Scripting the Vim editor, Part 2: User-defined functions

Create the fundamental building blocks of automation

Damian Conway, Dr.
CEO and Chief Trainer
Thoughtstream

07 July 2009

User-defined functions are an essential tool for decomposing an application into correct and maintainable components, in order to manage the complexity of real-world programming tasks. This article (the second in a series) explains how to create and deploy new functions in the Vimscript language, giving several practical examples of why you might want to.

View more content in this series

About Vimscript and this series

Vimscript is a powerful scripting language that lets you modify and extend the Vim editor. You can use it to create new tools, simplify common tasks, and even rework existing features of the editor. This ongoing series of articles assumes some familiarity with the Vim editor. You should also read "Scripting the Vim editor, Part 1," which covers variables, values, and expressions, prerequisite knowledge for building functions.

User-defined functions

Ask Haskell or Scheme programmers, and they'll tell you that functions are the most important feature of any serious programming language. Ask C or Perl programmers, and they'll tell you exactly the same thing.

Functions provide two essential benefits to the serious programmer:

1. They enable complex computational tasks to be subdivided into pieces small enough to fit comfortably into a single human brain.
2. They allow those subdivided pieces to be given logical and comprehensible names, so they can be competently manipulated by a single human brain.

Vimscript is a serious programming language, so it naturally supports the creation of user-defined functions. Indeed, it arguably has *better* support for user-defined functions than Scheme, C, or

Perl. This article explores the various features of Vimscript functions, and show how you can use those features to enhance and extend Vim's built-in functionality in a maintainable way.

Declaring functions

Functions in Vimscript are defined using the `function` keyword, followed by the name of the function, then the list of parameters (which is mandatory, even if the function takes no arguments). The body of the function then starts on the next line, and continues until a matching `endfunction` keyword is encountered. For example:

Listing 1. A correctly structured function

```
functionExpurgateText (text)
    let expurgated_text = a:text

    for expletive in [ 'cagal', 'frak', 'gorram', 'mebs', 'zarking']
        let expurgated_text
        \   = substitute(expurgated_text, expletive, '[DELETED]', 'g')
    endfor

    return expurgated_text
endfunction
```

The return value of the function is specified with a `return` statement. You can specify as many separate `return` statements as you need. You can include none at all if the function is being used as a procedure and has no useful return value. However, Vimscript functions *always* return a value, so if no `return` is specified, the function automatically returns zero.

Function names in Vimscript must start with an uppercase letter:

Listing 2. Function names start with an uppercase letter

```
function SaveBackup ()
    let b:backup_count = exists('b:backup_count') ? b:backup_count+1 : 1
    return writefile(getline(1,'$'), bufname('%') . '_' . b:backup_count)
endfunction

nmap <silent>  <C-B>  :call SaveBackup()<CR>
```

This example defines a function that increments the value of the current buffer's `b:backup_count` variable (or initializes it to 1, if it doesn't yet exist). The function then grabs every line in the current file (`getline(1,'$')`) and calls the built-in `writefile()` function to write them to disk. The second argument to `writefile()` is the name of the new file to be written; in this case, the name of the current file (`bufname('%')`) with the counter's new value appended. The value returned is the success/failure value of the call to `writefile()`. Finally, the `nmap` sets up CTRL-B to call the function to create a numbered backup of the current file.

Instead of using a leading capital letter, Vimscript functions can also be declared with an explicit scope prefix (like variables can be, as described in Part 1). The most common choice is `s:`, which makes the function local to the current script file. If a function is scoped in this way, its name need not start with a capital; it can be any valid identifier. However, explicitly scoped functions must always be called with their scoping prefixes. For example:

Listing 3. Calling a function with its scoping prefix

```
" Function scoped to current script file...
function s:save_backup ()
    let b:backup_count = exists('b:backup_count') ? b:backup_count+1 : 1
    return writefile(getline(1,'$'), bufname('%') . '_' . b:backup_count)
endfunction

nmap <silent>  <C-B>  :call s:save_backup()<CR>
```

Redeclarable functions

Function declarations in Vimscript are runtime statements, so if a script is loaded twice, any function declarations in that script will be executed twice, re-creating the corresponding functions.

