Scripting the Vim editor, Part 4: Dictionaries

Learn when to use dictionaries for cleaner, faster code

Damian Conway (damian@conway.org)
CEO and Chief Trainer
Thoughtstream

10 February 2010

A dictionary is a container data structure that offers different optimizations and trade-offs from a list. In particular, in a dictionary the order of the elements stored is irrelevant and the identity of each element is explicit. In this fourth article in a series introducing Vimscript, Damian Conway introduces you to dictionaries, including an overview of their basic syntax and many functions. He concludes with several examples that illustrate the use of dictionaries for more efficient data processing and cleaner code.

View more content in this series

A *dictionary* in Vimscript is essentially the same as an AWK associative array, a Perl hash, or a Python dictionary. That is, it's an unordered container, indexed by strings rather than integers.

This fourth article in a series on Vimscript introduces this important data structure and explains its various functions for copying, filtering, extending, and pruning. The examples focus on the differences between lists and dictionaries, and on those cases where the use of a dictionary is a better alternative to the list-based solutions developed in Part 3 on built-in lists.

Dictionaries in Vimscript

You create a dictionary in Vimscript by using curly braces around a list of key/value pairs. In each pair, the key and value are separated by a colon. For example:

Listing 1. Creating a dictionary

```
let seen = {}   " Haven't seen anything yet

let daytonum = { 'Sun':0, 'Mon':1, 'Tue':2, 'Wed':3, 'Thu':4, 'Fri':5, 'Sat':6 }
let diagnosis = {
    \   'Perl'   : 'Tourettes',
    \   'Python' : 'OCD',
    \   'Lisp'   : 'Megalomania',
    \   'PHP'    : 'Idiot-Savant',
    \   'C++'    : 'Savant-Idiot',
    \   'C#'     : 'Sociopathy',
    \   'Java'   : 'Delusional',
    \}
```

Once you have created a dictionary, you can access its values using the standard square-bracket indexing notation, but using a string as the index instead of a number:

```
let lang = input("Patient's name? ")

let Dx = diagnosis[lang]
```

If the key doesn't exist in the dictionary, an exception is thrown:

```
let Dx = diagnosis['Ruby']
**E716: Key not present in Dictionary: Ruby**
```

However, you can access potentially non-existent entries safely, using the `get()` function. `get()` takes two arguments: the dictionary itself, and a key to look up in it. If the key exists in the dictionary, the corresponding value is returned; if the key doesn't exist, `get()` returns zero. Alternately, you can specify a third argument, in which case `get()` returns that value if the key isn't found:

```
let Dx = get(diagnosis, 'Ruby')
" Returns: 0

let Dx = get(diagnosis, 'Ruby', 'Schizophrenia')
" Returns: 'Schizophrenia'
```

There's a third way to access a particular dictionary entry. If the entry's key consists only of identifier characters (alphanumerics and underscores), you can access the corresponding value using the "dot notation," like so:

```
let Dx = diagnosis.Lisp                    " Same as: diagnosis['Lisp']

diagnosis.Perl = 'Multiple Personality'    " Same as: diagnosis['Perl']
```

This special limited notation makes dictionaries very easy to use as records or structs:

```
let user = {}

let user.name    = 'Bram'
let user.acct    = 123007
let user.pin_num = '1337'
```

Batch-processing of dictionaries

Vimscript provides functions that allow you to get a list of all the keys in a dictionary, a list of all its values, or a list of all its key/value pairs:

```
let keylist   = keys(dict)
let valuelist = values(dict)
let pairlist  = items(dict)
```

This `items()` function actually returns a list of lists, where each "inner" list has exactly two elements: one key and the corresponding value. Hence `items()` is especially handy for iterating through the entries of a dictionary:

```
for [next_key, next_val] in items(dict)
    let result = process(next_val)
    echo "Result for " next_key " is " result
endfor
```

Assignments and identities

Assignments in dictionaries work exactly as they do for Vimscript lists. Dictionaries are represented by references (that is, pointers), so assigning a dictionary to another variable aliases the two variables to the same underlying data structure. You can get around this by first copying or deep-copying the original:

```
let dict2 = dict1              " dict2 just another name for dict1

let dict3 = copy(dict1)        " dict3 has a copy of dict1's top-level elements

let dict4 = deepcopy(dict1)    " dict4 has a copy of dict1 (all the way down)
```

