Scripting the Vim editor, Part 1: Variables, values, and expressions

Start with the basic elements of Vimscript

Damian Conway, Dr.

May 06, 2009

Vimscript is a mechanism for reshaping and extending the Vim editor. Scripting allows you to create new tools, simplify common tasks, and even redesign and replace existing editor features. This article (the first in a series) introduces the fundamental components of the Vimscript programming language: values, variables, expressions, statements, functions, and commands. These features are demonstrated and explained through a series of simple examples.

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.

A great text editor

There's an old joke that Emacs would be a great operating system if only it had a decent text editor, whereas vi would be a great text editor if only it had a decent operating system. This gag reflects the single greatest strategic advantage that Emacs has always had over vi: an embedded extension programming language. Indeed, the fact that Emacs users are happy to put up with RSI-inducing control chords and are willing to write their extensions in Lisp shows just how great an advantage a built-in extension language must be.

But vi programmers no longer need cast envious glances towards Emacs' parenthetical scripting language. Our favorite editor can be scripted too—and much more humanely than Emacs.

In this series of articles, we'll look at the most popular modern variant of vi, the Vim editor, and at the simple yet extremely powerful scripting language that Vim provides. This first article explores the basic building blocks of Vim scripting: variables, values, expressions, simple flow control, and a few of Vim's numerous utility functions.

I'll assume that you already have access to Vim and are familiar with its interactive features. If that's not the case, some good starting points are Vim's own Web site and various online resources and hardcopy books, or you can simply type `:help` inside Vim itself.

Unless otherwise indicated, all the examples in this series of articles assume you're using Vim version 7.2 or higher. You can check which version of Vim you're using by invoking the editor like so:

```
vim --version
```

or by typing `:version` within Vim itself. If you're using an older incarnation of Vim, upgrading to the latest release is strongly recommended, as previous versions do not support many of the features of Vimscript that we'll be exploring.

Vimscript

Vim's scripting language, known as Vimscript, is a typical dynamic imperative language and offers most of the usual language features: variables, expressions, control structures, built-in functions, user-defined functions, first-class strings, high-level data structures (lists and dictionaries), terminal and file I/O, regex pattern matching, exceptions, and an integrated debugger.

You can read Vim's own documentation of Vimscript via the built-in help system, by typing:

```
:help vim-script-intro
```

inside any Vim session. Or just read on.

Running Vim scripts

There are numerous ways to execute Vim scripting commands. The simplest approach is to put them in a file (typically with a .vim extension) and then execute the file by `:source`-ing it from within a Vim session:

```
:source /full/path/to/the/scriptfile.vim
```

Alternatively, you can type scripting commands directly on the Vim command line, after the colon. For example:

```
:call MyBackupFunc(expand('%'), { 'all':1, 'save':'recent'})
```

But very few people do that. After all, the whole point of scripting is to *reduce* the amount of typing you have to do. So the most common way to invoke Vim scripts is by creating new keyboard mappings, like so:

```
:nmap ;s :source /full/path/to/the/scriptfile.vim<CR>
:nmap \b :call MyBackupFunc(expand('%'), { 'all': 1 })<CR>
```

Commands like these are usually placed in the .vimrc initialization file in your home directory. Thereafter, when you're in Normal mode (in other words, not inserting text), the key sequence `;s` will execute the specified script file, and a `\b` sequence will call the `MyBackupFunc()` function (which you presumably defined somewhere in your .vimrc as well).

All of the Vimscript examples in this article use key mappings of various types as triggers. In later articles, we'll explore two other common invocation techniques: running scripts as colon commands from Vim's command line, and using editor events to trigger scripts automatically.

A syntactic example

Vim has very sophisticated syntax highlighting facilities, which you can turn on with the built-in `:syntax enable` command, and off again with `:syntax off`.

It's annoying to have to type ten or more characters every time you want to toggle syntax highlighting, though. Instead, you could place the following lines of Vimscript in your .vimrc file:

Listing 1. Toggling syntax highlighting

```
function! ToggleSyntax()
   if exists("g:syntax_on")
      syntax off
   else
      syntax enable
   endif
endfunction

nmap <silent>  ;s  :call ToggleSyntax()<CR>
```

This causes the `;s` sequence to flip syntax highlighting on or off each time it's typed when you're in Normal mode. Let's look at each component of that script.

