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

Early Praise for *The VimL Primer*

Ben's book is an eye-opener: I've used Vim for years but ignored the power and flexibility it offers. Now I'm paying attention. *The VimL Primer* is a gentle, thoughtful introduction to a new world for Vim users.

➤ **Michael Easter**
Software developer, ScreenScape Networks

Vim is an incredibly useful tool in any developer's toolkit, and Ben Klein offers an easy-to-read, helpful, and at-times-witty guide to scripting it with VimL. A must-read for all Vim-using developers.

➤ **Joshua Scott**
Managing partner, Resonant Media Technologies, LLC

The VimL Primer gets straight to the point and shows you the ropes for dealing with Vim plugins. Much like Vim itself, this book communicates a lot of detail efficiently and effectively. This book can help you take your Vim skills to the next level.

➤ **Kevin Munc**
Founder, Method Up LLC

The VimL Primer does an incredible job of showing you how to take one of the most enduring text editors and extend it so that it becomes even more useful. Do you want to bend Vim to your will? This is where you start!

➤ **Jared Richardson**
Principal consultant, Agile Artisans, Inc.

With Drew Neil's *Practical Vim*, you've mastered all the magic of Vim, but with Ben Klein's fast-paced *VimL Primer*, it's time you learned how to write your own spells and plugins, with the VimL language wand, like a pro!

➤ **Guillaume Laforge**
Groovy project lead

Everything you need to start working on the next popular Vim plugin.

➤ **Mac Liaw**
CTO, CylaTech.com, Inc.

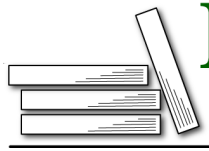
The VimL Primer

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: P1.1—February 2015

Contents

An Introduction

The World's Shortest History Lesson

If you've used Vim for more than a couple of minutes, you're probably familiar with at least a few of its commands. To save a file, you run `:w`. When you want to exit the editor, you run `:q`.

Commands such as `:w` and `:q` are called *Ex commands* because they originated in *Ex*, the line-based editor. Pioneering computer scientist Bill Joy invented *vi*, a visual mode for Ex, in 1975. In the years since, vi has inspired newer editors and has been ported several times. Of these, the most popular and perhaps the most enduring editor is the one I'm using to write this book: Vim.

In Vim, Ex commands can be run on the command line, but they also make up the bulk of Vim's built-in scripting language, *VimL*. Recent Vim versions (notably, version 7) have added data types, functions, and many other common language features that together turn VimL into a highly capable scripting language. In this book, you'll learn how to work with VimL.

Who Should Read This Book

This book is for Vim users who want to get started with VimL. I assume that you're familiar with how to use Vim for basic text editing. You don't have to be an expert Vim user, because this is an introduction, after all. VimL is not a wildly advanced topic if you're already comfortable with the editor. You just need to know your way around.

If you're not familiar with Vim buffers, windows, the command line, and modes, I suggest you try the Vim tutorial first.[1] It's a splendid interactive tutorial for first-time Vim users.

1. http://vimdoc.sourceforge.net/htmldoc/usr_01.html#01.3

This is also not a book on advanced Vim usage. For users looking to advance their Vim editing skills, I recommend Drew Neil's *Practical Vim: Edit Text at the Speed of Thought [Nei12]*.

How to Read This Book

We start with Chapter 1, *The Lay of the Land*, on page 1, where you learn the basics of the VimL language. In Chapter 2, *A Real Live Plugin*, on page 15, we put that knowledge to use by starting a new project: creating a Vim plugin. In each following chapter you learn new aspects of Vim scripting and build on this project. In Chapter 3, *The Autoload System*, on page 23, we take advantage of the autoload facility. In Chapter 4, *Recognizing File Types*, on page 31, we talk about automatically executed commands and how to detect a filetype. In Chapter 5, *Highlighting Syntax*, on page 41, we tell Vim about our own syntax rules and how to highlight syntax. Finally, in Chapter 6, *Commands and Mappings*, on page 53, you learn how to write your own commands to run in the Vim command line and how to map keys to your plugin's functionality.

Because of its project-based format, this is not a good book for skimming or jumping around. It's also not a good book to just read straight through. The goal is to code along! When we get to the end, you'll not just have read through an introductory textbook—you'll have written a fully functional Vim plugin.

Online Resources

You can download the code from this book from the Pragmatic Bookshelf website.[2] Click on the *Source code* link on the book's page. The code is broken into at least one directory for each chapter; the `intro` directory contains the example code from the first chapter, and the other directories contain the example code from each chapter's version of the plugin.

Also on the book's website is the forum. Click *Discuss* on the book's page to ask questions, make comments, or just discuss the book with fellow readers and me. You'll also find a link to report errata; if you come across a problem in the code, something that's not explained clearly enough, or even a typo, head over there to report it.

And with that, let's be off!

2. http://pragprog.com/book/bkviml/the-viml-primer

Acknowledgments

Thanks to the entire Pragmatic Bookshelf team for letting me work with them. Dave Thomas, Andy Hunt, Susannah Pfalzer, Fahmida Y. Rashid, and all of the rest that I've worked with have been unreasonably patient and very kind; particular thanks go to Susannah for answering all of my many questions, to Dave for bearing with my explanations of some of the joyous oddities in VimL syntax, and especially to Fahmida for keeping this project on track in spite of my constant mangling of schedules.

Many thanks to the world's greatest technical reviewers—Barry Arthur, John Cater, Ingo Karkat, Guillaume Laforge, William LaFrance, Mac Liaw, and Jared Richardson—for detailed insights into the code, as well as general high-level suggestions. Thanks also to Nathan Neff (the greatest programmer in the world) for his multitude of suggestions and corrections; to Christopher Coleman, Michael Easter, Kevin Munc, and Josh Scott for reading and for encouraging me through the writing process; and to Tim Berglund, Chad Fowler, Scott Minnich, Paul Nelson, and Kurt Wise for encouraging me, sometimes wittingly and sometimes probably not, in this effort.

