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

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

CHAPTER 2

A Real Live Plugin

Have you ever added code to or edited your Vim configuration file, .vimrc? If so, you've written code in VimL. As you saw in the previous chapter, VimL largely consists of commands like we run on the Vim command line. A .vimrc, the traditional place to put customizations and user functions, is a VimL script file. Beyond simply editing our .vimrc, we can modularize our VimL code and make it easily distributable—either for our own use or to share with other Vim users—by packaging it as a Vim plugin.

As you saw in the previous chapter, a plugin can be as small as a single script that lives in the plugin directory. That's what we'll start with here. When we finish this chapter, we'll have a plugin that opens a new split window, calls mpc to get its playlist, and then displays the playlist in a new buffer.

But First, a Function

At the end of *The Structure of a Vim Plugin*, on page ?, we created our main plugin directory, mpc. This is where we'll be putting the different directories in which Vim looks for VimL source files.

Under mpc, create the plugin directory and then create a file under it called mpc.vim. It should look like this:

plugin/mpc/plugin/mpc.vim
```
function! OpenMPC()
  let cmd = "mpc --format '%title% (%artist%)' current"
  echomsg system(cmd)[:-2]
endfunction
```

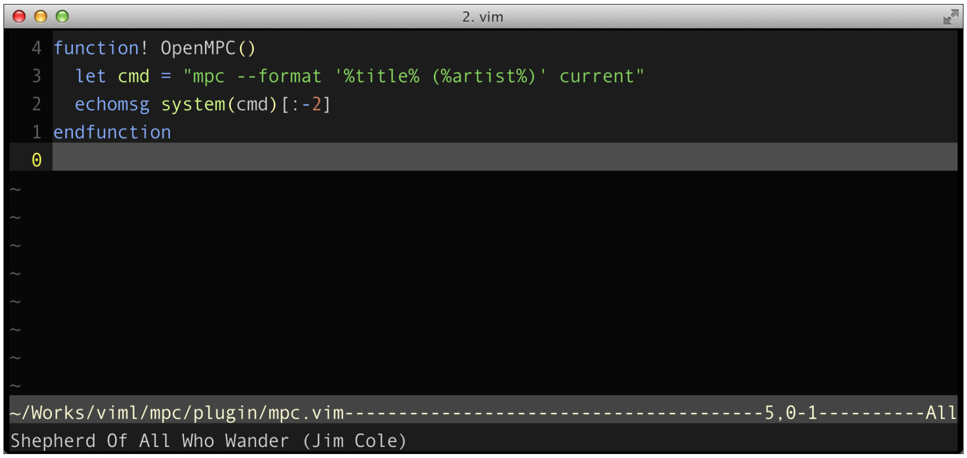
Because we appended the plugin directory to our runtimepath, Vim will load it automatically the next time we start it up. For now, though, save the file and then source it:

```
:source %
```

Now Vim should have OpenMPC() ready to go. Make sure that mpc is running and then, from the Vim command line, run this:

```
:call OpenMPC()
```

As you can see in the following figure, Vim will display a message containing the track that mpc is playing.



```
function! OpenMPC()
  let cmd = "mpc --format '%title% (%artist%)' current"
  echomsg system(cmd)[:-2]
endfunction

~
~
~
~
~
~
~
~
~/Works/viml/mpc/plugin/mpc.vim-------------------------------------5,0-1---------All
Shepherd Of All Who Wander (Jim Cole)
```

Running External Commands

Our OpenMPC() function gets the current track by running mpc current, with the --format argument. This is a shell command, and in VimL we can use the system() function to call shell commands.

system() works like Vim's bang command (:!). You might be familiar with using that command to execute a shell command from inside Vim:

```
:!date
Sat Oct 18 19:35:53 CDT 2014
Press ENTER or type command to continue
```

In like manner, system() takes a String command to run and executes it. It then returns the command's output to us as another String.

Now notice how we display the command's output. In the previous chapter we made a lot of use of :echo to echo messages to the user. In OpenMPC(), we're using another echoing command: :echomsg. :echomsg, unlike :echo, saves its messages in Vim's message history. Run :messages to see the history—if you've just recently started Vim, you should see something like this:

```
Messages maintainer: Bram Moolenaar <Bram@vim.org>
".../mpc/plugin/mpc.vim" [New] 6L, 131C written
Shepherd Of All Who Wander (Jim Cole)
```

This message-saving is really the main difference between :echo and :echomsg. There's another interesting difference: :echomsg only takes String messages. If we try to give it a List, say, we'll get an error:

```
let numbers = [1, 2, 3]
echomsg numbers
```

```
E730: using List as a String
```

That's easy to get around—we can just use Vim's handy built-in string() function, which returns String versions of whatever other-type values we give it:

```
let numbers = [1, 2, 3]
echomsg string(numbers)   " → [1, 2, 3]
```

And lastly, notice the actual String that we're passing to :echomsg. Because mpc is a shell command, we get a newline character appended to its output before we get that output. This is good if we're at the command line, but for our purposes in Vim, we just want the single line that describes the currently playing track.