Redeclaring a function is treated as a fatal error (to prevent *collisions* where two separate scripts accidentally declare functions of the same name). This makes it difficult to create functions in scripts that are designed to be loaded repeatedly, such as custom syntax-highlighting scripts.

So Vimscript provides a keyword modifier (`function!`) that allows you to indicate that a function declaration may be safely reloaded as often as required:

Listing 4. Indicating that a function declaration may be safely reloaded

```
function! s:save_backup ()
    let b:backup_count = exists('b:backup_count') ? b:backup_count+1 : 1
    return writefile(getline(1,'$'), bufname('%') . '_' . b:backup_count)
endfunction
```

No redeclaration checks are performed on functions defined with this modified keyword, so it is best used with explicitly scoped functions (in which case the scoping already ensures that the function won't collide with one from another script).

Calling functions

To call a function and use its return value as part of a larger expression, simply name it and append a parenthesized argument list:

Listing 5. Using a function's return value

```
"Clean up the current line...
let success = setline('.', ExpurgateText(getline('.')) )
```

Note, however, that, unlike C or Perl, Vimscript does *not* allow you to throw away the return value of a function without using it. So, if you intend to use the function as a procedure or subroutine and ignore its return value, you must prefix the invocation with the `call` command:

Listing 6. Using a function without using its return value

```
"Checkpoint the text...
call SaveBackup()
```

Otherwise, Vimscript will assume that the function call is actually a built-in Vim command and will most likely complain that no such command exists. We'll look at the difference between functions and commands in a future article in this series.

Parameter lists

Vimscript allows you to define both *explicit parameters* and *variadic parameter lists*, and even combinations of the two.

You can specify up to 20 explicitly named parameters immediately after the declaration of the subroutine's name. Once specified, the corresponding argument values for the current call can be accessed within the function by prefixing an `a:` to the parameter name:

Listing 7. Accessing argument values within the function

```
function PrintDetails(name, title, email)
    echo 'Name:   '  a:title  a:name
    echo 'Contact:'  a:email
endfunction
```

If you don't know how many arguments a function may be given, you can specify a variadic parameter list, using an ellipsis (`...`) instead of named parameters. In this case, the function may be called with as many arguments as you wish, and those values are collected into a single variable: an array named `a:000`. Individual arguments are also given positional parameter names: `a:1`, `a:2`, `a:3`, etc. The number of arguments is available as `a:0`. For example:

Listing 8. Specifying and using a variadic parameter list

```
function Average(...)
    let sum = 0.0

    for nextval in a:000"a:000 is the list of arguments
        let sum += nextval
    endfor

    return sum / a:0"a:0 is the number of arguments
endfunction
```

Note that, in this example, `sum` must be initialized to an explicit floating-point value; otherwise, all the subsequent computations will be done using integer arithmetic.

Combining named and variadic parameters

Named and variadic parameters can be used in the same function, simply by placing the variadic ellipsis *after* the list of named parameters.

For example, suppose you wanted to create a `CommentBlock()` function that was passed a string and formatted it into an appropriate comment block for various programming languages. Such a function would always require the caller to supply the string to be formatted, so that parameter should be explicitly named. But you might prefer that the comment introducer, the "boxing" character, and the width of the comment all be optional (with sensible defaults when omitted). Then you could call:

Listing 9. A simple CommentBlock function call

```
call CommentBlock("This is a comment")
```

and it would return a multi-line string containing:

Listing 10. The CommentBlock return

```
//******************
// This is a comment
//******************
```

Whereas, if you provided extra arguments, they would specify non-default values for the comment introducer, the "boxing" character, and the comment width. So this call:

Listing 11. A more involved CommentBlock function call

```
call CommentBlock("This is a comment", '#', '=', 40)
```

would return the string:

Listing 12. The CommentBlock return

```
#======================================
# This is a comment
#======================================
```

Such a function might be implemented like so:

Listing 13. The CommentBlock implementation

```
function CommentBlock(comment, ...)
    "If 1 or more optional args, first optional arg is introducer...
    let introducer =  a:0 >= 1  ?  a:1  :  "//"

    "If 2 or more optional args, second optional arg is boxing character...
    let box_char   =  a:0 >= 2  ?  a:2  :  "*"

    "If 3 or more optional args, third optional arg is comment width...
    let width      =  a:0 >= 3  ?  a:3  :  strlen(a:comment) + 2

    " Build the comment box and put the comment inside it...
    return introducer . repeat(box_char,width) . "\<CR>"
    \    . introducer . " " . a:comment        . "\<CR>"
    \    . introducer . repeat(box_char,width) . "\<CR>"
endfunction
```

If there is at least one optional argument (`a:0 >= 1`), the introducer parameter is assigned that first option (that is, *`a:1`*); otherwise, it is assigned a default value of `"//"`. Likewise, if there are two or more optional arguments (`a:0 >= 2`), the `box_char` variable is assigned the second option (*`a:2`*), or else a default value of `"*"`. If three or more optional arguments are supplied, the third option is assigned to the *`width`* variable. If no width argument is given, the appropriate width is autocomputed from the comment argument itself (`strlen(a:comment)+2`).

Finally, having resolved all the parameter values, the top and bottom lines of the comment box are constructed using the leading comment introducer, followed by the appropriate number of repetitions of the boxing character (`repeat(box_char,width)`), with the comment text itself sandwiched between them.

Of course, to use this function, you'd need to invoke it somehow. An insertion map is probably the ideal way to do that:

Listing 14. Invoking the function using an insertion map

```
"C++/Java/PHP comment...
imap <silent>  ///  <C-R>=CommentBlock(input("Enter comment: "))<CR>

"Ada/Applescript/Eiffel comment...
imap <silent>  ---  <C-R>=CommentBlock(input("Enter comment: "),'--')<CR>

"Perl/Python/Shell comment...
imap <silent>  ###  <C-R>=CommentBlock(input("Enter comment: "),'#','#')<CR>
```

In each of these maps, the built-in `input()` function is first called to request that the user type in the text of the comment. The `CommentBlock()` function is then called to convert that text into a comment block. Finally, the leading `<C-R>=` inserts the resulting string.

Note that the first mapping passes only a single argument, so it defaults to using `//` as its comment marker. The second and third mappings pass a second argument to specify `#` or `--` as their respective comment introducers. The final mapping also passes a third argument, to make the "boxing" character match its comment introducer.

Functions and line ranges

You can invoke any standard Vim command—including `call`—with a preliminary line range, which causes the command to be repeated once for every line in the range:

```
"Delete every line from the current line (.) to the end-of-file ($)...
:.,$delete
"Replace "foo" with "bar" everywhere in lines 1 to 10
:1,10s/foo/bar/
"Center every line from five above the current line to five below it...
:-5,+5center
```

You can type `:help cmdline-ranges` in any Vim session to learn more about this facility.

In the case of the `call` command, specifying a range causes the requested function to be called repeatedly: once for each line in the range. To see why that's useful, let's consider how to write a function that converts any "raw" ampersands in the current line to proper XML `&` entities, but that is also smart enough to ignore any ampersand that is already part of some other entity. That function could be implemented like so:

Listing 15. Function to convert ampersands

```
function DeAmperfy()
    "Get current line...
    let curr_line   = getline('.')

    "Replace raw ampersands...
    let replacement = substitute(curr_line,'&\(\w\+;\)\@!','&amp;','g')

    "Update current line...
    call setline('.', replacement)
endfunction
```

The first line of `DeAmperfy()` grabs the current line from the editor buffer (`getline('.')`). The second line looks for any `&` in that line that *isn't* followed by an identifier and a colon, using the negative lookahead pattern `'&\(\w\+;\)\@!'`(see `:help \@!` for details). The `substitute()` call then replaces all such "raw" ampersands with the XML entity `&`. Finally, the third line of `DeAmperfy()`updates the current line with the modified text.

If you called this function from the command line:

```
:call DeAmperfy()
```

it would perform the replacement on the current line only. But if you specified a range before the `call`:

```
:1,$call DeAmperfy()
```

then the function would be called once for each line in the range (in this case, for every line in the file).

