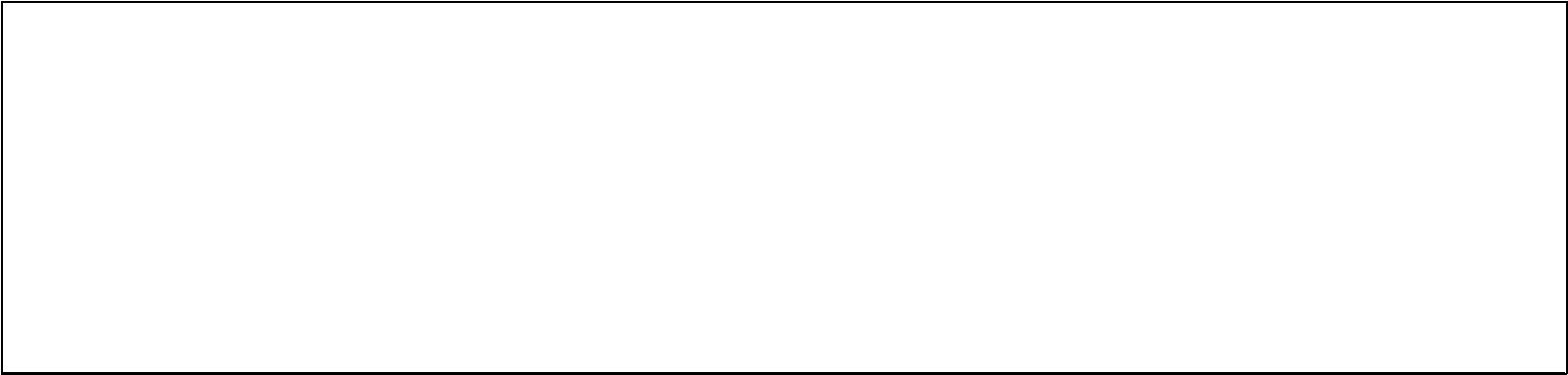
­­­­­

**Scripting the Vim editor, PartDictionaries4:**

**Learn when to use dictionaries for cleaner, faster code**



|  |  |  |
| --- | --- | --- |
| A dictionary is a container data structure that offers different optimizations and | | trad |
| list. In particular, in a dictionary the order of the elements stored is irrelevant | | and |
| each element is explicit. In this fourth  [seriesarticl](http://www.ibm.com/developerworks/views/linux/libraryview.jsp?site_id=1&contentarea_by=Linux&sort_by=Date&sort_order=1&start=1&end=4&topic_by=All%20topics%20and%20related%20products&product_by=&type_by=Articles&show_abstract=true&search_by=scripting%20the%20vim%20editor)eintroducinga | Vimscript, Damian Conway |  |
| introduces you to dictionaries, including an overview of their basic syntax and many fu | | |
| He concludes with several examples that illustrate the use of dictionaries for more eff | | |
| processing and cleaner code. |  |  |
| [View more content in this](https://www.ibm.com/developerworks/library/?series_title_by=scripting+the+vim+editor) series |  |  |

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

This fourth article [seriesi](http://www.ibm.com/developerworks/views/linux/libraryview.jsp?site_id=1&contentarea_by=Linux&sort_by=Date&sort_order=1&start=1&end=4&topic_by=All%20topics%20and%20related%20products&product_by=&type_by=Articles&show_abstract=true&search_by=scripting%20the%20vim%20editor)na on Vimscript introduces this important data structure and expla its various functions for copying, filtering, extending, and pruning. The examples focus differences between lists and dictionaries, and on those cases where the use of a dictio better alternative to the list-based solutions  [developedPar](http://www.ibm.com/developerworks/linux/library/l-vim-script-3/index.html)t3on builtin-in lists.