Just as for lists, you can compare identity with the *is* operator, and value with the == operator:

```
if dictA is dictB
    " They alias the same container, so must have the same keys and values
elseif dictA == dictB
    " Same keys and values, but maybe in different containers
else
    " Different keys and/or values, so must be different containers
endif
```

Adding and removing entries

To add an entry to a dictionary, just assign a value to a new key:

```
let diagnosis['COBOL'] = 'Dementia'
```

To merge in multiple entries from another dictionary, use the `extend()` function. Both the first argument (which is being extended) and the second argument (which contains the extra entries) must be dictionaries:

```
call extend(diagnosis, new_diagnoses)
```

`extend()` is also convenient when you want to add multiple entries explicitly:

```
call extend(diagnosis, {'COBOL':'Dementia', 'Forth':'Dyslexia'})
```

There are two ways to remove a single entry from a dictionary: the built-in `remove()` function, or the `unlet` command:

```
let removed_value = remove(dict, "key")unlet dict["key"]
```

When removing multiple entries from a dictionary, it is cleaner and more efficient to use `filter()`. The `filter()` function works much the same way as for lists, except that in addition to testing each entry's value using `v:val`, you can also test its key using `v:key`. For example:

Listing 2. Testing values and keys

```
" Remove any entry whose key starts with C...
call filter(diagnosis, 'v:key[0] != "C"')

" Remove any entry whose value doesn't contain 'Savant'...
call filter(diagnosis, 'v:val =~ "Savant"')

" Remove any entry whose value is the same as its key...
call filter(diagnosis, 'v:key != v:val')
```

Other dictionary-related functions

In addition to `filter()`, dictionaries can use several other of the same built-in functions and procedures as lists. In almost every case (the notable exception being `string()`), a list function applied to a dictionary behaves as if the function had been passed a list of the values of the dictionary. Listing 3 shows the most commonly used functions.

Listing 3. Other list functions that also work on dictionaries

```
let is_empty = empty(dict)           " True if no entries at all

let entry_count = len(dict)          " How many entries?

let occurrences = count(dict, str)   " How many values are equal to str?

let greatest = max(dict)             " Find largest value of any entry
let least    = min(dict)             " Find smallest value of any entry

call map(dict, value_transform_str)  " Transform values by eval'ing string

echo string(dict)                    " Print dictionary as key/value pairs
```

The `filter()` built-in is particularly handy for normalizing the data in a dictionary. For example, given a dictionary containing the preferred names of users (perhaps indexed by userids), you could ensure that each name was correctly capitalized, like so:

```
call map( names, 'toupper(v:val[0]) . tolower(v:val[1:])' )
```

The call to `map()` walks through each value, aliases it to `v:val`, evaluates the expression in the string, and replaces the value with the result of that expression. In this example, it converts the first character of the name to uppercase, and the remaining characters to lowercase, and then uses that modified string as the new name value.

Deploying dictionaries for cleaner code

The third article in this series explained Vimscript's *variadic* function arguments with a small example that generated comment boxes around a specified text. Optional arguments could be added after the text string to specify the comment introducer, the character used as the "box," and the width of the comment. Listing 4 reproduces the original function.

Listing 4. Passing optional arguments as variadic parameters

```
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
```

Variadic arguments are convenient for specifying function options but suffer from two major drawbacks: they impose an explicit ordering on the function's parameters, and they leave that ordering implicit in function calls.

Revisiting autocomments

As Listing 4 illustrates, when any arguments are optional, it is usually necessary to decide in advance the order in which they must be specified. This necessity presents a design problem, however: in order to specify a later option, the user will have to explicitly specify all the options before it as well. Ideally, the first option would be the most commonly used one, the second would be the second-most commonly used, etc. In reality, deciding on this order before the function is widely deployed can be difficult: how are you supposed to know which option will be most important to most people?