Internalizing function line ranges

This *call-the-function-repeatedly-for-each-line* behavior is a convenient default. However, sometimes you might prefer to specify a range but then have the function called only once, and then handle the range semantics within the function itself. That's also easy in Vimscript. You simply append a special modifier (`range`) to the function declaration:

Listing 16. Range semantics within a function

```
function DeAmperfyAll() range"Step through each line in the range...
    for linenum in range(a:firstline, a:lastline)
        "Replace loose ampersands (as in DeAmperfy())...
        let curr_line   = getline(linenum)
        let replacement = substitute(curr_line,'&\(\w\+;\)\@!','&amp;','g')
        call setline(linenum, replacement)
    endfor

    "Report what was done...
    if a:lastline > a:firstline
        echo "DeAmperfied" (a:lastline - a:firstline + 1) "lines"
    endif
endfunction
```

With the `range` modifier specified after the parameter list, any time `DeAmperfyAll()` is called with a range such as:

```
:1,$call DeAmperfyAll()
```

then the function is invoked only once, and two special arguments, `a:firstline` and `a:lastline`, are set to the first and last line numbers in the range. If no range is specified, both `a:firstline` and `a:lastline` are set to the current line number.

The function first builds a list of all the relevant line numbers (`range(a:firstline, a:lastline)`). Note that this call to the built-in `range()` function is entirely unrelated to the use of the `range`

modifier as part of the function declaration. The `range()` function is simply a list constructor, very similar to the `range()` function in Python, or the `..` operator in Haskell or Perl.

Having determined the list of line numbers to be processed, the function uses a `for` loop to step through each:

```
for linenum in range(a:firstline, a:lastline)
```

and updates each line accordingly (just as the original `DeAmperfy()` did).

Finally, if the range covers more than a single line (in other words, if `a:lastline > a:firstline`), the function reports how many lines were updated.

Visual ranges

Once you have a function call that can operate on a range of lines, a particularly useful technique is to call that function via Visual mode (see `:help Visual-mode` for details).

For example, if your cursor is somewhere in a block of text, you could encode all the ampersands anywhere in the surrounding paragraph with:

```
Vip:call DeAmperfyAll()
```

Typing `v` in Normal mode swaps you into Visual mode. The `ip` then causes Visual mode to highlight the entire paragraph you're inside. Then, the `:` swaps you to Command mode and automatically sets the command's range to the range of lines you just selected in Visual mode. At this point you call `DeAmperfyAll()` to deamperfy all of them.

Note that, in this instance, you could get the same effect with just:

```
Vip:call DeAmperfy()
```

The only difference is that the `DeAmperfy()` function would be called repeatedly: once for each line the `Vip` highlighted in Visual mode.

A function to help you code

Most user-defined functions in Vimscript require very few parameters, and often none at all. That's because they usually get their data directly from the current editor buffer and from contextual information (such as the current cursor position, the current paragraph size, the current window size, or the contents of the current line).

Moreover, functions are often far more useful and convenient when they obtain their data through context, rather than through their argument lists. For example, a common problem when maintaining source code is that assignment operators fall out of alignment as they accumulate, which reduces the readability of the code:

Listing 17. Assignment operators out of alignment

```
let applicants_name = 'Luke'
let mothers_maiden_name = 'Amidala'
let closest_relative = 'sister'
let fathers_occupation = 'Sith'
```

Realigning them manually every time a new statement is added can be tedious:

Listing 18. Manually realigned assignment operators

```
let applicants_name     = 'Luke'
let mothers_maiden_name = 'Amidala'
let closest_relative    = 'sister'
let fathers_occupation  = 'Sith'
```

To reduce the tedium of that everyday coding task, you could create a key-mapping (such as `;=`) that selects the current block of code, locates any lines with assignment operators, and automatically aligns those operators. Like so:

Listing 19. Function to align assignment operators

```
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

nmap <silent>  ;=  :call AlignAssignments()<CR>
```