**Dictionaries in Vimscript**

You create a dictionary in Vimscript by using curly braces around a list of key/value pa 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-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:

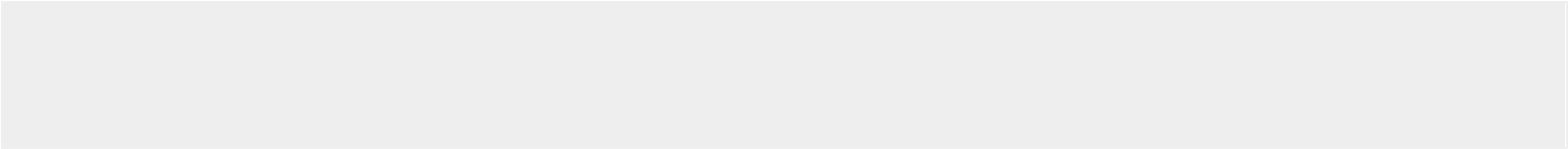


let Dx = diagnosis['Ruby']

\*\*E716: Key not present in Dictionary: Ruby\*\*

However, you can access potentially non-existent entries safely,get()usingfunctiontheget()

takes two arguments: the dictionary itself, and a key to look up in it. If the key exist dictionary, the corresponding value is returned; if the key get()doesn'treturnexi**s**t,zero. Alternately, you can specify a third argument, in get() returnscase that value if the key is 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 identifier characters (alphanumerics and underscores), you can access the corresponding 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 dictionar 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 exact elements: one key and the corresponding value.

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)

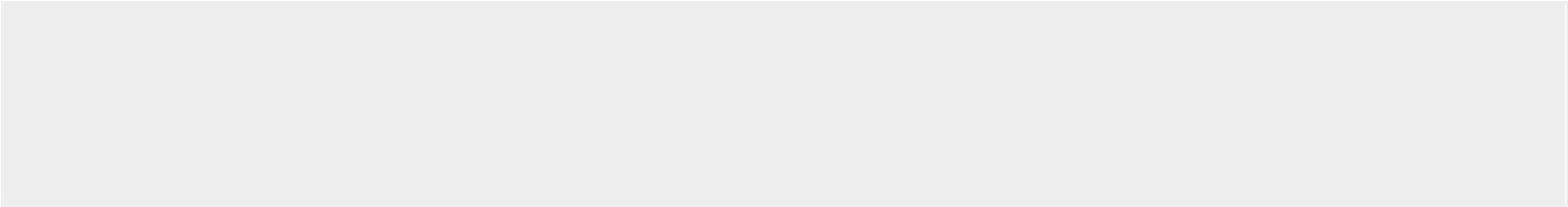
echo "Result for " next\_key " is " result endfor

**Assignments and identities**

Assignments in dictionaries work exactly as they do for Vimscript lists. Dictionaries ar by references (that is, pointers), so assigning a dictionary to another variable aliases variables to the same underlying data structure. You can get around this by first copyin 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*is* withoperator,the and value with*==* theoperator:



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,extend()thefunction. Both the first argument (which is being extended) and the second argument (which contains the extra ent 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'})

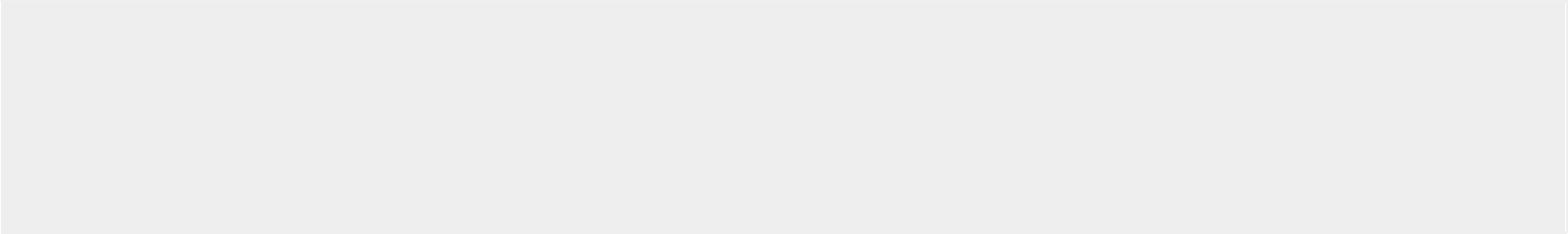
There are two ways to remove a single entry from a dictionary:remove()builtfunction,-in or the unlet command:



let removed\_value = remove(dict, "key")unlet dict["key"]