The `CommentBlock()` function in Listing 4, for example, assumes that the comment introducer is the optional argument that is most likely to be needed, and so places it first in the parameter list. But what if a user of the function only ever programs in C and C++, and so never alters the default introducer? Worse, what if it turns out that the width of comment blocks varies for every new project? This will prove very annoying, because developers will now have to specify all three optional arguments every time, even though the first two are always given their default values:

```
" Comment of required width, with standard delimiter and box character...
let new_comment = CommentBlock(comment_text, '//', '*', comment_width)
```

This leads directly to the second issue, namely that when any options do need to be specified explicitly, it is likely that several of them will have to be specified. However, because options default to the most commonly needed values, the user may be unfamiliar with specifying options, and hence unfamiliar with the necessary order. This can lead to implementation errors like the following:

```
" Box comment using ==== to standard line width...
let new_comment = CommentBlock(comment_text, '=', 72)
```

…which, rather disconcertingly, produces a (non-)comment that looks like this:

```
=72727272727272727272727272727272 = A bad comment =72727272727272727272727272727272
```

The problem is that the optional arguments have nothing explicit to indicate which option they are supposed to set. Their meaning is determined implicitly by their position in the argument list, and so any mistake in their ordering silently changes their meaning.

This is a classic case of using the wrong tool for the job. Lists are perfect when order is significant and identity is best implied by position. But, in this example, the order of the optional arguments is more a nuisance than a benefit and their positions are easily confused, which can lead to subtle errors of misidentification.

What's wanted is, in a sense, the exact opposite of a list: a data structure where order is irrelevant, and identity is explicit. In other words, a dictionary. Listing 5 shows the same function, but with its options specified via a dictionary, rather than with variadic parameters.

Listing 5. Passing optional arguments in a dictionary

```
function! CommentBlock(comment, opt)
    " Unpack optional arguments...
    let introducer = get(a:opt, 'intro', '//'                )
    let box_char   = get(a:opt, 'box',   '*'                 )
    let width      = get(a:opt, 'width', 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
```

In this version of the function, only two arguments are passed: the essential comment text, followed by a dictionary of options. The built-in `get()` function is then used to retrieve each option, or its default value, if the option was not specified. Calls to the function then use the named option/value pairs to configure its behavior. The implementation of the parameter parsing within the function becomes a little cleaner, and calls to the function becomes much more readable, and less error-prone. For example:

```
" Comment of required width, with standard delimiter and box character...
let new_comment = CommentBlock(comment_text, {'width':comment_width})

" Box comment using ==== to standard line width...
let new_comment = CommentBlock(comment_text, {'box':'=', 'width':72})
```

Refactoring autoalignments

In the third article in this series, we updated an earlier example function called `AlignAssignments()`, converting it to use lists to store the text lines it was modifying. Listing 6 reproduces that updated version of the function.

Listing 6. The 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
```

This version greatly improved the efficiency of the function, by caching data rather than reloading it, but it did so at the expense of maintainability. Specifically, because it stored the various components of each line in small three-element arrays, the code is littered with "magic indexes" (such as `v:val[0]` and `line[1]`) whose names give no clue as to their purpose.

Dictionaries are tailor-made for solving this problem, because, like lists, they aggregate data into a single structure, but, unlike lists, they label each datum with a string, rather than with a number. If those strings are selected carefully, they can make the resulting code much clearer. Instead of magic indexes, we get meaningful names (such as `v:val.lval` for each line's `lvalue` and `line.op` for each line's operator).

Rewriting the function using dictionaries is trivially easy, as Listing 7 demonstrates.

Listing 7. A further-improved 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)
        if len(fields)
            call add(lines, {'lval':fields[1], 'op':fields[2], 'rval':fields[3]})
        else
            call add(lines, {'text':linetext,  'op':''                          })
        endif
    endfor

    " Determine maximal lengths of lvalue and operator...
    let op_lines = filter(copy(lines),'!empty(v:val.op)')
    let max_lval = max( map(copy(op_lines), 'strlen(v:val.lval)') ) + 1
    let max_op   = max( map(copy(op_lines), 'strlen(v:val.op)'  ) )

    " Recompose lines with operators at the maximum length...
    let linenum = firstline
    for line in lines
        let newline = empty(line.op)
        \ ? line.text
        \ : printf("%-*s%*s%s", max_lval, line.lval, max_op, line.op, line.rval)
        call setline(linenum, newline)
        let linenum += 1
    endfor
endfunction
```

The differences in this new version are marked in bold. There are only two: the record for each line is now a dictionary rather than a hash, and the subsequent accesses to elements of each record use named lookups instead of numeric indexing. The overall result is that the code is more readable and less prone to the kinds of off-by-one errors common to array indexing.