Extreme thanks go to my family, who played the part of foremost supporters even while I spent so much time unavailable, working on the book. To Dad and Mom (Dave and Debbie) and to Zak and Beth, Abi, Sarah, Solomon, Hannah, Joanna, Rebekah, Susannah, Noah, Samuel, Gideon, Joshua, and Daniel: thanks for asking how it was going, listening to my spontaneous lectures on the writing process, assuring me that I would one day finish, and everything in between. I love you all.

And thanks most of all to my Creator, Jesus Christ. A faithful saying still worthy of all acceptation is that He came into the world to save sinners, of whom I am chief (the apostle Paul has since moved on). To Him be glory.[3]

3. 1 Timothy 1:15

CHAPTER 1

The Lay of the Land

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, but our functions have to begin with uppercase letters in order 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. (We show output using the Vim comment syntax, followed by an arrow.)

```
:call EchoQuote()
" → A poet can but ill spare time for prose.
```

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 of code, 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, written using prefixes. In our last function, where we wrote variables with the a: prefix (as in a:quote), we were using *argument scope*, used for function arguments. Two others are the *global scope*, which is the default scope, and the *local 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 (local to a function). The scope is marked by the prefix, just like variables in the argument scope use the a: prefix.

The local 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-local prefix, such as l:quote. (For 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. For more on variable scopes, see :help internal-variables.)

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, [q, '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
```

Loops and Comparisons

VimL has a while loop. It starts with while and a condition, and it ends with endwhile:

intro/loop.vim

```
let animalKingdom = ['Crocodile', 'Bug', 'Octopus', 'Penguin']

while len(animalKingdom) > 0
  echo animalKingdom[0] . ' Friend'
  call remove(animalKingdom, 0)
endwhile

" → Crocodile Friend
"   Bug Friend
"   Octopus Friend
"   Penguin Friend
```

The condition here, len(animalKingdom) > 0, checks that the size of animalKingdom is greater than 0. To do that, it uses len(). On a List, this function returns the number of items. It can also be used to get the length of a String value, but there's a dedicated function, strlen(), for that.

In the body of the while loop, the first statement, echo animalKingdom[0] . ' Friend', echoes a value to the user, based on the current first item in animalKingdom. The second statement removes that value using remove().

Iterating with for

A for loop in VimL, similar to a while loop, starts with for and then a variable name, in a List. It ends with endfor.

intro/loop.vim

```
let scientists = ['Robert Whate', 'Bill Cook', 'Brad Noggin', 'Squirt']

for scientist in scientists
  echo 'Dr. ' . scientist
endfor

" → Dr. Robert Whate
"   Dr. Bill Cook
"   Dr. Brad Noggin
"   Dr. Squirt
```

The Degrees of Equality

We check between a series of conditions using an if statement, which starts with if and ends with endif. To check for multiple specific conditions, we use an else statement or an elseif.

intro/comparison.vim

```
let bees = 32
let mice = 4

if bees < 1
  echo 'I suppose the mice keep the bees out--'
elseif mice < 1
  echo '--or the bees keep the mice out.'
else
  echo 'I don''t know which.'
endif
```

The == operator has a bit of a gotcha in VimL: its behavior depends on the user's setting of ignorecase, an option that tells Vim whether to ignore case in commands and search expressions. This means that we have to be careful about using the operator in scripts that we intend for more than our own Vim instance.

Let's say that our user has ignorecase turned on:

```
set ignorecase

let farewell = 'We love you. Ebenezer!'
echo toupper(farewell)                    " → WE LOVE YOU. EBENEZER!
```

The toupper() gives us an all-uppercase version of the mixed-case variable farewell.

```
function! CheckCase(normal, upper)
  return a:normal == a:upper ? 'Equal.' : 'Not equal.'
endfunction
```

Here we have a function called CheckCase(). It uses a *ternary expression* to tell us whether its two String arguments are equal. If the expression a:normal == a:upper evaluates to 1 true, the function will return the String following the ?. If it evaluates to 0 false, we'll get the String after the : instead.

What happens when our user calls CheckCase()?

```
:echo CheckCase(farewell, toupper(farewell))  " → Equal.
```

On the other hand, we've told Vim *not* to ignore case:

```
:set noignorecase
:echo CheckCase(farewell, toupper(farewell))  " → Not equal.
```

You might think that this behavior would make == useless in practice; it doesn't, really, because (for example) when we compare Number values, we don't care about case sensitivity. But to be safe, when we're dealing with String values it's best to stick with one of VimL's two more specific equality operators: ==# is always case sensitive, and ==? is never case sensitive.

intro/comparison.vim

```
" Compares values using ==#
function! CheckCaseSensitive(normal, upper)
  if a:normal ==# a:upper
    return 'Equal (case sensitive).'
  else
    return 'Not equal (case sensitive).'
  endif
endfunction

" Compares values using ==?
function! CheckCaseInsensitive(normal, upper)
  if a:normal ==? a:upper
    return 'Equal (case insensitive).'
  else
    return 'Not equal (case insensitive).'
  endif
endfunction
```

The functions CheckCaseSensitive() and CheckCaseInsensitive() are more reliable for use with String values:

```
let farewell = 'We love you. Ebenezer!'
let response = 'Will you stop that!'

:echo CheckCaseSensitive(farewell, toupper(farewell))
" → Not equal (case sensitive).

:echo CheckCaseInsensitive(farewell, toupper(farewell))
" → Equal (case insensitive).