When removing multiple entries from a dictionary, it is cleaner and more filter(). to us The filter() function works much the same way as for lists, except that in addition to te entry's value usingv:val, you can also test its keyv:keyusing. 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 filter(), dictionaries can use several other of the same built-in functions a procedures as lists. In almost every case (the notable exceptionstring()being),a list function applied to a dictionary behaves as if the function had been passed a list of the values 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 given a dictionary containing the preferred names of users (perhaps indexed by userids), could ensure that each name was correctly capitalized, like so:



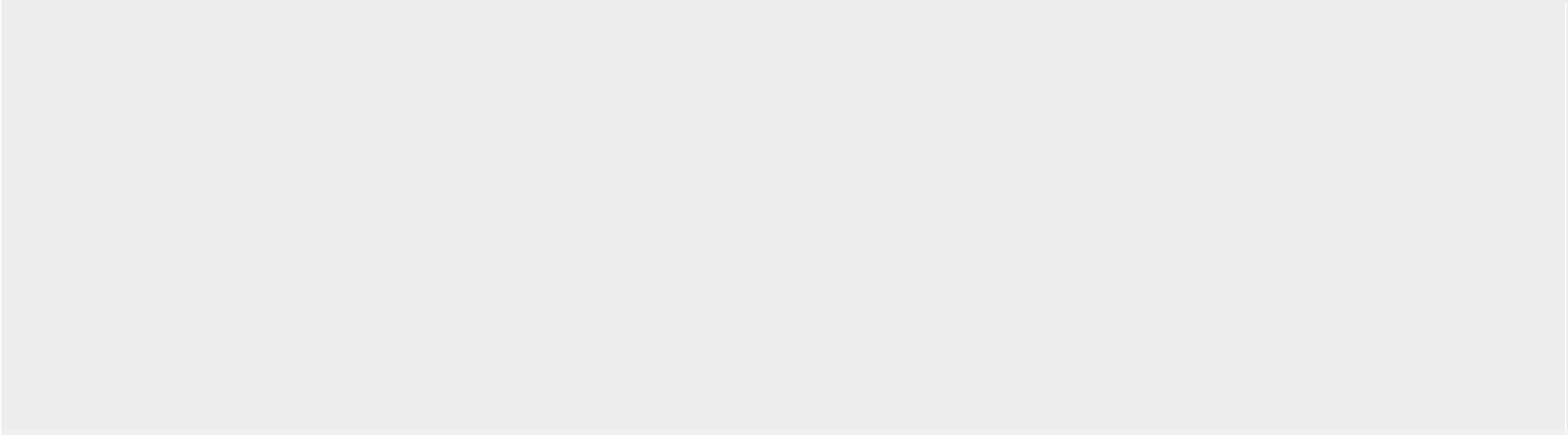
call map( names, 'toupper(v:val[0]) . tolower(v:val[1:])' )

The call map() walks through each value, aliasesv:val ,toevaluates the expression in the string, and replaces the value with the result of that expression. In this example, it c character of the name to uppercase, and the remaining characters to lowercase, and then that modified string as the new name value.

**Deploying dictionaries for cleaner code**

The  [third](http://www.ibm.com/developerworks/linux/library/l-vim-script-3/index.html) articlein this series explained Vimsc*variadic*pt's function arguments with a small example that generated comment boxes around a specified text. Optional arguments could b added after the text string to specify the comment introducer, the character used as the 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 ma drawbacks: they impose an explicit ordering on the function's parameters, and they leave ordering implicit in function calls.

**Revisiting autocomments**

As  [Listing](#page5) 4illustrates, when any arguments are optional, it is usually necessary to deci advance the order in which they must be specified. This necessity presents a design prob however: in order to specify a later option, the user will have to explicitly specify al before it as well. Ideally, the first option would be the most commonly used one, the se be the second-most commonly used, etc. In reality, deciding on this order before the fun is widely deployed can be difficult: how are you supposed to know which option will be m important to most people?