The first block of code is obviously a function declaration, defining a function named `ToggleSyntax()`, which takes no arguments. That user-defined function first calls a built-in Vim function named `exists()`, passing it a string. The `exists()` function determines whether a variable with the name specified by the string (in this case, the global variable `g:syntax_on`) has been defined.

If so, the `if` statement executes a `syntax off`; otherwise it executes a `syntax enable`. Because `syntax enable` defines the `g:syntax_on` variable, and `syntax off` undefines it, calling the `ToggleSyntax()` function repeatedly alternates between enabling and disabling syntax highlighting.

All that remains is to set up a key sequence (`;s` in this example) to call the `ToggleSyntax()` function:

```
nmap <silent> ;s :call ToggleSyntax()<CR>
```

`nmap` stands for "**n**ormal-mode key **map**ping." The `<silent>` option after the `nmap` causes the mapping not to echo any command it's executing, ensuring that the new `;s` command will do its work unobtrusively. That work is to execute the command:

```
:call ToggleSyntax()<CR>
```

which is how you call a function in Vimscript when you intend to ignore the return value.

Note that the `<CR>` at the end is the literal sequence of characters `<`,`C`,`R`,`>`. Vimscript recognizes this as being equivalent to a literal carriage return. In fact, Vimscript understands many other similar representations of unprintable characters. For example, you could create a keyboard mapping to make your space bar act like the page-down key (as it does in most Web browsers), like so:

```
:nmap <Space> <PageDown>
```

You can see the complete list of these special symbols by typing `:help keycodes` within Vim.

Note too that `ToggleSyntax()` was able to call the built-in `syntax` command directly. That's because every built-in colon command in Vim is automatically also a statement in Vimscript. For example, to make it easier to create centered titles for documents written in Vim, you could create a function that capitalizes each word on the current line, centers the entire line, and then jumps to the next line, like so:

Listing 2. Creating centered titles

```
function! CapitalizeCenterAndMoveDown()
   s/\<./\u&/g   "Built-in substitution capitalizes each word
   center        "Built-in center command centers entire line
   +1            "Built-in relative motion (+1 line down)
endfunction

nmap <silent>  \C  :call CapitalizeCenterAndMoveDown()<CR>
```

Vimscript statements

As the previous examples illustrate, all statements in Vimscript are terminated by a newline (as in shell scripts or Python). If you need to run a statement across multiple lines, the continuation marker is a single backslash. Unusually, the backslash doesn't go at the end of the line to be continued, but rather at the start of the continuation line:

Listing 3. Continuing lines using backslash

```
call SetName(
\             first_name,
\             middle_initial,
\             family_name
\           )
```

You can also put two or more statements on a single line by separating them with a vertical bar:

```
echo "Starting..." | call Phase(1) | call Phase(2) | echo "Done"
```

That is, the vertical bar in Vimscript is equivalent to a semicolon in most other programming languages. Unfortunately, Vim couldn't use the semicolon, as that character already means something else at the start of a command (specifically, it means "from the current line to..." as part of the command's line range).

Comments

One important use of the vertical bar as a statement separator is in commenting. Vimscript comments start with a double-quote and continue to the end of the line, like so:

Listing 4. Commenting in Vimscript

```
if exists("g:syntax_on")
   syntax off
else
   syntax enable
endif
```

Unfortunately, Vimscript strings can also start with a double-quote and always take precedence over comments. This means you can't put a comment anywhere that a string might be expected, because it will always be interpreted as a string:

```
echo "> " "Print generic prompt
```

The echo command expects one or more strings, so this line produces an error complaining about the missing closing quote on (what Vim assumes to be) the second string.

Comments can, however, always appear at the very start of a statement, so you can fix the above problem by using a vertical bar to explicitly begin a new statement before starting the comment, like so:

```
echo "> " |"Print generic prompt
```

Values and variables

Variable assignment in Vimscript requires a special keyword, `let`:

Listing 5. Using the let keyword

```
let name = "Damian"

let height = 165

let interests = [ 'Cinema', 'Literature', 'World Domination', 101 ]

let phone     = { 'cell':5551017346, 'home':5558038728, 'work':'?' }
```

Note that strings can be specified with either double-quotes or single-quotes as delimiters. Double-quoted strings honor special "escape sequences" such as `"\n"` (for newline), `"\t"` (for tab), `"\u263A"` (for Unicode smiley face), or `"\<ESC>"` (for the escape character). In contrast, single-quoted strings treat everything inside their delimiters as literal characters—except two consecutive single-quotes, which are treated as a literal single-quote.