:echo CheckCaseInsensitive(farewell, response)
" → Not equal (case insensitive).
```

Our Project: An Interface for mpc

Now that you have an understanding of basic VimL syntax, we can put that understanding to use. In the next chapters we're going to use VimL to write a Vim plugin: a Vim interface for mpc, the command-line client for the Music Player Daemon (MPD).

MPD is a music-playing server. It keeps a flat file database of audio files which client applications use to organize and play the music. Our Vim plugin will

interact with mpc, a command-line client for mpd, to let us view and play tracks in MPD's playlist from within a Vim window.

Setting Up MPD and mpc

First, you need to download and install MPD and mpc.[1,2] (If you're on OS X, you can install both using Homebrew,[3] the package manager for OS X; run brew install mpd for MPD and then brew install mpc to get mpc.)

Once you have MPD and mpc installed, you'll need to set up the database and the configuration file. Create a new directory and place some audio files in it. Then, in your main user directory, create the file .mpdconf. In Windows, that file should be mpd.conf. This is what I have in that file:

```
bind_to_address         "127.0.0.1"
music_directory         "/Users/ebenezer/music"
db_file                 "/Users/ebenezer/path/to/mpd.db"
audio_output {
  name "audio"
  type "osx"
}
```

The music_directory should have the full path to that directory containing the audio files. Note that if you're on Linux, you can substitute the audio_output value with alsa instead of osx. If you're on Windows, try winmm.

Create the database file db_file by running the following command, substituting the path that you used for db_file:

```
touch /Users/ebenezer/path/to/mpd.db
```

On Windows, the command would be type nul > C:\path\to\mpd.db.

You should now be able to run mpd at the command line to start MPD. After doing that, run the following commands:

```
mpc update
mpc ls | mpc add
```

That will add the music directory's files to the MPD playlist.

The Structure of a Vim Plugin

Vim reads VimL files from several different directories under its home directory, which will be kept in your user directory. On OS X and Linux it looks

1. http://www.musicpd.org/download.html
2. http://www.musicpd.org/clients/mpc/
3. http://brew.sh/

for these directories under .vim/; on Windows they live in a directory called vimfiles.

- plugin: This is the main directory for plugin script files. A Vim plugin can be as small as a single file that lives in this directory.
- autoload: The autoload directory stores VimL script files that are loaded on demand. You'll learn about Vim's autoload system in Chapter 3, *The Autoload System*, on page 23.
- ftdetect: This is where we place VimL files that detect the type of file Vim is editing.
- ftplugin: Almost like plugin, this directory is used mainly by *filetype plugins*. The code in ftplugin files is used only on files of a particular filetype. In Chapter 4, *Recognizing File Types*, on page 31, we'll take advantage of this to recognize a filetype of our own.
- syntax: Vim syntax files, like the files in ftplugin, are specific to a filetype, and they describe the syntax elements of that filetype. Syntax files are stored in this directory; we'll work with a syntax file in Chapter 5, *Highlighting Syntax*, on page 41.
- doc: This is the home of plugin documentation files. Vim help files have their own special syntax and stylistic standards, but they're stored as plain text. When we write a help file for a plugin, it goes under doc.

A plugin can use any or all of these directories. The default way to install a plugin is to copy each of its files to the correct directory, but as you install more and more plugins, this situation quickly becomes hard to maintain (especially when it comes to upgrading or uninstalling plugins). To make this process easier, developers have written systems such as Pathogen and Vundle,[4] [5] which allow each plugin's directories to be stored separately. Then, for example, instead of a single autoload directory holding every plugin's autoloaded files, each plugin has its own directory and its own autoload subdirectory.

We'll take an approach similar to Vundle's or Pathogen's. To keep things simple for our purposes, we'll just add our plugin's directory to Vim's runtimepath. The runtimepath is a Vim option that is set to a list of directories, and when Vim starts up, it looks through each of these directories for script files to load. If we create a project directory that uses the directory structure above

4. https://github.com/tpope/vim-pathogen
5. https://github.com/gmarik/Vundle.vim

and then add it to this option's list, Vim will load our plugin directory's files on startup.

We'll create our main plugin directory first—we'll call it mpc. If you are on OS X or Linux, add the following line in your .vimrc file. On Windows, the file would be _vimrc:

```
set runtimepath+=/full/path/to/plugin/directory/
```

You'll have to restart Vim for that change to kick in. Once you've done that, you're ready to begin! By the end of the next chapter, we'll have a working Vim plugin.

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 12, 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
```

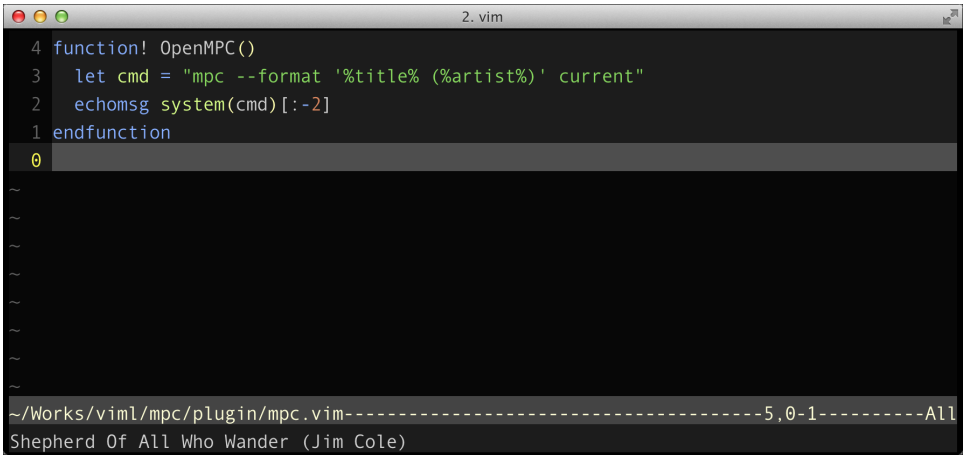
Because we appended the plugin directory to our runtimepath, Vim will load it automatically the next time we start it up. For now, though, save the file and then source it:

```
: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.



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 6. 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

```
Line 1  function! OpenMPC()
    -     let cmd = "mpc --format '%position% %artist% / %album% / %title%' playlist"
    -     let playlist = split(system(cmd), '\n')
    -
    5     new
    -
    -     for track in playlist
    -       if(playlist[0] == track)
    -         execute "normal! I" . track
   10       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.