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



" 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 spec explicitly, it is likely that several of them will have to be specified. However, becaus default to the most commonly needed values, the user may be unfamiliar with specifying o and hence unfamiliar with the necessary order. This can lead to implementation errors li 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:

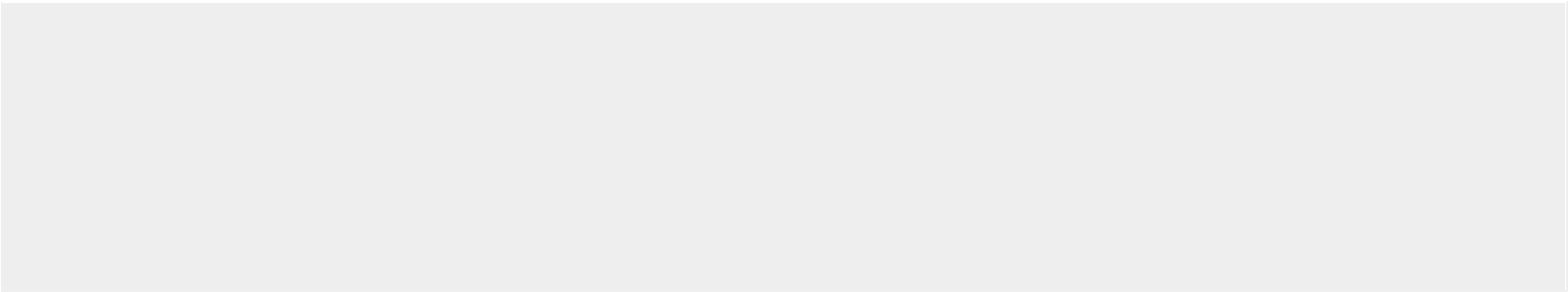
=727272727272727272727272727272 = A bad comment =727272727272727272727272727272

The problem is that the optional arguments have nothing explicit to indicate which optio supposed to set. Their meaning is determined implicitly by their position in the argumen 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 and identity is best implied by position. But, in this example, the order of the optiona more a nuisance than a benefit and their positions are easily confused, which can lead t errors of misidentification.

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

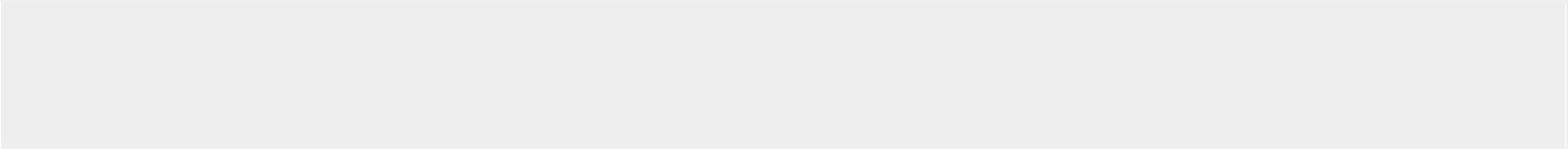
**Listing 5. Passing optional arguments in a dictionary**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 te followed by a dictionary of options. Theget()builtfunction-in is then used to retrieve each opti or its default value, if the option was not specified. Calls to the function then use th value pairs to configure its behavior. The implementation of the parameter parsing withi function becomes a little cleaner, and calls to the function becomes much more readable, error-prone. For example:



* Comment of required width, with standard delimiter and box character...

let new\_comment = CommentBlock(comment\_text, {'width':comment\_width})