Values in Vimscript are typically one of the following three types:

- *scalar*: a single value, such as a string or a number. For example: `"Damian"` or `165`
- *list*: an ordered sequence of values delimited by square brackets, with implicit integer indices starting at zero. For example: `['Cinema', 'Literature', 'World Domination', 101]`

- *dictionary*: an unordered set of values delimited by braces, with explicit string keys. For example: `{'cell':5551017346, 'home':5558038728, 'work':'?'}`

Note that the values in a list or dictionary don't have to be all of the same type; you can mix strings, numbers, and even nested lists and dictionaries if you wish.

Unlike values, variables have no inherent type. Instead, they take on the type of the first value assigned to them. So, in the preceding example, the `name` and `height` variables are now scalars (that is, they can henceforth store only strings or numbers), `interests` is now a list variable (that is, it can store only lists), and `phone` is now a dictionary variable (and can store only dictionaries). Variable types, once assigned, are permanent and strictly enforced at runtime:

```
let interests = 'unknown' " Error: variable type mismatch
```

By default, a variable is scoped to the function in which it is first assigned to, or is global if its first assignment occurs outside any function. However, variables may also be explicitly declared as belonging to other scopes, using a variety of prefixes, as summarized in Table 1.

Table 1. Vimscript variable scoping

Prefix	Meaning
g:*varname*	The variable is global
s:*varname*	The variable is local to the current script file
w:*varname*	The variable is local to the current editor window
t:*varname*	The variable is local to the current editor tab
b:*varname*	The variable is local to the current editor buffer
l:*varname*	The variable is local to the current function
a:*varname*	The variable is a parameter of the current function
v:*varname*	The variable is one that Vim predefines

There are also *pseudovariables* that scripts can use to access the other types of value containers that Vim provides. These are summarized in Table 2.

Table 2. Vimscript pseudovariables

Prefix	Meaning
&*varname*	A Vim option (local option if defined, otherwise global)
&l:*varname*	A local Vim option
&g:*varname*	A global Vim option
@varname	A Vim register
$*varname*	An environment variable

The "option" pseudovariables can be particularly useful. For example, you could set up two key-maps to increase or decrease the current tabspacing like so:

```
nmap <silent> ]] :let &tabstop += 1<CR>
nmap <silent> [[ :let &tabstop -= &tabstop > 1 ? 1 : 0<CR>
```

Expressions

Note that the `[[` key-mapping in the previous example uses an expression containing a C-like "ternary expression":

```
&tabstop > 1 ? 1 : 0
```

This prevents the key map from decrementing the current tab spacing below the sane minimum of 1. As this example suggests, expressions in Vimscript are composed of the same basic operators that are used in most other modern scripting languages, and with generally the same syntax. The available operators (grouped by increasing precedence) are summarized in Table 3.

Table 3. Vimscript operator precedence table

Operation	Operator syntax
Assignment Numeric-add-and-assign Numeric-subtract-and-assign String-concatenate-and-assign	**let***var***=***expr* **let** *var***+=***expr* **let** *var***-=***expr* **let** *var***.=***expr*
Ternary operator	*bool***?***expr-if-true***:***expr-if-false*
Logical OR	*bool***||***bool*
Logical AND	*bool***&&***bool*
Numeric or string equality Numeric or string inequality Numeric or string greater-then Numeric or string greater-or-equal Numeric or string less than Numeric or string less-or-equal	*expr***==***expr* *expr***!=***expr* *expr***>***expr* *expr***>=***expr* *expr***<***expr* *expr***<=***expr*
Numeric addition Numeric subtraction String concatenation	*num***+***num* *num***-***num* *str***.***str*
Numeric multiplication Numeric division Numeric modulus	*num*******num* *num***/***num* *num***%***num*
Convert to number Numeric negation Logical NOT	**+***num* **-***num* **!***bool*
Parenthetical precedence	**(***expr***)**

Logical caveats

In Vimscript, as in C, only the numeric value zero is false in a boolean context; any non-zero numeric value—whether positive or negative—is considered true. However, all the logical and comparison operators consistently return the value 1 for *true*.

