Extracted from:

The VimL Primer

Edit Like a Pro with Vim Plugins and Scripts

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

The VimL Primer

Edit Like a Pro with Vim Plugins and Scripts

Benjamin Klein

Edited by Lynn Beighley and Fahmida Y. Rashid

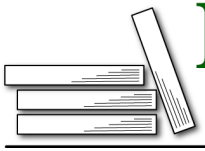
The VimL Primer

Edit Like a Pro with Vim Plugins and Scripts

Benjamin Klein

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Lynn Beighley and Fahmida Y. Rashid (editor)
Candace Cunningham (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-040-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2015

VimL, as you learned in the introduction, is based on Ex commands. To take full advantage of its capabilities, though, we need to move beyond those commands to functions—both the built-in ones that Vim provides and our own—types, logic, and the other additions that bring VimL from the Ex *command set* to the language level.

In this chapter we briefly go over VimL's syntax. You'll see how to write and call functions, define variables, iterate over collections of items, and more. We'll finish by looking at the directory structure of a typical Vim plugin and getting ready to create our own plugin.

Functions, Types, and Variables

Vim includes many built-in functions that we can call in our own code—everything from sort() and search() to browse() and winheight(). We can also write our own functions, using function and endfunction. Our functions have to begin with uppercase letters to distinguish them from built-in functions. Here's an example that uses the command :echo to output a message to the user:

intro/function.vim

```
function! EchoQuote()
  echo 'A poet can but ill spare time for prose.'
endfunction
```

To call this function, we need to save this code in a file, so let's do that first. Then we need to tell Vim to load, or source, that file. We do this by calling the :source command on the Vim command line, like this:

```
:source %
```

We pass % in the command as an argument. % is a shortcut character that stands for the name of the file we're currently editing, so that :source % essentially means to source the current file.

When we call the command, Vim prints the function's output as a message:

```
" → A poet can but ill spare time for prose.
```

(We show output using the Vim comment syntax, followed by an arrow.)

Let's look at our function file again. Did you catch the ! (*bang*) at the end of the function's first line?

```
function! EchoQuote()
```

When Vim loads this file, it will define a function called EchoQuote(). If there's already a function with that name—for example, if there's one from when we last loaded this file—we would have a name collision. So adding the bang to

the end of function tells Vim that if this happens, it should overwrite the existing EchoQuote() function with this one.

The ! modifier is common with Ex commands—for example, :q! quits Vim without asking us about unsaved changes. Similarly, the command :function! silently overwrites existing functions, so it's good to be careful about adding the bang if there's any chance that our function could conflict with an existing one declared elsewhere.

Notice also that, in our function above, there's no colon (:) at the beginning of the :echo command. Normally we would use the colon to start a command in a Vim session, but in a VimL script colons are optional.

We declare variables with let:

```
function! EchoQuote()
  let quote = 'A poet can but ill spare time for prose.'
  echo quote
endfunction
```

And we can take arguments. If our function requires an argument, we include the argument's name between the parentheses when we declare the function; these are called *named arguments*. To refer to a named argument in our function, we append the a: argument prefix:

```
function! EchoQuote(quote)
  echo a:quote
endfunction
call EchoQuote('A poet can but ill spare time for prose.')

" → A poet can but ill spare time for prose.
```

We can also take optional arguments—arguments that *can* be given to our function but aren't required. To allow optional arguments, we add ellipses (...) after the named arguments in the function declaration. Within our function, Vim numbers optional arguments beginning with 1 and automatically stores them in a List variable called a:000.

So to access our optional arguments, we can either refer to them by their number or refer to their entry in the List. In this version of EchoQuote(), we take both approaches:

```
function! EchoQuote(quote, ...)
  let year = a:1
  let author = a:000[1]
  echo 'In ' . year . ', ' . author . ' said: "' . a:quote . '"'
endfunction
```

```
call EchoQuote('A poet can but ill spare time for prose.',
            \ '1784', 'William Cowper')

" → In 1784, William Cowper said: "A poet can but ill spare time for prose."
```

Here, we define two variables, year and author, using the first two optional arguments. Unlike the numbering system Vim uses for optional arguments, a VimL List (like a:000) is zero-indexed, meaning it starts counting from 0. So a:1 is the *first* optional argument, but a:000[1] is the *second* argument.

In the last line, we use the :call command to call our function. At the end of the function, the line that we echo is a concatenated String variable; as you can see, we use the dot (.) to concatenate String values.

One final thing about this function: you might have noticed that the last line of code, where we call EchoQuote(), is actually broken into two lines. We can split a line up like this using \, VimL's *line-continuation operator*. When we want to break up lines, we just have to start each new line with this operator. Note that it starts each new line—it doesn't end the first line. This can be helpful when we have long lines that might scroll way off of the screen, or even just to help us format function arguments neatly. (For more on this operator, see :help line-continuation.)