Dictionaries as data structures

Vim provides a built-in command that allows you to remove duplicate lines from a file:

```
:%sort u
```

The `u` option causes the built-in `sort` command to remove duplicate lines (once they've been sorted), and the leading `%` applies that special `sort` to the entire file. That's handy, but only if you don't care about preserving the original order of the unique lines in the file. This might be a problem if the lines are a list of prize winners, a sign-up sheet for a finite resource, a to-do list, or any other sequence in which first-in should remain best-dressed.

Sort-free uniqueness

The keys of a dictionary are inherently unique, so it's possible to use a dictionary to remove duplicate lines from a file, and to do so in a way that preserves the original ordering of those lines. Listing 8 illustrates a simple function that achieves this goal.

Listing 8. A function for order-preserving uniqueness

```
function! Uniq () range
    " Nothing unique seen yet...
    let have_already_seen = {}
    let unique_lines = []

    " Walk through the lines, remembering only the hitherto-unseen ones...
    for original_line in getline(a:firstline, a:lastline)
        let normalized_line = '>' . original_line
        if !has_key(have_already_seen, normalized_line)
            call add(unique_lines, original_line)
            let have_already_seen[normalized_line] = 1
        endif
    endfor

    " Replace the range of original lines with just the unique lines...
    exec a:firstline . ',' . a:lastline . 'delete'
    call append(a:firstline-1, unique_lines)
endfunction
```

The `Uniq()` function is declared to take a range, so it will only be called once, even when invoked on a range of lines in the buffer.

When called, it first sets up an empty dictionary (`have_already_seen`) that will be used to track which lines have already been encountered within the specified range. Lines that haven't been seen before will then be added to the list stored in `unique_lines`.

The function then provides a loop that does precisely that. It grabs the specified range of lines from the buffer with a `getline()` and iterates through each. It first adds a leading `'>'` to each line to ensure it is not empty. Vimscript dictionaries cannot store an entry whose key is an empty string, so empty lines from the buffer would not be correctly added to `have_already_seen`.

Once the line is normalized, the function then checks whether that line has already been used as a key in the `have_already_seen` dictionary. If so, an identical line must already have been seen and added to `unique_lines`, so the copy can be ignored. Otherwise, the line is being encountered for the first time, so the original (un-normalized) line must be added to `unique_lines`, and the normalized version must be added as a key in `have_already_seen`.

When all the lines have been filtered in this way, `unique_lines` will contain only the unique subset of them, in the order in which they were first encountered. All that remains is to delete the original set of lines and replace it (via an `append()`) with these accumulated unique lines.

With such a function available, you could set up a Normal-mode keymap to invoke the command on entire files, like so:

```
nmap ;u :%call Uniq()<CR>
```

Or you could apply it to a specific set of lines (for example, a range that had been selected in Visual mode), like so:

```
vmap  u :call Uniq()<CR>
```

Looking ahead

The basic features of Vimscript covered so far (statements and functions, arrays, and hashes) are sufficient to create almost any kind of addition to Vim's core feature set. But all the extensions we have seen have required the user to explicitly request behavior, by issuing a Normal-mode command or typing a particular sequence in Insert mode.

In the next article in this series, we'll investigate Vim's built-in event model and explore how to set up user-defined functions that trigger automatically as the user edits.

Resources

Learn

- 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.
- "Scripting the Vim editor, Part 3: Built-in lists" (developerWorks, January 2010) introduces the list data structure and walks through several examples illustrating its uses.
- 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
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tutorials and Linux tips on developerWorks.
- Stay current with developerWorks technical events and Webcasts.
- Follow developerWorks on Twitter.

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. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Damian Conway

Damian Conway is an Adjunct Associate Professor of Computer Science at Monash University, Australia, and CEO of Thoughtstream, an international IT training company. Vim is his primary code development environment, and Vimscript is one of his two favorite programming languages.