This brings us to the final difference between :echo and :echomsg. Instead of interpreting what Vim refers to as *unprintable* characters like the newline character, which is what :echo does, :echomsg translates them to something *printable* and displays them as part of the String. If we just gave the result of the system() call to :echomsg, we would get something like this:

```
echomsg system(cmd)   " → Shepherd Of All Who Wander (Jim Cole)^@
```

To remove that newline character, we instead give :echomsg a substring. The syntax for getting a substring is identical to what we use for List slicing, as you saw when we talked about the *List*, on page ?. When we want a substring of a String, we use [:] to specify the substring's beginning and ending bytes. Remember that if we don't supply a first number, 0 is the default. And just as with a List, we can use negative numbers to count from the end of the original value:

```
let professor = "Brad Noggin"
echomsg professor[5:-1]         " → Noggin
```

So in the OpenMPC() function, the following line tells Vim to echo everything up to the second-to-last character of the result from system(cmd).

```
echomsg system(cmd)[:-2]
```

That gives us the single line of output from mpc current.

Joe asks:

Is It echomsg or echom?

When you're reading VimL code out in the wild, you'll frequently see people using shortened versions of the various keywords and commands. Most commands have abbreviated forms, and you can use anything from the shortest possible abbreviation to the complete keyword. The documentation shows the shortest possible form and then the remaining characters inside brackets:

```
:echom[sg] {expr1} .. Echo the expression(s) as a true message, saving the message
                      in the message-history.
```

:echomsg is a good example of this; you'll usually see it written as echom. I think that some of the shortened forms accidentally prove very fitting—for example, I'm a minor fan of writing functions like so:

```
fun ForExample()
  echomsg "VimL *is* fun!"
endfun
```

Here, I could've written fun as func and then ended the function with endf. The choice of whether to use abbreviated forms or complete keywords comes down to preference, but to keep our code as readable as possible and to minimize confusion, we'll be sticking with the full keywords in this book.

Writing Text to a Buffer

Let's now expand OpenMPC() to display the entire playlist from mpc. For now, we'll have the function call mpc to get the playlist and then display that in a new split window. Modify the code to look like this:

plugin.1/mpc/plugin/mpc.vim

```
function! OpenMPC()
  let cmd = "mpc --format '%position% %artist% / %album% / %title%' playlist"
  let playlist = split(system(cmd), '\n')

  new

  for track in playlist
    if(playlist[0] == track)
      execute "normal! I" . track
    else
      call append(line('$'), track)
    endif
  endfor
endfunction
```

And now let's quickly go over this before we try it out.

On line 3 we define a List, playlist, to store the result of our mpc call. We assign it the output from that command, split by newlines. Then we open a new window using the :new command, which starts out its new window with a blank file. After that is where things (as they say) start to get interesting.

Once we've opened the window, we loop through each track in playlist (in lines 7 through 13). To see whether we've started outputting the list, we compare the track we're on to the first item (on line 8), and if it's the first item, we make use of a fascinating Vim command: :execute.

The :execute command takes a String and executes it as an Ex command. (If you're looking to get into metaprogramming in VimL, :execute isn't a bad place to start.) We're using :execute to call the :normal command, which itself takes a String of *normal mode* commands and runs them. By combining :normal and :execute, like we do on line 9, we can script what would've been our manual interaction with a Vim buffer. In this case, we run the normal-mode command I, which enters insert mode at the beginning of the line, and then enter the text of the track.

Again, note the bang (!) appended to the :normal command. This is important: when we run a :normal command that the user has remapped, the bang works the same way that it does for a function declaration, and Vim will use the command's unmapped default. For example, if our user had for some reason set up I to run :q!, :normal! would ignore that odd (if creative mapping) and enter insert mode at the beginning of the current line, as we would expect.

If we've already entered the first track, we call the built-in function append() to enter the rest. This function *appends* the text we give it to a buffer after a certain line in the file—it'd be like using the p command in normal mode. It takes two arguments: a line number and a String to append below the line of that number. We're giving append() the line number of the last line in the buffer, using another built-in function, line().

The line() function takes a *file position* and returns the line number of that position in the current file. (For the full list of file positions, see :help line().) We can use this to get the line number of a mark:

```
33G
ma
:echo line("'a")    " → 33
```

We can get the number of the first (or last) visible line in the file's window:

```
44G
:echo line("w0")   " → 16
:echo line("w$")   " → 44
```

We can get the line number of the current cursor position:

```
22G
:echo line(".")     " → 22
```

We can also get the last line in the current file:

```
:new
:echo line("$")   " → 1
```

For each of the remaining tracks in the playlist, we use this to append the text of the track to the buffer, and then our function ends.