Variable Scopes

Variable names can contain letters, underscores, and digits—although they can't start with digits. There are also several variable scopes, which we refer to using prefixes. What we saw in our last function, where our variables were written a:quote, was the *argument scope*, used for function arguments. Two others are the *global scope*, which is the default scope, and the *function scope*.

intro/variable.vim

```
let g:quote = 'A poet can but ill spare time for prose.'

function! EchoQuote()
  let l:quote = 'Local: A poet can but ill spare time for prose.'
  return l:quote
endfunction
```

In these examples, g:quote is a global variable, and l:quote is a function-specific variable. The scope is marked by the prefix, just like variables in the argument scope use the a: prefix.

The function scope doesn't relate to arguments, though—its purpose is to distinguish variables in our function from other variables with similar names. Similarly, we use the g: prefix, for global scope, to distinguish a variable outside

of our function from one defined inside of it. If our function had a quote of its own but we wanted to refer to a quote variable outside of the function—the global variable—we'd write g:quote. If we wanted to define a variable with a name that's reserved or already taken, we could name it using the function-specific prefix, such as l:quote. Except in these kinds of cases, the prefixes are optional; we can give all of our variables the correct prefixes, or we can leave them off unless they're needed.

As with scopes, VimL has a number of variable types—six, to be exact. We've already seen examples of List and String, but there are also Number, Funcref—a variable referring to a function—Dictionary, and Float. Let's quickly go over each.

Number

Number variables can be decimal, octal, or hexadecimal. They're easy to tell apart: octal numbers start with 0, hexadecimal numbers start with either 0x or 0X, and any other number is decimal. Another way to tell them apart is to use the :echo command, which prints only decimal values:

```
:echo 10          " → 10
:echo 023         " → 19
:echo 0x10        " → 16
```

Of course, since a 0 at the beginning is what distinguishes an octal Number, we can't start decimal numbers with 0.

Negative numbers start with a - character. That's also the subtraction operator, and the other usual arithmetic operators also work as we might expect:

```
:echo 20 - 10      " → 10
:echo 10 + -012    " → 0
:echo 0x32 / 0xa   " → 5
:echo 59 * 19      " → 1121
```

String

As with Number, there are a couple of different kinds of String variables.

```
"I sing the Sofa. I who lately sang\nTruth, Hope, and Charity..."
'I sing the Sofa. I who lately sang\nTruth, Hope, and Charity...'
```

Those two are exactly the same String. What happens when we echo them?

intro/string.vim

```
:echo "I sing the Sofa. I who lately sang\nTruth, Hope, and Charity..."
" → I sing the Sofa. I who lately sang
    Truth, Hope, and Charity...

:echo 'I sing the Sofa. I who lately sang\nTruth, Hope, and Charity...'
```

```
" → I sing the Sofa. I who lately sang\nTruth, Hope, and Charity...
```

The only difference between these two strings is the quotes. In VimL, double-quoted strings can use a variety of special characters (see :help expr-quote). Our string above contains an \n, the special character for a new line. In single-quoted strings, we can escape a single quote by putting two together, but other than that the characters themselves are preserved, as you can see.

A funny thing about the double-quoted String is what happens when we leave off the ending quotes:

```
:echo "I sing the Sofa. I who lately sang"
" Truth, Hope, and Charity, and touch'd with awe
:echo "The solemn chords..."
```

The double quote is also what starts out a VimL comment. Comments can be either on their own lines or following commands on a line:

```
:ls " The command to list all buffers.
```

The catch is that we can't do this with commands that expect a double quote as part of an argument.

Funcref

A Funcref is a variable that refers to a function. It's like a variable placeholder for the function—we use it in place of the function itself, and, like function names, Funcref names have to begin with an uppercase letter.

To assign a Funcref variable, we use function():

intro/funcref.vim

```
let Example = function('EchoQuote')
call Example()
```

```
A poet can but ill spare time for prose.
```

And look at what we do with our Funcref: because it refers to a function, we can use it in place of a function name. In the example, we use it with the :call command, which can take either a function name or a Funcref variable.

The call() function works like the :call command, and we can substitute a Funcref for a function name there, too. This function can also take arguments for us, in case our function (or the function that our Funcref refers to) requires them. We simply include the arguments as a List:

```
function! EchoQuote(quote, ...)
  let year = a:1
  let author = a:000[1]
  return 'In ' . year . ', ' . author . ' said: "' . a:quote . '"'
```

```
endfunction

let Example = function('EchoQuote')
let q = 'This crocodile mouth is the perfect helmet all the family will enjoy.'

echo call(Example, [quote, '2014', 'Dr. Carl Grommy'])
```

To get the name of the function that a Funcref references, we use string(). The String representation of a Funcref looks like what we write to assign one:

```
echo string(Example)

function('EchoQuote')
```