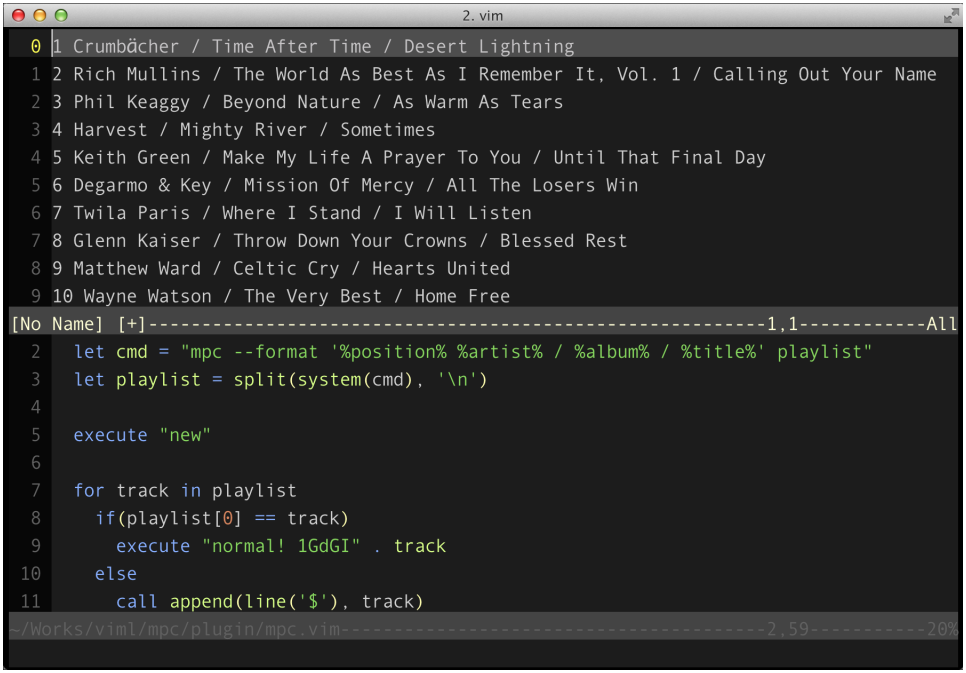
So now that we have OpenMPC() ready and we've seen what the new code is doing, let's give this the proverbial whirl. Save the plugin file and run our trusty :source command:

```
:source %
```

Then call the function:

```
:call OpenMPC()
```

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



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.

CHAPTER 3

The Autoload System

Vim's autoload system lets us easily break out our plugin's code into manageable scripts with reusable functions. There are other ways to have multiple files working together in a plugin and reuse our code, but they can get rickety. The autoload system is specifically designed for this purpose, and it also keeps our plugin's functions from colliding with any similarly named others—in that regard we can use it to provide a form of *namespace* for our functions.

We're going to start this chapter by seeing how to take advantage of autoload—and as a bonus, we'll make our plugin a bit smarter while we're at it. Then we'll see how we can get text from the playlist buffer so that users are able to play a selected song from the playlist.

Autoloading Functions

The main point of autoload is to make it easy to use reusable functions. To use a function from more than one script file, we place it in a script file under the `autoload` directory. Vim calls this kind of file a *library script*. We can call the functions in that file from anywhere else in the plugin.

There's a special syntax for calling autoloaded functions. As an example, if we had a file called `mpc.vim` in the `autoload` directory and it contained the function `FromAutoload()`, this is how we would call `FromAutoload()` from another function:

```
function! ForExample()
  call mpc#FromAutoload()
endfunction
```

Everything before the `#` represents a part of the path to the `mpc.vim` file; the last piece before the `#` is the filename minus the `.vim` extension. Here we're only including `mpc#`—the filename. Vim understands that this means it has to look for the file `mpc.vim` under `autoload` and then call that file's `FromAutoload()` function.

We can have our own subdirectories and multiple files under autoload. We just have to make sure that when we call an autoloaded function, the first name we include before the function's name is either the directory under autoload or the filename, and that the last name is the filename. In the preceding example, we have only one piece, which is obviously both first and last. Here's how we would call FromAutoload() if the autoloaded file were in a subdirectory called mpd:

```
function! ForExample()
  call mpd#mpc#FromAutoload()
endfunction
```

Letting us reuse code isn't the only thing the autoload system does. When we put a function in a file under the autoload directory, Vim waits to load that function until the user (or another script) calls it. In contrast, code under plugin and other directories is loaded whenever Vim starts.

We can see this in that last example. When a user starts up Vim and calls ForExample(), FromAutoload() isn't defined. So Vim, following the path that we gave it, looks in the autoload directory, finds FromAutoload() inside the mpc.vim file, and loads it. In larger plugins, this is preferable to putting all of our code under the plugin directory; not trying to load all of a plugin's source right away can keep a plugin from being too hard on Vim's memory and startup time.

So let's see how we can use the autoload system in our mpc plugin. To start with, we'll need an autoload directory under our main plugin directory—if you haven't created that directory yet, do that now. Create the file mpc.vim and save it in the directory. This is how the file should look:

autoload/mpc/autoload/mpc.vim

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

  for track in playlist
    if(playlist[0] == track)
      execute "normal! 1GdGI" . track
    else
      call append(line('$'), track)
    endif
  endfor
endfunction
```

You might recognize this code—it's the bulk of code in our original OpenMPC() function in plugin/mpc.vim. Go back to plugin/mpc.vim and replace the entire OpenMPC() body with the following two lines:

autoload/mpc/plugin/mpc.vim
```
execute "new"
call mpc#DisplayPlaylist()
```

And now the original plugin file should have nearly disappeared; this should be all that's left:

```
function! OpenMPC()
  execute "new"
  call mpc#DisplayPlaylist()
endfunction
```

Now we have the playlist-loading code moved to an autoloaded file, but we need to tweak that code a bit. We have a problem with the way we're opening the playlist window.

Finding Windows by Buffers

The mpc#DisplayPlaylist() code that we just copied from the original OpenMPC() opens a new window, and then loses track of it. The next time the function is called, it opens *another* window.

To see this in action, open the editor and then start the plugin with :call OpenMPC(). Run that command a few times: you'll see a new window open every time the function calls mpc#DisplayPlaylist(). The code doesn't check for or switch to any window that's already opened to our playlist.

Let's remedy this. We'll change mpc#DisplayPlaylist() so that if we already have a playlist window open when we call OpenMPC(), we switch to the open playlist from the current window. Because the job of mpc#DisplayPlaylist() is only to display the playlist, we'll also move the window-managing code back to OpenMPC().

Modify OpenMPC() to look like this:

```
function! OpenMPC()
  if(bufexists('mpc.mpdv'))
    let mpcwin = bufwinnr('mpc.mpdv')
    if(mpcwin == -1)
      execute "sbuffer " . bufnr('mpc.mpdv')
    else
      execute mpcwin . 'wincmd w'
      return
    endif
  else
    execute "new mpc.mpdv"
  endif
  call mpc#DisplayPlaylist()
endfunction
```

We'll go over this one piece at a time, but before we do, let's try running :call OpenMPC() a few times again. After each call, switch to the window you were in before. Once you've called the function one time, you should see Vim switch focus back and forth, from that window to the playlist window it opened the first time.

Now we have a specific window for our function to use. Let's look at our code again to see how we're handling that window.

The Built-in Buffer Functions

When we finished with OpenMPC() at the end of Chapter 2, *A Real Live Plugin*, on page 15, we opened a window with the :new, which simply opens a new window and immediately moves to it. When we run this command, the window we open displays the buffer name *[No Name]*, because it *has* no name—we've opened a new window to nothing.

In our updated OpenMPC(), if we've already opened a window, we reuse it. The :new command now opens a named buffer in the new window, and from then on we can refer to that window by the name of the buffer to which it's opened.

So here's our new process for opening the playlist window. We start by checking for a buffer with the buffer name we've devised:

```
function! OpenMPC()
  if(bufexists('mpc.mpdv'))
    let mpcwin = bufwinnr('mpc.mpdv')
```

bufexists() takes a buffer name and returns 1 if there's a buffer with that name. If there is, we assign the variable mpcwin to the window that's open to that buffer. We get the window's number by calling another function, bufwinnr(), and giving it the buffer name. If there's a window open to the buffer, we'll get its number. If there isn't, bufwinnr() will return -1, so we check for that next:

```
if(mpcwin == -1)
  execute "sbuffer " . bufnr('mpc.mpdv')
else
  execute mpcwin . 'wincmd w'
  return
endif
```

Remember that at this point we know that our buffer exists, so we *have* opened the playlist window before. If there's no window open to the playlist buffer, we use :execute to call the :sbuffer command, which opens a new split window to the buffer whose number we give it. We get that buffer number from another built-in function, bufnr().

bufnr() takes a buffer name and returns that buffer's number. If our buffer's number were 2, this command would be the same as if we ran the following:

```
:new
:buffer 2
```

If we already have a window open to the buffer, we use the :wincmd command. Think of it as the command form of running Ctrl-w with an argument. Ctrl-w k moves us to the window above the current window; :wincmd k does the same thing. Ctrl-w w takes a number and goes to the window of that number, so 1 Ctrl-w w would take us to window number 1.

```
execute mpcwin . 'wincmd w'
```

This code simply says in VimL that we want to execute the command :wincmd w with the number of our buffer's window. Then, since we just needed to switch windows, we return out of the function.

```
  else
    execute "new mpc.mpdv"
  endif
  call mpc#DisplayPlaylist()
endfunction
```

Finally, if we have *not* opened our buffer yet, we run :execute new with a buffer name. This will open that buffer in a new window with that name, and we can now switch to that window the next time the user calls this function. Then, at the end of the function—unless the window was previously opened—we call mpc#DisplayPlaylist(), which loads the playlist.

Retrieving the Text of a Line

We now have the playlist displayed in a window we can easily open. Let's add the ability to play a specific song chosen by the user.

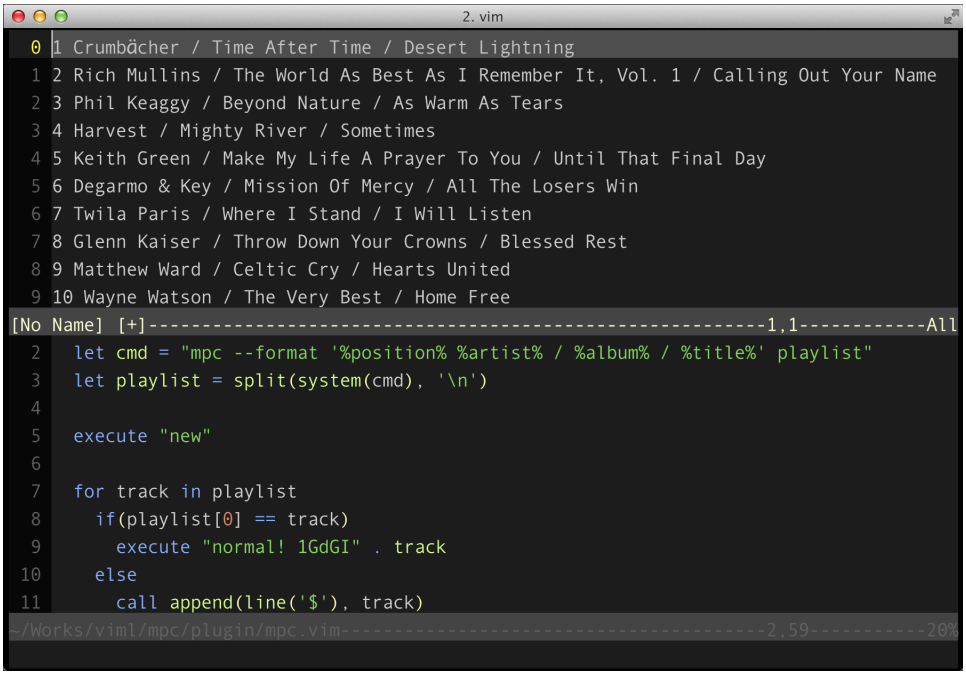
We're going to create a new function, mpc#PlaySong(), within our autoload directory. It will take the number of a song in the playlist—what mpc refers to as the song's position—and send that number back to mpc using system(). Add the following code to autoload/mpc.vim:

autoload.1/mpc/autoload/mpc.vim

