. That element is itself a list, so the second indexing (`[y]`) returns one of the nested list's elements.

### List assignments and aliasing

When you assign any list to a variable, you're really assigning a pointer or reference to the list. So, assigning from one list variable to another causes them to both point at or refer to the same underlying list. This usually leads to unpleasant action-at-a-distance surprises like the one you see here:

## Listing 3. Assign with caution

```
let old_suffixes = ['.c', '.h', '.py']
let new_suffixes = old_suffixes
let new_suffixes[2] = '.js'
echo old_suffixes      |" echoes: ['.c', '.h', '.js']
echo new_suffixes      |" echoes: ['.c', '.h', '.js']
```

To avoid this aliasing effect, you need to call the built-in `copy()` function to duplicate the list, and then assign the copy instead:

### Listing 4. Copying a list

```
let old_suffixes = ['.c', '.h', '.py']
let new_suffixes = copy(old_suffixes)
let new_suffixes[2] = '.js'
echo old_suffixes      |" echoes: ['.c', '.h', '.py']
echo new_suffixes      |" echoes: ['.c', '.h', '.js']
```

Note, however, that `copy()` only duplicates the top level of the list. If any of those values is itself a nested list, it's really a pointer/reference to some separate external list. In that case, `copy()` will duplicate that pointer/reference, and the nested list will still be shared by both the original and the copy, as shown here:

### Listing 5. Shallow copy

```
let pedantic_pow = copy(pow)
let pedantic_pow[0][0] = 'indeterminate'
" also changes pow[0][0] due to shared nested list
```

If that's not what you want (and it's almost always not what you want), then you can use the built-in `deepcopy()` function instead, which duplicates any nested data structure "all the way down":

### Listing 6. Deep copy

```
let pedantic_pow = deepcopy(pow)
let pedantic_pow[0][0] = 'indeterminate'
" pow[0][0] now unaffected; no nested list is shared
```

## Basic list operations

Most of Vim's list operations are provided via built-in functions. The functions usually take a list and return some property of it:

### Listing 7. Finding size, range, and indexes

```
" Size of list...
let list_length   = len(a_list)
let list_is_empty = empty(a_list)     " same as: len(a_list) == 0" Numeric minima and maxima...
let greatest_elem = max(list_of_numbers)
let least_elem    = min(list_of_numbers)

" Index of first occurrence of value or pattern in list...
let value_found_at = index(list, value)      " uses == comparison
let pat_matched_at = match(list, pattern)    " uses =~ comparison
```

The `range()` function can be used to generate a list of integers. If called with a single-integer argument, it generates a list from zero to one less than that argument. Called with two arguments, it generates an inclusive list from the first to the second. With three arguments, it again generates an inclusive list, but increments each successive element by the third argument:

### Listing 8. Generating a list using the range() function

```
let sequence_of_ints = range(max)              " 0...max-1
let sequence_of_ints = range(min, max)         " min...max
let sequence_of_ints = range(min, max, step)   " min, min+step,...max
```

You can also generate a list by splitting a string into a sequence of "words":

### Listing 9. Generating a list by splitting text

```
let words = split(str)                          " split on whitespace
let words = split(str, delimiter_pat)           " split where pattern matches
```

To reverse that, you can join the list back together:

### Listing 10. Joining the elements of a list

```
let str = join(list)                            " use a single space char to join
let str = join(list, delimiter)                 " use delimiter string to join
```

## Other list-related procedures

You can explore the many other list-related functions by typing `:help function-list` in any Vim session, then scrolling down to "`List manipulation`"). Most of these functions are actually procedures, however, because they modify their list argument in-place.

For example, to insert a single extra element into a list, you can use `insert()` or `add()`:

### Listing 11. Adding a value to a list

```
call insert(list, newval)          " insert new value at start of list
call insert(list, newval, idx)     " insert new value before index idx
call    add(list, newval)          " append new value to end of list
```

You can insert a list of values with `extend()`:

### Listing 12. Adding a set of values to a list

```
call extend(list, newvals)         " append new values to end of list
call extend(list, newvals, idx)    " insert new values before index idx
```

Or remove specified elements from a list:

### Listing 13. Removing elements

```
call remove(list, idx)             " remove element at index idx
call remove(list, from, to)        " remove elements in range of indices
```

Or sort or reverse a list:

### Listing 14. Sorting or reversing a list

```
call sort(list)                    " re-order the elements of list alphabetically

call reverse(list)                 " reverse order of elements in list
```

### A common mistake with list procedures

Note that all list-related procedures also return the list they've just modified, so you could write:

```
let sorted_list = reverse(sort(unsorted_list))
```

Doing so would almost always be a serious mistake, however, because even when their return values are used in this way, list-related functions still modify their original argument. So, in the previous example, the list in `unsorted_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 aliasing").