When a string is used as a boolean, it is first converted to an integer, and then evaluated for truth (*non-zero*) or falsehood (*zero*). This implies that the vast majority of strings—including most non-empty strings—will evaluate as being false. A typical mistake is to test for an empty string like so:

Listing 6. Flawed test for empty string

```
let result_string = GetResult();

if !result_string
   echo "No result"
endif
```

The problem is that, although this does work correctly when `result_string` is assigned an empty string, it also indicates `"No result"` if `result_string` contains a string like `"I am NOT an empty string"`, because that string is first converted to a number (*zero*) and then to a boolean (*false*).

The correct solution is to explicitly test strings for emptiness using the appropriate built-in function:

Listing 7. Correct test for empty string

```
if empty(result_string)
   echo "No result"
endif
```

Comparator caveats

In Vimscript, comparators always perform numeric comparison, unless both operands are strings. In particular, if one operand is a string and the other a number, the string will be converted to a number and the two operands then compared numerically. This can lead to subtle errors:

```
let ident = 'Vim'
if ident == 0 "Always true (string 'Vim' converted to number 0)
```

A more robust solution in such cases is:

```
if ident == '0'   "Uses string equality if ident contains string"but numeric equality if ident contains
 number
```

String comparisons normally honor the local setting of Vim's `ignorecase` option, but any string comparator can also be explicitly marked as case-sensitive (by appending a `#`) or case-insensitive (by appending a `?`):

Listing 8. Casing string comparators

```
if name ==? 'Batman'         |"Equality always case insensitive
   echo "I'm Batman"
elseif name <# 'ee cummings' |"Less-than always case sensitive
   echo "the sky was can dy lu minous"
endif
```

Using the "explicitly cased" operators for all string comparisons is strongly recommended, because they ensure that scripts behave reliably regardless of variations in the user's option settings.

Arithmetic caveats

When using arithmetic expressions, it's also important to remember that, until version 7.2, Vim supported only integer arithmetic. A common mistake under earlier versions was writing something like:

Listing 9. Problem with integer arithmetic

```
"Step through each file...
for filenum in range(filecount)
   " Show progress...
   echo (filenum / filecount * 100) . '% done'" Make progress...
   call process_file(filenum)
endfor
```

Because `filenum` will always be less than `filecount`, the integer division `filenum/filecount` will always produce zero, so each iteration of the loop will echo:

```
Now 0% done
```

Even under version 7.2, Vim does only floating-point arithmetic if one of the operands is explicitly floating-point:

```
let filecount = 234

echo filecount/100   |" echoes 2
echo filecount/100.0 |" echoes 2.34
```

Another toggling example

It's easy to adapt the syntax-toggling script shown earlier to create other useful tools. For example, if there is a set of words that you frequently misspell or misapply, you could add a script to your .vimrc to activate Vim's match mechanism and highlight problematic words when you're proofreading text.

For example, you could create a key-mapping (say: `;p`) that causes text like the previous paragraph to be displayed within Vim like so:

```
It's easy to adapt the syntax-toggling script shown earlier to create other useful
tools. For example, if there is a set of words that you frequently misspell or
misapply, you could add a script toyour .vimrc to activate Vim's match mechanism and
highlight problematic words when you're proofreading text.
```

That script might look like this:

Listing 10. Highlighting frequently misused words

```
"Create a text highlighting style that always stands out...
highlight STANDOUT term=bold cterm=bold gui=bold

"List of troublesome words...
let s:words = [
            \ "it's",  "its",
            \ "your",  "you're",
            \ "were",  "we're",   "where",
            \ "their", "they're", "there",
            \ "to",    "too",     "two"
            \ ]

"Build a Vim command to match troublesome words...
let s:words_matcher
```

```
\ = 'match STANDOUT /\c\<\(' . join(s:words, '\|') . '\)\>/'

"Toggle word checking on or off...
function! WordCheck ()
   "Toggle the flag (or set it if it doesn't yet exist)...
   let w:check_words = exists('w:check_words') ? !w:check_words : 1

   "Turn match mechanism on/off, according to new state of flag...
   if w:check_words
      exec s:words_matcher
   else
      match none
   endif
endfunction

"Use ;p to toggle checking...
nmap <silent>  ;p  :call WordCheck()<CR>
```

The variable `w:check_words` is used as a boolean flag to toggle word checking on or off. The first line of the `WordCheck()` function checks to see if the flag already exists, in which case the assignment simply toggles the variable's boolean value:

```
let w:check_words = exists('w:check_words') ? !w:check_words : 1
```

If `w:check_words` does not yet exist, it is created by assigning the value `1` to it:

```
let w:check_words = exists('w:check_words') ? !w:check_words : 1
```

Note the use of the `w:` prefix, which means that the flag variable is always local to the current window. This allows word checking to be toggled independently for each editor window (which is consistent with the behavior of the `match` command, whose effects are always local to the current window as well).