```
Line 1  function! mpc#PlaySong(no)
    2    let song = split(getline(a:no), " ")
    3    let results = split(system("mpc --format '%title% (%artist%)' play "
    4          \ . song[0]), "\n")
    5    let message = '[mpc] NOW PLAYING: ' . results[0]
    6    echomsg message
    7  endfunction
```

There is a fair bit happening here. mpc#PlaySong() takes one argument, no, which actually represents a line in the playlist buffer. Vim's function getline() takes a number and returns the contents of the line by that number. On line 2 we call getline() on no and then split the resulting line contents into a List, with items delimited by spaces. We assign that List to the variable song.

To make this clearer, let's look at an example from the playlist window we saw in the previous chapter. Here it is:



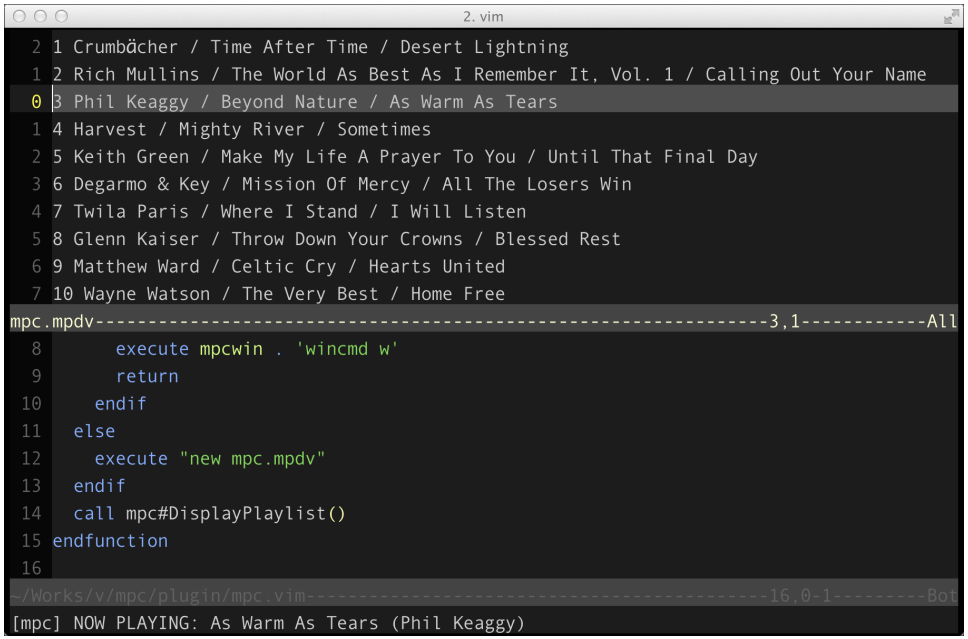
The last track in the window is number 10 in the playlist, so it has the position 10. If we were to call mpc#PlaySong(10), this would be the song we would get:

```
['10', 'Wayne', 'Watson', '/', 'The', 'Very', 'Best', '/', 'Home', 'Free']
```

And as you can see, this also gets the item separators, /. No matter—the important thing is the first item in the List: the position (10).

So next, on line 3, we use system() to call mpc play, passing it that first item in the song List, the position. We call split() again, this time on the output we get from mpc when we play the song, and assign that—another List—to the variable results. So results is a List of mpc's output, broken up by new lines. We finish on lines 5 and 6 by echoing a "now playing" message, which contains the first item (or line) of results, to the user.

Try it out. Open Vim and run :call OpenMPC(). Then run :call mpc#PlaySong(3). If MPD is running, you should hear the third track in the playlist starting up and see something like what's shown in the figure below.



No, calling the function manually isn't really user-friendly. In Chapter 6, *Commands and Mappings*, on page 53, we'll see how we can bind that function call to our own key mapping.

As we continue with our plugin project, you'll find the autoload directory is a useful place to put most of the code. The next area of VimL you'll learn about is filetypes, including the use of the ftdetect and ftplugin directories, which allow you to add support for filetypes Vim doesn't support by default. In the next chapter we'll see how we can have our plugin recognize and add behavior based on filetypes.

CHAPTER 4

Recognizing File Types

In Chapter 3, *The Autoload System*, on page 23, we named our mpc playlist's buffer by giving it the filename mpc.mpdv. With a filename, we now can work with Vim's filetype support.

Standard Vim supports a *lot* of different filetypes. We turn on filetype recognition using the :filetype command in our vimrc; usually we set the command like this, which also tells Vim to indent files if it can do that for the filetype:

```
filetype plugin indent on
```

With this set, Vim can recognize a file extension or line of code in a file and then set indentation levels, define custom commands, and enable the correct syntax highlighting.

Filetype recognition is buffer-local; that is, Vim sets the filetype option separately for each buffer. If we open a C file, Vim will recognize the .c and set that buffer's filetype to c. If we then open a Java file in another window, that's a different buffer, and Vim will set its filetype to java. Vim recognizes many filetypes by their extensions, but as we'll see in the mpc plugin, that's not the only way it can detect a filetype.

In this chapter you'll discover how you can execute Vim commands automatically, learn about the ftplugin directory, and even go over how to write your own statusline. First up? Vim autocommands!

Autocommands and Their Events

An *autocommand* is a command, or a series of commands, that Vim automatically executes when a particular condition occurs. The conditions are called *events*, and Vim 7.4 includes more than 80 events, which we can trigger by doing anything from creating a new buffer to changing the color scheme to losing the user's interest (see :help UserGettingBored). We can use autocommands

to have Vim automatically execute code when a user loads a certain kind of file, set a setting to a specific value, or trigger some other kind of event.

Let's look at an example. This is a very basic autocommand:

```
autocmd VimLeave * echo 'Bye!' | sleep 1000m
```

The keyword is autocmd. Following the keyword is the event name, VimLeave, and then the *file pattern* for Vim to watch for. This autocommand tells Vim that for any file (*), it should wait for the VimLeave event, which is triggered right before Vim quits. When the event is triggered, Vim will :echo the message Bye! to the user, :sleep for one second, and then quit.

The file pattern is a *glob* expression: it generally contains special *wildcard* characters, which Vim expands before using the expression. (See :help file-patterns and :help autocmd-patterns.) Vim checks the current file's filename against the file pattern to decide whether to execute the autocommand. (If the pattern includes a directory slash /, Vim looks at the whole path to the file; otherwise, it checks just the filename.) After the file pattern, we include the commands that we want to run when the event and the file pattern coincide. The commands are on a single line; if we want to run multiple commands, as we do above, we can use the | character. To include multiple lines, we can use the \ line-continuation operator.

Here's a more useful example, an autocommand from the vimrc_example.vim file that comes with Vim. You can get to it by running :e $VIMRUNTIME/vimrc_example.vim.