This is highly counterintuitive for most programmers, who typically expect functions like `sort` and `reverse` to return modified copies of the original data, without changing the original itself.

Vimscript lists simply don't work that way, so it's important to cultivate good coding habits that will help you avoid nasty surprises. One such habit is to only ever call `sort()`, `reverse()`, and the like, as pure functions, and to always pass a copy of the data to be modified. You can use the built-in `copy()` function for this purpose:

```
let sorted_list = reverse(sort(copy(unsorted_list)))
```

## Filtering and transforming lists

Two particularly useful procedural list functions are `filter()` and `map()`. The `filter()` 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 to `filter()` converts the string that is passed as its second argument to a piece of code, which it then applies to each element of the list that is passed as its first argument. In other words, it repeatedly performs an `eval()` on its second argument. For each evaluation, it passes the next element of its first argument to the code, via the special variable *v:val*. If the result of the evaluated 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 pattern `/.*nix/`, type:

```
let non_starnix = filter(copy(list_of_systems), 'v:val !~ ".*nix"')
```

## The map() function

The `map()` function is similar to `filter()`, except that instead of removing some elements, it replaces every element with a user-specified transformation of its original value. The syntax is:

```
let transformed_list = map(copy(list), transformation_as_str)
```

Like `filter()`, `map()` evaluates the string passed as its second argument, passing each list element in turn, via `v:val`. But, unlike `filter()`, a `map()` 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 that `filter()` 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 the + and += 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 think of += as "`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 brackets of an indexing operation. The limits of the range can be constants, variables with numeric values, or any 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 ending index, the sublist finishes at the last element. For example, to split a list into two (near-)equal halves, 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

The full power and utility of lists is best illustrated by example. Let's start by improving an existing tool.

The second article in this series explored a user-defined function called `AlignAssignments()`, which lined up assignment operators in elegant columns. Listing 19 reproduces that function.

### 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: once (in the first `for` loop) to gather information on the paragraph's existing structure, and a second time (in the final `for` 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 some internal data structure and reuse them directly. Knowing what you do about lists, it is indeed possible to rewrite `AlignAssignments()` more efficiently and more cleanly. Listing 20 shows a new version of

the function that takes advantage of several list data structures and the various list-manipulation 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 in the original. As before, they locate the range of lines whose assignments are to be aligned, based on the current indentation of the text.

The changes begin in the third code block, which uses the two-argument form of the built-in `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 in `ASSIGN_LINE` using the built-in `matchlist()` function:

```
let fields = matchlist(linetext, ASSIGN_LINE)
```

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

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

- The full line (because `matchlist()` *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 call to `add()` adds a sublist of the final three fields to the lines list. If the match failed (that is, the line didn't contain an assignment), then `matchlist()` will return an empty list, so the sublist that `add()` appends (`fields[1:3]` below) will also be empty. This will be used to indicate a line of no further interest to the reformatter:

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

The fourth code block deploys the `filter()` and `map()` functions to analyze the structure of each line containing an assignment. It first uses a `filter()` to winnow the list of lines, keeping only those that were successfully decomposed into multiple components by the previous code block:

```
let op_lines = filter(copy(lines), '!empty(v:val)')
```

Next the function determines the length of each assignment's `lvalue`, by mapping the `strlen()` function over a copy of the filtered lines:

```
map(copy(op_lines), 'strlen(v:val[0])')
```

The resulting list of `lvalue` lengths is then passed to the built-in `max()` function to determine the longest `lvalue` in any assignment. The maximal length determines the column at which all the assignment operators will need to be aligned (that is, one column beyond the widest `lvalue`):

```
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 columns required to accommodate the various assignment operators that were found, by mapping and then 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 original buffer line numbers (`linenum`) 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 is, whether it was decomposed successfully around an assignment operation). If so, the function creates a new version of the line, using a `printf()` to reformat the line's components:

```
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 buffer by calling `setline()`, 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 all the relevant assignment operators aligned to a suitable column. Because it can take advantage of Vimscript's excellent support for lists and list operations, the code for this second version of `AlignAssignments()` is about 15 percent shorter than that of the previous version. Far more importantly, however, the function does only one-third as many buffer accesses, and the code is much clearer and more maintainable.

## Example 2: Enhancing Vim’s completion facilities

Vim has a sophisticated built-in text-completion mechanism, which you can learn about by typing `:help ins-completion` in any Vim session.