Word checking is enabled by setting Vim's `match` command. A `match` expects a text-highlighting specification (`STANDOUT` in this example), followed by a regular expression that specifies which text to highlight. In this case, that regex is constructed by `OR`'ing together all of the words specified in the script's `s:words` list variable (that is: `join(s:words, '\|')`). That set of alternatives is then bracketed by case-insensitive word boundaries (`\c\<\(...\)\>`) to ensure that only entire words are matched, regardless of capitalization.

The `WordCheck()` function then converts the resulting string as a Vim command and executes it (`exec s:words_matcher`) to turn on the matching facility. When `w:check_words` is toggled off, the function performs a `match none` command instead, to deactivate the special matching.

Scripting in Insert mode

Vimscripting is by no means restricted to Normal mode. You can also use the `imap` or `iabbrev` commands to set up key-mappings or abbreviations that can be used while inserting text. For example:

```
imap <silent> <C-D><C-D> <C-R>=strftime("%e %b %Y")<CR>
imap <silent> <C-T><C-T> <C-R>=strftime("%l:%M %p")<CR>
```

With these mappings in your .vimrc, typing CTRL-D twice while in Insert mode causes Vim to call its built-in `strftime()` function and insert the resulting date, while double-tapping CTRL-T likewise inserts the current time.

You can use the same general pattern to cause an insertion map or an abbreviation to perform *any* scriptable action. Just put the appropriate Vimscript expression or function call between an initial `<C-R>=` (which tells Vim to insert the result of evaluating what follows) and a final `<CR>` (which tells Vim to actually evaluate the preceding expression). Remember, though, that `<C-R>` (Vim's abbreviation for CTRL-R) is not the same as `<CR>` (Vim's abbreviation for a carriage return).

For example, you could use Vim's built-in `getcwd()` function to create an abbreviation for the current working directory, like so:

```
iabbrev <silent> CWD <C-R>=getcwd()<CR>
```

Or you could embed a simple calculator that can be called by typing CTRL-C during text insertions:

```
imap <silent> <C-C> <C-R>=string(eval(input("Calculate: ")))<CR>
```

Here, the expression:

```
string( eval( input("Calculate: ") ) )
```

first calls the built-in `input()` function to request the user to type in their calculation, which `input()` then returns as a string. That input string is then passed to the built-in `eval()`, which evaluates it as a Vimscript expression and returns the result. Next, the built-in `string()` function converts the numeric result back to a string, which the key-mapping's `<C-R>=` sequence is then able to insert.

A more complex Insert-mode script

Insertion mappings can involve scripts considerably more sophisticated than the previous examples. In such cases, it's usually a good idea to refactor the code out into a user-defined function, which the key-mapping can then call.

For example, you could change the behavior of CTRL-Y during insertions. Normally a CTRL-Y in Insert mode does a "vertical copy." That is, it copies the character in the same column from the line immediately above the cursor. For example, a CTRL-Y in the following situation would insert an "m" at the cursor:

```
Glib jocks quiz nymph to vex dwarf
Glib jocks quiz ny_
```

However, you might prefer your vertical copies to ignore any intervening empty lines and instead copy the character from the same column of the first *non-blank* line anywhere above the insertion point. That would mean, for instance, that a CTRL-Y in the following situation would also insert an "m", even though the immediately preceding line is empty:

```
Glib jocks quiz nymph to vex dwarf
```

```
Glib jocks quiz ny_
```

You could achieve this enhanced behavior by placing the following in your .vimrc file:

Listing 11. Improving vertical copies to ignore blank lines

```
"Locate and return character "above" current cursor position...
function! LookUpwards()
   "Locate current column and preceding line from which to copy...
   let column_num      = virtcol('.')
   let target_pattern  = '\%' . column_num . 'v.'
   let target_line_num = search(target_pattern . '*\S', 'bnW')

   "If target line found, return vertically copied character...
   if !target_line_num
      return ""
   else
      return matchstr(getline(target_line_num), target_pattern)
   endif
endfunction

"Reimplement CTRL-Y within insert mode...
imap <silent>  <C-Y>  <C-R><C-R>=LookUpwards()<CR>
```