The `AlignAssignments()` function first sets up two regular expressions (see `:help pattern` for the necessary details of Vim's regex syntax):

```
let ASSIGN_OP   = '[-+*/%|&]\?=\@<!=[=~]\@!'
let ASSIGN_LINE = '^\(.\{-}\)\s*\(' . ASSIGN_OP . '\)'
```

The pattern in *`ASSIGN_OP`* matches any of the standard assignment operators: `=`, `+=`, `-=`, `*=`, etc. but carefully avoids matching other operators that contain `=`, such as `==` and `=~`. If your favorite language has other assignment operators (such as `.=` or `||=` or `^=`), you could extend the *`ASSIGN_OP`* regex to recognize those as well. Alternatively, you could redefine *`ASSIGN_OP`* to recognize other types of "alignables," such as comment introducers or column markers, and align them instead.

The pattern in *`ASSIGN_LINE`* matches only at the start of a line (`^`), matching a minimal number of characters (`.\{-}`), then any whitespace (`\s*`), then an assignment operator.

Note that both the initial "minimal number of characters" subpattern and the operator subpattern are specified within capturing parentheses: `\(...\)`. The substrings captured by those two components of the regex will later be extracted using calls to the built-in `submatch()` function; specifically, by calling `submatch(1)` to extract everything before the operator, and `submatch(2)` to extract the operator itself.

`AlignAssignments()` then locates the range of lines on which it will operate:

```
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
```

In earlier examples, functions have relied on an explicit command range or a Visual mode selection to determine which lines they operate on, but this function computes its own range directly. Specifically, it first calls the built-in `matchstr()` function to determine what leading whitespace (`'^\s*'`) appears at the start of the current line (`getline('.')`. It then builds a new regular expression in *`indent_pat`* that matches exactly the same sequence of whitespace at the start of any non-empty line (hence the trailing `'\S'`).

`AlignAssignments()` then calls the built-in `search()` function to search upwards (using the flags `'bnW'`) and locate the first line above the cursor that does *not* have precisely the same indentation. Adding 1 to this line number gives the start of the range of interest, namely, the first contiguous line with the same indentation as the current line.

A second call to `search()` then searches downwards (`'nW'`) to determine *`lastline`*: the number of the final contiguous line with the same indentation. In this second case, the search might hit the end of the file without finding a differently indented line, in which case `search()` would return `-1`. To handle this case correctly, the following `if` statement would explicitly set *`lastline`* to the line number of the end of file (that is, to the line number returned by `line('$')`).

The result of these two searches is that `AlignAssignments()` now knows the full range of lines immediately above or below the current line that all have precisely the same indentation as the current line. It uses this information to ensure that it aligns only those assignment statements at the same scoping level in the same block of code. Unless, of course, the indentation of your code doesn't correctly reflect its scope, in which case you fully deserve the formatting catastrophe about to befall you.

The first `for` loop in `AlignAssignments()` determines the column in which the assignment operators should be aligned. This is done by walking through the list of lines in the selected range (the lines retrieved by `getline(firstline, lastline)`) and checking whether each line contains an assignment operator (possibly preceded by whitespace):

```
let left_width = match(linetext, '\s*' . ASSIGN_OP)
```

If there is no operator in the line, the built-in `match()` function will fail to find a match and will return `-1`. In that case, the loop simply skips on to the next line. If there *is* an operator, `match()` will return the (positive) index at which that operator appears. The `if` statement then uses the built-in `max()` function to determine whether this latest column position is further right than any previously located operator, thereby tracking the maximum column position required to align all the assignments in the range:

```
let max_align_col = max([max_align_col, left_width])
```

The remaining two lines of the `if` use the built-in `matchstr()` function to retrieve the actual operator, then the built-in `strlen()` to determine its length (which will be 1 for a `"="` but 2 for `'+='`, `'-='`, etc.) The *`max_op_width`* variable is then used to track the maximum width required to align the various operators in the range:

```
let op_width     = strlen(matchstr(linetext, ASSIGN_OP))
let max_op_width = max([max_op_width, op_width+1])
```

Once the location and width of the alignment zone have been determined, all that remains is to iterate through the lines in the range and reformat them accordingly. To do that reformatting, the function uses the built-in `printf()` function. This function is very useful, but also very badly named. It is *not* the same as the `printf` function in C or Perl or PHP. It is, in fact, the same as the `sprintf` function in those languages. That is, in Vimscript, `printf` doesn't print a formatted version of its list of data arguments; it returns a *string* containing a formatted version of its list of data arguments.