One of the most commonly used completion modes is *keyword completion*. You can use it any time you’re inserting text, by pressing **CTRL-N**. When you do, it searches various locations (as specified by the "`complete`" option), looking for words that start with whatever sequence of characters immediately precedes the cursor. By default, it looks in the current buffer you’re editing, any other buffers you’ve edited in the same session, any tag files you’ve loaded, and any files that are included from your text (via the `include` option).

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

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

Vim would search the text and determine that the only word beginning with *"so..."* was *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 of **CTRL-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 type **CTRL-E**; to accept and insert the currently selected alternative, you can type **CTRL-Y**. Typing anything else (typically, a space or newline) also accepts and inserts the currently selected word, as well as whatever extra character you typed.

## Designing smarter completions

There's no doubt that Vim's built-in completion mechanism is extremely useful, but it's not very clever. 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 the left of the cursor.

The completion mechanism is also not very ergonomic. **CTRL-N** isn't the easiest sequence to type, nor is it the one a programmer's fingers are particularly used to typing. Most command-line users are more accustomed to using **TAB** or **ESC** as their completion key.

Happily, with Vimscript, we can easily remedy those deficiencies. Let's redefine the **TAB** key in insertion mode so that it can be taught to recognize patterns in the text on either side of the cursor and select an appropriate completion for that context. We'll also arrange it so that, if the new mechanism doesn't recognize the current insertion context, it will fall back to Vim's built-in **CTRL-N** completion mechanism. Oh, and while we're at it, we should probably make sure we can still use the **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 responses" to a completion request. So we'll need a list. Or rather, a list of lists, given each contextual response will itself consist of four elements. Listing 22 shows how to set up that data structure.

## 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    )
```

```
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, and will be stored in the list variable `s:completions`. Each entry in the list will itself be a list, with four values:

- 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 position, after the completion text has been inserted

To populate the table, we create a small function: `AddCompletion()`. This function expects four arguments: the left and right contexts, and the replacement text, and the "`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 the start of the `s:completions` variable using the built-in `insert()` function:

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

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

The first call to `AddCompletion()`:

```
"                   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 and nothing to the right, it should insert a closing curly brace and then restore the cursor to its pre-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.

The second call to `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 to the right of the cursor, it should insert a newline, outdent the closing curly (via a **CTRL-D**), then escape from insertion mode (**ESC**) and open a new line above the closing curly (`O`).

Assuming the "`smartindent`" option is enabled, the net effect of the sequence is that, when you 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 table, **TAB**-completion after an opening brace closes it on the same line, and then immediately doing a second **TAB**-completion "stretches" the block across several lines (with correct indenting).

The remaining calls to `AddCompletion()` replicate this arrangement for the three other kinds of brackets (square, round, and angle) and also provide special completion semantics for single- and double-quotes. Completing after a double-quote appends the matching double-quote, while completing between two double quotes appends a `\n` (newline) metacharacter. Completing after a single quote appends the matching single quote, and then a second completion attempt does nothing.

## Implementing smarter completions

Once the list of completion-specifications has been set up, all that remains is to implement a function to select the appropriate completion from the table, and then bind that function to the **TAB** 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>"
```

```
    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 the built-in `getpos()` function with a `'.'` argument (that is, "get position of cursor"). That call returns a list of four elements: the buffer number (usually zero), the row and column numbers (both indexed from 1), and a special "virtual offset" (which is also usually zero, and not relevant here). We're primarily interested in the middle two values, as they indicate the location of the cursor. In particular, `SmartComplete()` needs the column number, which is extracted by indexing into the list that `getpos()` returned, like so:

```
let cursorcol = cursorpos[2]
```

The function also needs to know the text on the current line, which can be retrieved using `getline()`, and is stored in `curr_line`.

`SmartComplete()` is going to convert each entry in the `s:completions` table into a pattern to be matched against the current line. In order to correctly match left and right contexts around the cursor, it needs to ensure the pattern matches only at the cursor's column. Vim has a special subpattern for that: `\%Nc` (where `N` is the column number required). So, the function creates that 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 to the **TAB** key, we'd like the function to still insert a **TAB** whenever possible, and especially at the start of a line. So `SmartComplete()` first checks if there is only whitespace to the left of the cursor position, in which case it returns a simple tabspace:

```
if curr_line =~ '^\s*' . curr_pos_pat
    return "\<TAB>"
endif
```

If the cursor isn't at the start of a line, then `SmartComplete()` needs to check all the entries in the completion table and determine which, if any, apply. Some of those entries will specify that the cursor should be returned to its previous position after completion, which will require executing a custom command from within insertion mode. That command is simply a call to the built-in `setpos()` function, passing the value the original information from the earlier call to `getpos()`. To execute that function call from within insertion mode requires a **CTRL-O** escape (see `:help i_CTRL-O` in any Vim session). So `SmartComplete()` prebuilds the necessary **CTRL-O** command as a string and stores in `cursor_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 version of the `for` statement. The standard `for` loop in Vimscript walks through a one-dimensional list, one element at a time:

## 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 you often want to "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 nested list from `list_of_lists` and assigns the first element of that nested list to `name`, the second nested element to `rank`, and the third to `serial`.

Using this special form of `for` loop makes it easy for `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
```

## Recognizing a completion context

Within the loop, `SmartComplete()` constructs a regular expression by placing the left and right 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 completion (the text of which is already in completion) and can return it immediately. Of course, it also needs to append the cursor restoration command it built earlier, if the selected completion has requested it (that is, if `restore` is true).

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

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-line

```
if cursorcol == strlen(curr_line)+1 && strlen(completion)==1
    let cursor_back = "\<LEFT>"
endif
```

All that remains is to return the selected completion, appending the `cursor_back` command if cursor restoration was requested:

```
return completion . (restore ? cursor_back : "")
```

If none of the entries from the completion table match the current context, `SmartComplete()` will eventually fall out of the `for` loop and will then try two final alternatives. If the character immediately before the cursor was a "keyword" character, it invokes a normal keyword-completion by returning a **CTRL-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 like a normal **TAB** key, by 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 call `SmartComplete()` in order to work out what it should insert. That's done with an `inoremap`, like so:

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

The key-mapping converts any insert-mode **TAB** to a **CTRL-R=**, calling `SmartComplete()` and inserting the completion string it returns (see `:help i_CTRL-R` or the first article in this series for details of this mechanism).

The `inoremap` form of `imap` is used here because some of the completion strings that `SmartComplete()` returns also contain a **TAB** character. If a regular `imap` were 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 the `inoremap` in place, we now have a **TAB** 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 a **TAB** everywhere else

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

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

so that tabbing immediately after a function keyword appends the corresponding `endfunction` keyword on the next line.

Or, you could autocomplete C/C++ comments intelligently (assuming the `cindent` option 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 cursor in the middle of it:

```
/*
 * _
 */
```

## Looking ahead

The ability to store and manipulate lists of data greatly increases the range of tasks that Vimscript can easily accomplish, but lists are not always the ideal solution for aggregating and storing collections of information. For example, the re-implemented version of `AlignAssignments()` shown in Listing 20 contains a `printf()` call that looks like this:

```
printf("%-*s%*s%s", max_lval, line[0], max_op, line[1], line[2])
```

Using `line[0]`, `line[1]`, and `line[2]` for the various components of a code line is certainly not very readable, and hence both error-prone during initial implementation, and unnecessarily hard to maintain thereafter.

This is a common situation: related data needs to be collected together, but has no inherent or meaningful order. In such cases, each datum is often better identified by some logical name, rather than by a numeric index. Of course, we could always create a set of variables to "name" the respective numeric constants:

```
let LVAL = 0
let OP   = 1
let RVAL = 2

" and later...

printf("%-*s%*s%s", max_lval, line[LVAL], max_op, line[OP], line[RVAL])
```

But that’s a clunky and brittle solution, prone to hard-to-find errors if the order of components were to change within the line list, but the variables weren’t updated appropriately.

Because collections of named data are such a common requirement in programming, in most dynamic languages there’s a common construct that provides them: the *associative array*, or *hash table*, or *dictionary*. As it turns out, Vim has dictionaries too. In the next article in this series, we’ll look at Vimscript’s implementation of that very useful data structure.

## Related topics

- Start learning about Vimscript, the embedded language for extending the Vim editor, with the first article in this series: "Scripting the Vim editor, Part 1: Variables, values, and expressions" (developerWorks, May 2009).
- "Scripting the Vim editor, Part 2: User-defined functions" (developerWorks, July 2009) tours Vimscript's scalar data types: strings, numbers, and booleans.
- See the following resources to continue learning about the Vim editor and its many commands:
  - The Vim homepage
  - The online book A Byte of Vim
  - Various hardcopy books on Vim
  - Vim's own manual
  - Steve Oualline's Vim Cookbook
- For more extensive examples of Vim scripting, see:
  - The Vim Tips wiki
  - The Vimscript archive
- Start at the Vim distributions downloads page to upgrade to the latest version of Vim for your platform.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.