One note about how we're doing this: append() *can* take a String value to append, but it can also take a List. If we give it a List, it will go through that List and append each item in turn. This means that we could've set up our function like so:

plugin.1/alternate.vim

```
function! OpenMPC()
  let cmd = "mpc --format '%position% %artist% / %album% / %title%' playlist"
  let playlist = split(system(cmd), '\n')

  new
  call append(0, playlist)
endfunction
```

And when we give 0 to append() as the line number, it actually *prepends* the text to the buffer, or puts it before line 1, the first line. Cool, right? There wasn't really any compelling reason to not write it like this; I just wanted to show you the coolness that is :normal combined with :execute.

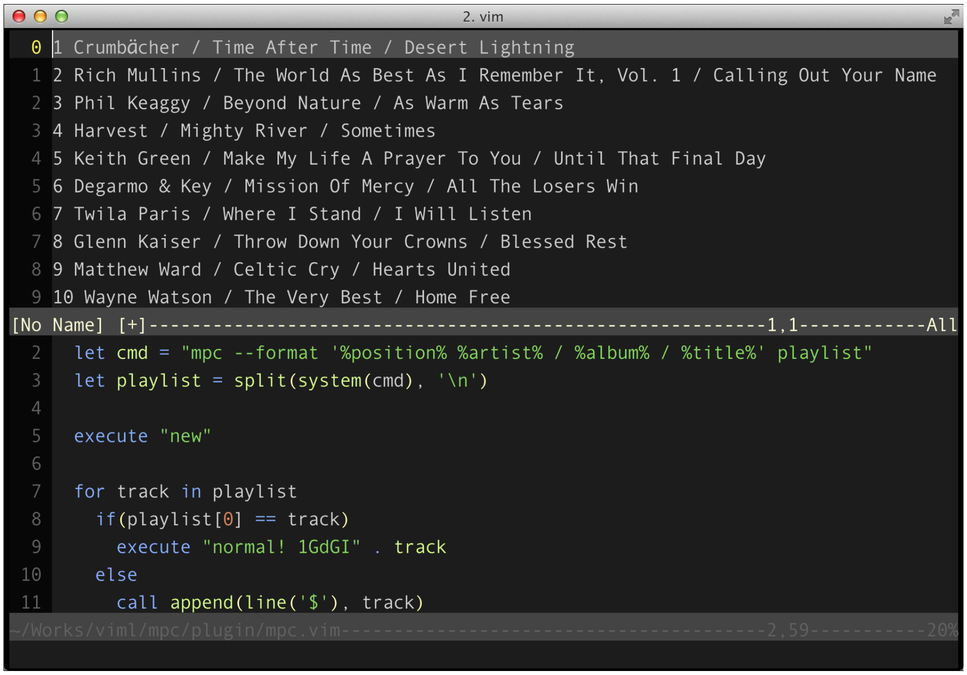
So now that we have OpenMPC() ready and we've seen what the new code is doing, let's give this the proverbial whirl. Save the plugin file and run our trusty :source command:

```
:source %
```

Then call the function:

```
:call OpenMPC()
```

You should see something like what we have in the following figure.



```
2. vim
1 Crumbächer / Time After Time / Desert Lightning
2 Rich Mullins / The World As Best As I Remember It, Vol. 1 / Calling Out Your Name
3 Phil Keaggy / Beyond Nature / As Warm As Tears
4 Harvest / Mighty River / Sometimes
5 Keith Green / Make My Life A Prayer To You / Until That Final Day
6 Degarmo & Key / Mission Of Mercy / All The Losers Win
7 Twila Paris / Where I Stand / I Will Listen
8 Glenn Kaiser / Throw Down Your Crowns / Blessed Rest
9 Matthew Ward / Celtic Cry / Hearts United
10 Wayne Watson / The Very Best / Home Free
[No Name] [+]----------------------------------------------------------1,1------------All
  let cmd = "mpc --format '%position% %artist% / %album% / %title%' playlist"
  let playlist = split(system(cmd), '\n')

  execute "new"

  for track in playlist
    if(playlist[0] == track)
      execute "normal! 1GdGI" . track
    else
      call append(line('$'), track)
~/Works/viml/mpc/plugin/mpc.vim---------------------------------------2,59-----------20%
```

Our plugin is still in its early stages, but we now have a basis to build on as we continue to learn about VimL. We have a function that interacts with the system, opens a new buffer, and runs normal-mode commands to manage that buffer. This is all in a single script file that could have been in our .vimrc, but what we have now is portable; another Vim user could add this functionality to a Vim installation just by dropping the file into the plugin directory.

In the next chapter, you'll discover the autoload mechanism—the autoload directory is where we'll be keeping the bulk of our plugin's functionality. Among other uses, the autoload system helps us keep our plugin code organized. We'll continue working with the operating system and mpc to make use of our newly displayable playlist.