List

The List is a set of comma-separated items within square brackets. Items can be of any type, and built-in functions let us get, set, or remove items anywhere along the List:

intro/list.vim

```
let animalKingdom = ['Crocodile', 'Lizard', 'Bug', 'Squid']
echo animalKingdom
" → ['Crocodile', 'Lizard', 'Bug', 'Squid']

call add(animalKingdom, 'Penguin')
echo animalKingdom
" → ['Crocodile', 'Lizard', 'Bug', 'Squid', 'Penguin']

call remove(animalKingdom, 3)
call insert(animalKingdom, 'Octopus', 3)
echo animalKingdom[3]
" → Octopus

echo animalKingdom
" → ['Crocodile', 'Lizard', 'Bug', 'Octopus', 'Penguin']
```

All of these commands modify the original List—for example, when we call sort() before echoing a List, watch what happens:

```
let animalKingdom = ['Crocodile', 'Lizard', 'Bug', 'Octopus', 'Penguin']
echo animalKingdom
" → ['Crocodile', 'Lizard', 'Bug', 'Octopus', 'Penguin']

echo sort(animalKingdom)
" → ['Bug', 'Crocodile', 'Lizard', 'Octopus', 'Penguin']

echo animalKingdom
" → ['Bug', 'Crocodile', 'Lizard', 'Octopus', 'Penguin']
```

If we want to instead modify a copy of the List, we have a couple of options. copy() makes a distinct copy of the List, but with the original items—that is, if we were to add or remove from the copy, the original would be unchanged, but if we were to modify the items in the copy, that would affect the items in the original. The other option is deepcopy(), which makes a *full copy* of the List, including distinct items.

```
echo sort(copy(animalKingdom))
" → ['Bug', 'Crocodile', 'Lizard', 'Octopus', 'Penguin']

echo animalKingdom
" → ['Crocodile', 'Lizard', 'Bug', 'Octopus', 'Penguin']
```

We can get a sublist, or a *slice* of the List, by using [:] to specify the first and last items we want. To get the first three items of a List, for example, we could do this:

intro/list.vim

```
let animalKingdom = ['Frog', 'Rat', 'Crocodile', 'Lizard', 'Bug', 'Octopus',
                   \ 'Penguin']
let forest = animalKingdom[0:2]

echo forest
" → ['Frog', 'Rat', 'Crocodile']
```

If we don't specify a starting item, the default is 0. So we could also have written this like so:

```
let forest = animalKingdom[:2]
```

And if we want to end our sublist on the last item, we can count from the end of the List with a negative number (in this case, -1).

```
let animalKingdom = ['Frog', 'Rat', 'Crocodile', 'Lizard', 'Bug', 'Octopus',
                   \ 'Penguin']
echo animalKingdom[2:-1]
" → ['Crocodile', 'Lizard', 'Bug', 'Octopus', 'Penguin']
```

Dictionary

A Dictionary is an unordered array of keys and values. To access an entry, we put its key within brackets:

intro/dictionary.vim

```
let scientists = {'Retxab': 'Alfred Clark', 'Nielk': 'Bill von Cook'}

echo scientists['Retxab']            " → Alfred Clark
```

Keys must be of type String (or Number, but Number keys are automatically converted to String). Values, on the other hand, can be of any type—even Dictionary.

```
let scientists = {'Retxab': {'Clark': 'Alfred', 'Stoner': 'Fred', 'Noggin': 'Brad'},
      \ 'Nielk': {'Whate': 'Robert', 'von Cook': 'Bill'}}

echo scientists['Retxab']['Stoner']  " → Fred
```

To add entries, we use let:

```
let scientists['Trhok'] = 'Squirt'
echo scientists.Trhok                " → Squirt
```

And as you can see, we can also use a dot notation to access an entry, as long as its key consists only of letters, numbers, and underscores (this won't work for an entry with a key containing whitespace).

Float

Float variables are floating-point numbers:

intro/float.vim

```
let flotation = 96.7
```

The built-in function str2float(), as its name suggests, converts a String value to a Float. Another function, float2nr(), converts a Float to a Number. And speaking of Float and Number, if we add variables of those two types together, the result is converted to a Float:

```
let no = 42 + 96.7
echo no          " → 138.7
echo type(no)    " → 5
```

Look at what we echo on the last line: type(no). The function type() takes a value or variable and returns a number from 0 to 5 depending on the value's type: 0 for a Number, 1 for a String, 2 for a Funcref, 3 for a List, 4 for a Dictionary, and 5 for a Float. To keep us from having to memorize these numbers and then compare a variable to them, the official recommendation from Vim's documentation is to compare our variable to a value of a known type. (See :help type().)

```
echo type(no) == type(1.5)           " → 1
```

The no variable is a Float, so this code returns 1 for true. 0 would be false:

```
let no = 12.5
echo type(no) == type("warysammy")   " → 0
```