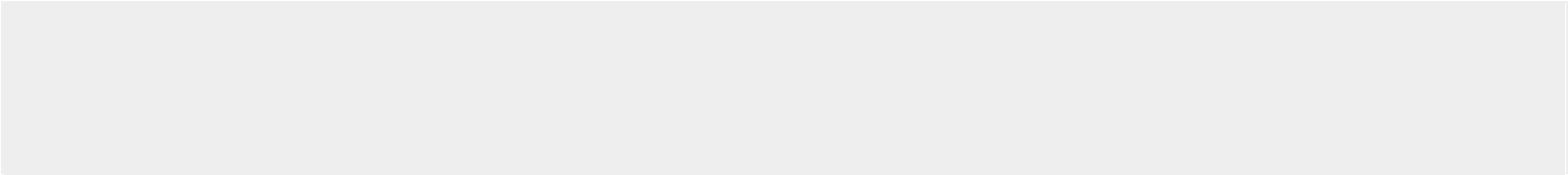
* Box comment using ==== to standard line width...

let new\_comment = CommentBlock(comment\_text, {'box':'=', 'width':72})

**Refactoring autoalignments**

In the [third](http://www.ibm.com/developerworks/linux/library/l-vim-script-3/index.html) articlein this series, we updated an earlier example function called AlignAssignments(), converting it to use lists to store the text lines it was modifying. reproduces that updated version of the function.

**Listing 6. The updated AlignAssignments() function**



function! AlignAssignments ()

" Patterns needed to locate assignment operators...

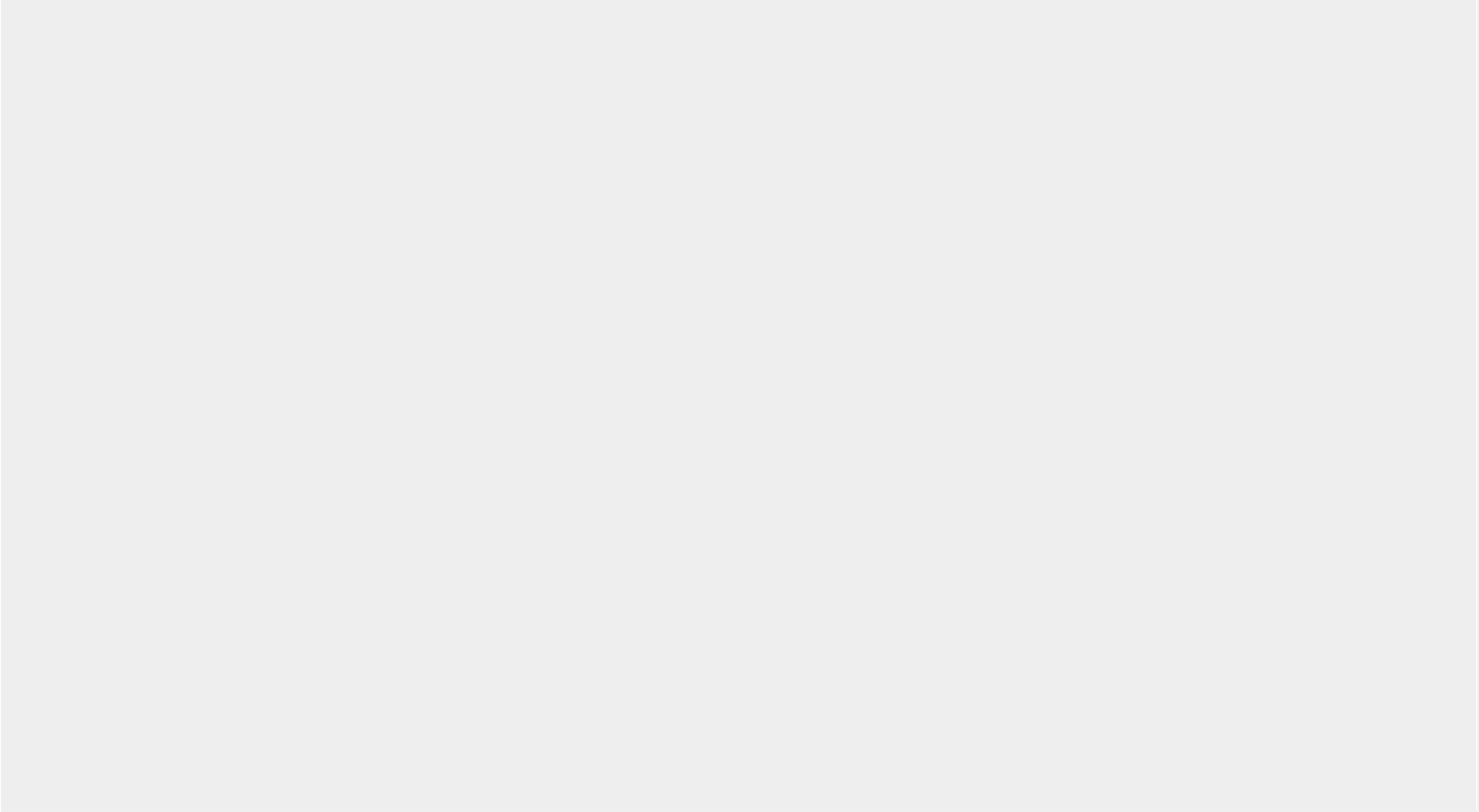
let ASSIGN\_OP = '[-+\*/%|&]\?=\@<!=[=~]\@!'