The `LookUpwards()` function first determines which on-screen column (or "virtual column") the insertion point is currently in, using the built-in `virtcol()` function. The `'.'` argument specifies that you want the column number of the current cursor position:

```
let column_num = virtcol('.')
```

`LookUpwards()` then uses the built-in `search()` function to look backwards through the file from the cursor position:

```
let target_pattern = '\%' . column_num . 'v.'
let target_line_num = search(target_pattern . '*\S', 'bnW')
```

The search uses a special target pattern (namely: `\%`*`column_num`*`v.*\S`) to locate the closest preceding line that has a non-whitespace character (`\S`) at or after (`.*`) the cursor column (`\%`*`column_num`*`v`). The second argument to `search()` is the configuration string `bnW`, which tells the function to search **b**ackwards but **n**ot to move the cursor nor to **w**rap the search. If the search is successful, `search()` returns the line number of the appropriate preceding line; if the search fails, it returns zero.

The `if` statement then works out which character—if any—is to be copied back down to the insertion point. If a suitable preceding line was not found, `target_line_num` will have been assigned zero, so the first return statement is executed and returns an empty string (indicating "insert nothing").

If, however, a suitable preceding line was identified, the second return statement is executed instead. It first gets a copy of that preceding line from the current editor buffer:

```
return matchstr(getline(target_line_num), target_pattern)
```

It then finds and returns the one-character string that the previous call to `search()` successfully matched:

```
return matchstr(getline(target_line_num), target_pattern)
```

Having implemented this new vertical copy behavior inside `LookUpwards()`, all that remains is to override the standard CTRL-Y command in Insert mode, using an imap:

```
imap <silent> <C-Y> <C-R><C-R>=LookUpwards()<CR>
```

Note that, whereas earlier imap examples all used `<C-R>=` to invoke a Vimscript function call, this example uses `<C-R><C-R>=` instead. The single-CTRL-R form inserts the result of the subsequent expression as if it had been directly typed, which means that any special characters within the result retain their special meanings and behavior. The double-CTRL-R form, on the other hand, inserts the result as verbatim text without any further processing.

Verbatim insertion is more appropriate in this example, since the aim is to exactly copy the text above the cursor. If the key-mapping used `<C-R>=`, copying a literal escape character from the previous line would be equivalent to typing it, and would cause the editor to instantly drop out of Insert mode.

Learning Vim's built-in functions

As you can see from each of the preceding examples, much of Vimscript's power comes from its extensive set of over 200 built-in functions. You can start learning about them by typing:

```
:help functions
```

or, to access a (more useful) categorized listing:

```
:help function-list
```

Looking ahead

Vimscript is a mechanism for reshaping and extending the Vim editor. Scripting lets you create new tools (such as a problem-word highlighter) and simplify common tasks (like changing tabspacing, or inserting time and date information, or toggling syntax highlighting), and even completely redesign existing editor features (for example, enhancing CTRL-Y's "copy-the-previous-line" behavior).

For many people, the easiest way to learn any new language is by example. To that end, you can find an endless supply of sample Vimscripts—most of which are also useful tools in their own right—on the Vim Tips wiki. Or, for more extensive examples of Vim scripting, you can trawl the 2000+ larger projects housed in the Vim script archive.

If you're already familiar with Perl or Python or Ruby or PHP or Lua or Awk or Tcl or any shell language, then Vimscript will be both hauntingly familiar (in its general approach and concepts)

and frustratingly different (in its particular syntactic idiosyncrasies). To overcome that cognitive dissonance and master Vimscript, you're going to have to spend some time experimenting, exploring, and playing with the language. To that end, why not take your biggest personal gripe about the way Vim currently works and see if you can script a better solution for yourself?

This article has described only Vimscript's basic variables, values, expressions, and functions. The range of "better solutions" you're likely to be able to construct with just those few components is, of course, extremely limited. So, in future installments, we'll look at more advanced Vimscript tools and techniques: data structures, flow control, user-defined commands, event-driven scripting, building Vim modules, and extending Vim using other scripting languages. In particular, the next article in this series will focus on the features of Vimscript's user-defined functions and on the many ways they can make your Vim experience better.

Related topics

- A Byte of Vim
- Various hardcopy books on Vim
- Vim Tips wiki