Ideally, in order to reformat each line, `AlignAssignments()` would use the built-in `substitute()` function, and replace everything up to the operator with a `printf`'d rearrangement of that text. Unfortunately, `substitute()` expects a fixed string as its replacement value, not a function call.

So, in order to use a `printf()` to reformat each replacement text, you need to use the special embedded replacement form: `"\=`*`expr`*`"`. The leading `\=` in the replacement string tells `substitute()` to evaluate the expression that follows and use the result as the replacement text. Note that this is similar to the `<C-R>=` mechanism in Insert mode, except this magic behavior only

works for the replacement string of the built-in `substitute()` function (or in the standard `:s/.../.../` Vim command).

In this example, the special replacement form will be the same `printf` for every line, so it is pre-stored in the *`FORMATTER`* variable before the second `for` loop begins:

```
let FORMATTER = '\=printf("%-*s%*s", max_align_col, submatch(1),
\                                   max_op_width,  submatch(2))'
```

When it is eventually called by `substitute()`, this embedded `printf()` will left-justify (using a `%-*s` placeholder) everything to the left of the operator (`submatch(1)`) and place the result in a field that's *`max_align_col`* characters wide. It will then right-justify (using a `%*s`) the operator itself (`submatch(2)`) into a second field that's *`max_op_width`* characters wide. See `:help printf()` for details on how the `-` and `*` options modify the two `%s` format specifiers used here.

With this formatter now available, the second `for` loop can finally iterate through the full range of line numbers, retrieving the corresponding text buffer contents one line at a time:

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

The loop then uses `substitute()` to transform those contents, by matching everything up to and including any assignment operator (using the pattern in *`ASSIGN_LINE`*) and replacing that text with the result of the `printf()` call (as specified by *`FORMATTER`*):

```
    let newline = substitute(oldline, ASSIGN_LINE, FORMATTER, "")
    call setline(linenum, newline)
endfor
```

Once the `for` loop has iterated all the lines, any assignment operators within them will now be aligned correctly. All that remains is to create a key-mapping to invoke `AlignAssignments()`, like so:

```
nmap <silent>  ;=  :call AlignAssignments()<CR>
```

Looking ahead

Functions are an essential tool for decomposing an application into correct and maintainable components, in order to manage the complexity of real-world *Vim* programming tasks.

Vimscript allows you to define functions with fixed or variadic parameter lists, and to have them interact either automatically or in user-controlled ways with ranges of lines in the editor's text buffer. Functions can call back to Vim's built-in features (for example, to `search()` or `substitute()` text), and they can also directly access editor state information (such as determining the current line the cursor is on via `line('.')`) or interact with any text buffer currently being edited (via `getline()` and `setline()`).

This is undoubtedly a powerful facility, but our ability to programmatically manipulate state and content is always limited by how cleanly and accurately we can represent the data on which our

code operates. So far in this series of articles, we've been restricted to the use of single scalar values (numbers, strings, and booleans). In the next two articles, we'll explore the use of much more powerful and convenient data structures: ordered lists and random-access dictionaries.

Resources

Learn

- Start with "Scripting the Vim editor, Part 1: Variables, values, and expressions" (developerWorks, May 2009) to learn Vimscript, the embedded language for extending the Vim editor. Create new tools, simplify common tasks, and redesign and replace existing editor features.
- To learn more about the Vim editor and its many commands, see:
  - 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
- 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.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Start at the Vim distributions downloads page to upgrade to the latest version of Vim for your platform.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the My developerWorks community; with your personal profile and custom home page, you can tailor developerWorks to your interests and interact with other developerWorks users.

About the author

Damian Conway, Dr.

Damian Conway is an Adjunct Associate Professor of Computer Science at Monash University, Australia, and CEO of Thoughtstream, an international IT training company. He has been a daily vi user for well over a quarter of a century, and there now seems very little hope he will ever conquer the addiction.