let ASSIGN\_LINE = '^\(.\{-}\)\s\*\(' . ASSIGN\_OP . '\)\(.\*\)$'

" Locate block of code to be considered (same indentation, no blanks)...

Scripting the Vim editor, Part 4: Dictionaries Page 6 of11

|  |  |  |  |
| --- | --- | --- | --- |
| ibm.com/developerWorks/ | | | developerWorks® |
| 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

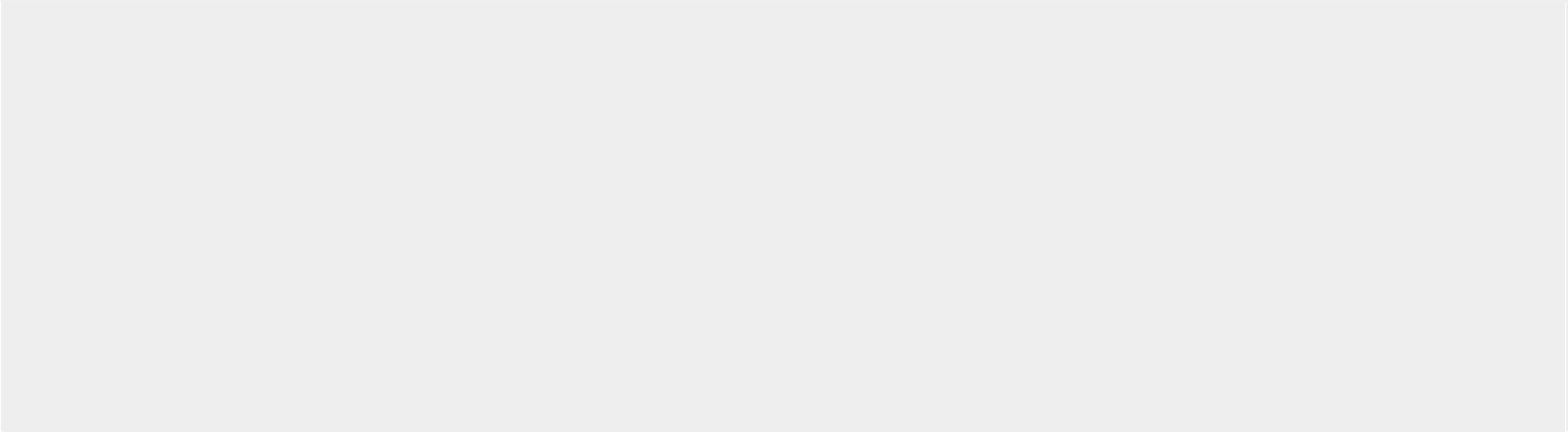
endfunction

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

Dictionaries are tailor-made for solving this problem, because, like lists, they aggrega a single structure, but, unlike lists, they label each datum with a string, rather than If those strings are selected carefully, they can make the resulting code much clearer. magic indexes, we get meaningful names (suchv:val.lval for each line'slvalue andline.op

for each line's operator).

**Listing 7. A further-improved AlignAssignments() function**

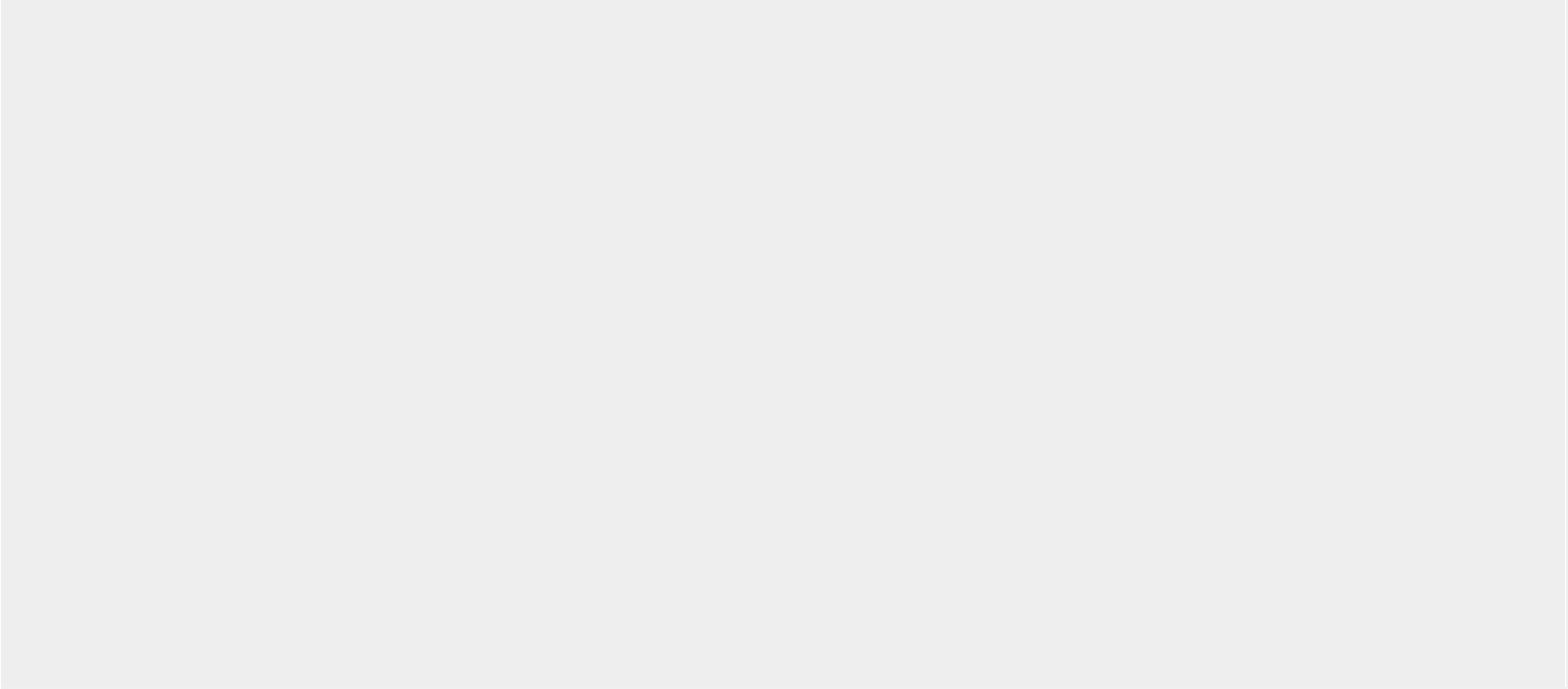


function! AlignAssignments ()

|  |  |  |  |
| --- | --- | --- | --- |
| " | 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 f line is now a dictionary rather than a hash, and the subsequent accesses to elements of record use named lookups instead of numeric indexing. The overall result is that the cod 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:



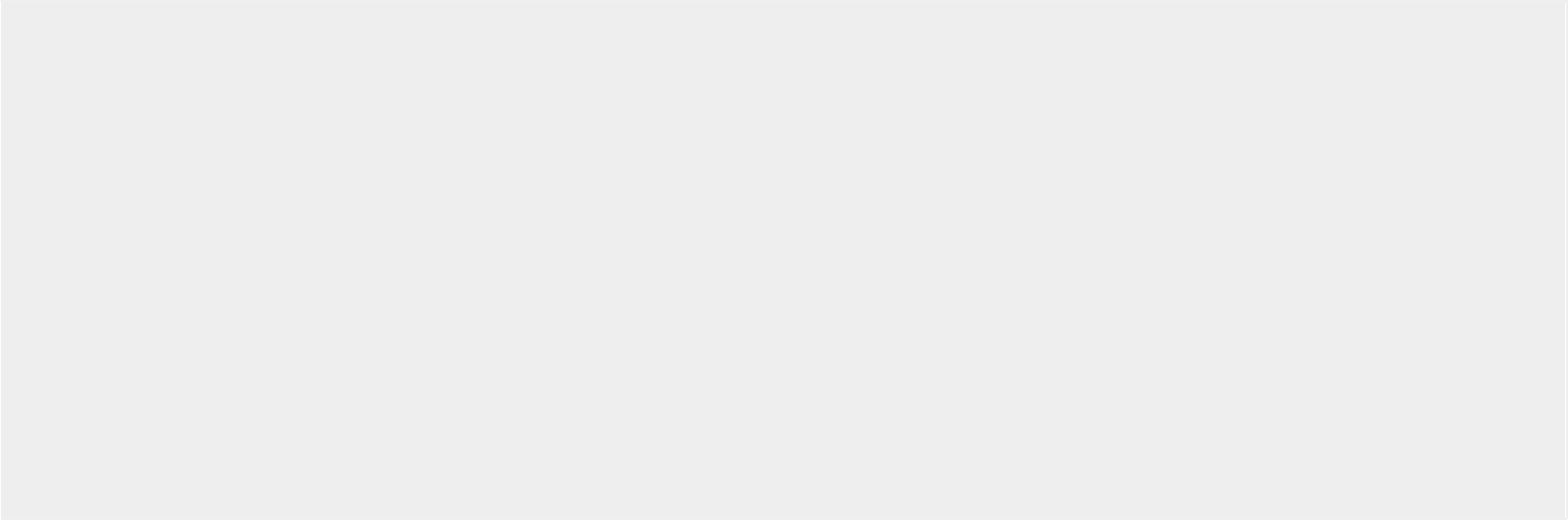
:%sort u

That's handy, but only if you don't care about preserving the original order of the unique lines in the file.

**Sort-free uniqueness**

The keys of a dictionary are inherently unique, so it's possible to use a dictionary to duplicate lines from a file, and to do so in a way that preserves the original ordering.

**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 whe on a range of lines in the buffer.

When called, it first sets up an empty have\_already\_seen) that will be used to track which lines have already been encountered within the specified range. Lines that haven't seen before will then be added to the listunique\_lines.

The function then provides a loop that does precisely that. It grabs the specified range the buffer withgetline() and iterates through each. It first adds'>' leadingtoeach line to ensure it is not empty. Vimscript dictionaries cannot store an entry whose key is an emp so empty lines from the buffer would not be correctlyhave\_already\_seen.

Once the line is normalized, the function then checks whether that line has already been a key in thehave\_already\_seen dictionary. If so, an identical line must already have been s and added tounique\_lines , so the copy can be ignored. Otherwise, the line is being encoun for the first time, so the original (un-normalized) line mustunique\_lines, toand the normalized version must be added as a keyhave\_already\_seen.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| When all | the lines have been filtered in |  | will contain only the unique subset |  |
| unique\_lines |  |
| of them, | in the order in which they were first encountered. All that remains is to delet | | |  |

set of lines and replace it append() ) with these accumulated unique lines.

you could set up a Normal-mode keymap to invoke the comm 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 sel Visual mode), like so:



vmap u :call Uniq()<CR>