```
autocmd FileType text setlocal textwidth=78
```

This one checks for a particular *option*. The FileType event is triggered when filetype is set to the specified value—in this case, text, which it will be set to when we open a text file. If the event occurs, we run :setlocal textwidth=78, which sets another option, textwidth, to 78 in the file's buffer. This autocommand uses :setlocal, which is like :set but changes an option only in the current window or buffer. :setlocal is helpful for options like filetype or textwidth because those options' values apply to only individual buffers, and we don't want to mess something up by overriding them across every open buffer or window.

Another autocommand from vimrc_example.vim automatically moves our cursor to its last known position when we reopen a file:

```
autocmd BufReadPost *
  \ if line("'\"") > 1 && line("'\"") <= line("$") |
  \   exe "normal! g`\"" |
  \ endif
```

Note that this one uses the | (bar) and \ (backslash) characters. With just the bar, we could run all of this in Vim's command line as a single line, but for purposes of readability and sanity it's best if we break out longer autocommands into regular lines.

The if statement in this example checks the " mark, which stores the last cursor position in a file. If that mark is past the first line but at or before the last line, we use :execute to run the normal-mode command sequence g`". In the autocommand, the " is escaped with a \ character. This takes us to the line and column of the " mark.

There are all kinds of autocommand events. Many of them are related to opening and closing buffers and windows, but there are others we can trigger by moving the cursor, pausing for a while, writing or reading files, and so on. There are also some, like FileType, that track the settings of different options. For the complete list with details on each event, see :help autocommand-events.

Autocommands, unlike other VimL constructs that we'll get into later, don't have a dedicated file or directory of their own. Instead, when we include them in a file, we can organize them into *autocommand groups*.

We define a new group using the keyword augroup followed by a group name:

```
augroup nameOfOurGroup
```

In actuality, every autocommand we write is included in an autocommand group. By default, new autocommands are placed in the default group. In fact, when we define a new group, we're essentially laying out a *break* from the default group: all of the autocommands declared before our group begins are part of the default group, the ones declared after it begins are part of our group, and the ones declared after our group ends are back to being in the default group. If we're declaring several related autocommands, it's a good idea to collect them in our own group.

Like the autoload system, autocommand groups serve a dual purpose. By separating autocommands into groups, we can execute a group's autocommands specifically. This means that, similar to how we declare functions, we can override previously defined autocommands before we declare new ones. A standard practice in defining autocommand groups is to start by deleting or clearing all previous commands that might be part of the group, like so:

```
augroup nameOfOurGroup
autocmd!
```

With that bang appended, autocmd! clears the nameOfOurGroup group, which would prevent collisions if we were to reload the source file. It would also ensure that our nameOfOurGroup autocommands are always the latest.

We're going to go back to that vimrc_example.vim file, because it contains a great example of how we use groups. Here are the autocommands we've just looked at, but in their context:

filetype/vimrc_example.vim

```
augroup vimrcEx
au!

autocmd FileType text setlocal textwidth=78

autocmd BufReadPost *
  \ if line("'\"") > 1 && line("'\"") <= line("$") |
  \   exe "normal! g`\"" |
  \ endif

augroup END
```

This is most of the autocommand block from the example .vimrc file. It starts a group called vimrcEx and then uses a shortened form of autocmd! to clear that group. (Remember from *Is It echomsg or echom?*, on page 18, that most VimL keywords can be shortened to various abbreviated forms.)

Notice how the group ends. The closing line, augroup END, actually does what the opening line does: it denotes the name of an autocommand group. The name END refers to the default group, so that augroup END effectively closes the group we've defined, and then, as I described earlier, any autocommands declared after this line will be back in the default group.

So, why have we been talking about autocommands?

Detecting the Current File Type

When we went over the directories that can be used in a Vim plugin, back in *The Structure of a Vim Plugin*, on page 12, I mentioned the specialized ftdetect directory. In the average plugin, we probably wouldn't need to use this directory. Its entire purpose is to hold script files that detect the type of the file we're editing—and it turns out that one popular way to have the script files do that is to have them use autocommands.

Go ahead and create the ftdetect directory under the main mpc directory if you haven't already, and then add a file called mpdv.vim under that. It should contain the following line:

Filetype vs. Global Plugins

There are two categories that Vim plugins tend to fall into. They are *global plugins* and *filetype plugins*.

A global plugin is general-purpose. It can apply regardless of the file we're editing, and it might fill a common need, such as searching a directory using ack or toggling relative and specific line numbers.[a,b] Our mpc plugin fits into this category.

A filetype plugin is aimed at a particular filetype. It uses the ftdetect directory to determine when to activate its functionality and the ftplugin directory to store its functions. A filetype plugin's functionality typically applies to only the current buffer, where we're editing a relevant file.

Most Vim plugins would use either one of these sets of directories—ftdetect and ftplugin for a filetype plugin, or plugin, autoload, and others if it's a global plugin. For learning purposes, though, we'll be working with both sets in our plugin.

a. https://github.com/mileszs/ack.vim
b. https://github.com/jeffkreeftmeijer/vim-numbertoggle

filetype/mpc/ftdetect/mpdv.vim

```
autocmd BufRead,BufNewFile *.mpdv    set filetype=mpdv
```

There are a number of ways to detect a filetype in Vim. The one we're using here looks at the file extension—we're using an autocommand that watches for the BufRead and BufNewFile events, which occur when we open a new buffer or file, and we're giving it a file pattern that matches any file with the extension .mpdv. When we load a file that matches the pattern, Vim will run the command set filetype=mpdv before going on.

Another way to detect the filetype would be to actually look at the contents of the file. Instead of using an autocommand that applies to the filename, we could check the first lines of the file when it's loaded and compare them to a regular expression to see whether the file is of our filetype. Vim has HTML filetype recognition built in, but if it didn't and we wanted to write our own filetype plugin for HTML, we could do something like this:

```
if getline(1) =~ '\<html\>'
  set filetype=html
endif
```

This uses getline() to get us the contents of the first line as a String. If the line matches our regular expression, we set the filetype option accordingly.

Making Filetype-Specific Changes

We now have our filetype ready to go. Run the OpenMPC() function again:

```
:call OpenMPC()
```

And now for a handy trick: when we call the :set command on an option and include the ? flag instead of a value, :set echoes the option's current value. Try this now with filetype:

```
:set filetype?
```

You should see Vim report our filetype:

```
filetype=mpdv
```

With a filetype, we now can set buffer-specific options, using a file in that other filetype-related directory: ftplugin. Create that directory if it doesn't exist yet.

Within ftplugin, create the file mpdv.vim. We won't be doing too much in this file just yet—for now, just give it this line:

filetype/mpc/ftplugin/mpdv.vim

```
set buftype=nofile
```

The buftype option can have one of several values. The one we give it here, nofile, tells Vim that the buffer is not related to a file, so it won't be saved or written anywhere.

Let's try updating a more noticeable setting. When a user opens the mpc window, we'll have the ftplugin file give it a special custom statusline.

What Makes Up a Statusline

The statusline is actually one of Vim's options, which means that we can set it with :set, just like the other options:

```
set statusline=Hello!
```

But typically the statusline string contains special values. Each part of the string is called an *item*, and there are a number of special built-in items that stand for the number of lines in a file, the path to the file, the buffer number, whether the file is read-only, the current column number, and other useful bits of information.

There's also a special syntax for the string. To include an item, we precede it with a % character—for example, this is how we would include F, which represents the full path to the current file:

```
set statusline=%F
```

The default statusline uses a whole bunch of these items. Here's the string that makes up the default statusline when the ruler option is set:

```
%<%f\ %h%m%r%=%-14.(%l,%c%V%)\ %P
```

And from that chaotic assemblage of percent signs we get a statusline such as this:

```
wowc.txt-----------------------------------------93147,1---------87%-
```

This statusline tells us that we're on line 93,147 of wowc.txt, on column 1, and that the last line we can see in our window is 87 percent of the way through the file. (Yeah, wowc.txt is a big file.)

Let's deconstruct the string and see what each of those items does.

`%`	The beginning of the string.
`<`	The point from which to truncate the item if it ends up being too long.
`%f`	The path to the file we're editing.
`\`	A literal space (escaped with a backslash).
`%h`	A [Help] flag, which shows up if we're looking at a Vim help file.
`%m`	A *modified* flag. If we have unsaved changes to the current file, it will display [+].
`%r`	A RO flag, shown only if the current file is read-only.
`%=`	The division between the left-justified items in the statusline and the rest, which will be right-justified.
`%-14.`	Settings for the next group, which will have a minimum width of 14 characters.
`(`	The beginning of a new group of items. These groups are typically used so that width and justifying rules can be applied to multiple items.
`%l`	The current line number.
`,`	A literal comma.
`%c`	The number of the current column.
`%V`	The current *virtual column number*. (See :help virtcol().)
`%)`	The end of the item group.
`\`	Another escaped space.
`%P`	The last percentage of the file currently visible in the window.

One item that we don't see in this default statusline string is the function. Yes, statusline strings can include functions, as in the following example:

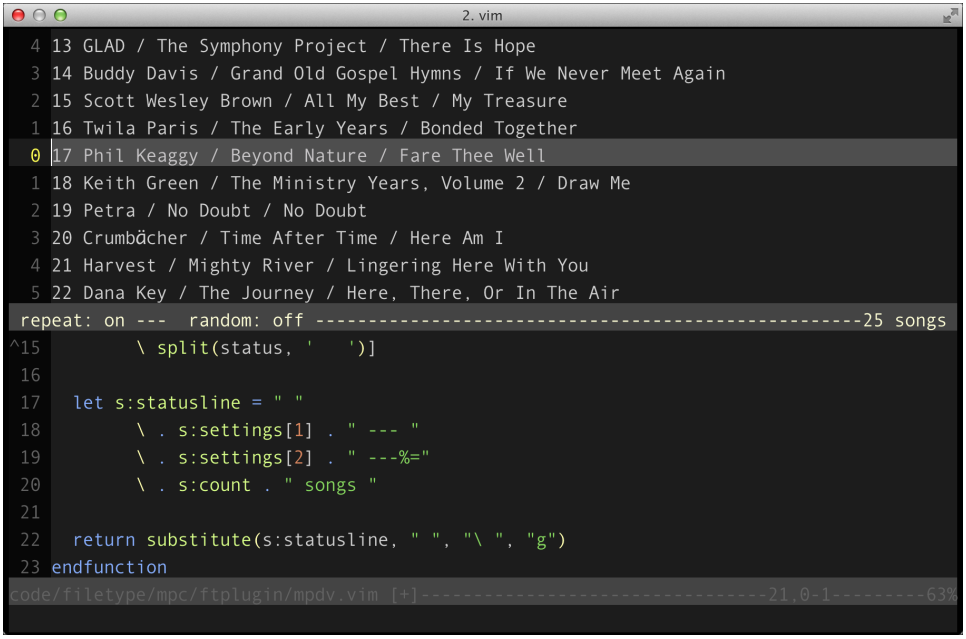
```
%!OurOwnPersonalStatusLine()
```

Vim would call the OurOwnPersonalStatusLine() function and use its return value for the statusline. For the metaprogrammers in the audience: yes, Vim can evaluate expressions in statusline items. (See :help statusline for detailed descriptions of %{} and other item types.)

This is what we'll do for our mpc plugin's statusline. In the mpdv.vim file, we'll add a function called GetMPCStatusLine(). It will call mpc status and break the output from that into bits, from which we will assemble an informative statusline.

Constructing a Statusline

Here is what we're after:



On the left, this statusline shows the current Repeat and Random settings; on the right, it shows the playlist's track count. Delightfully simple!

At the beginning of the mpdv.vim file, add the code for the GetMPCStatusLine() function:

filetype/mpc/ftplugin/mpdv.vim
```
function! GetMPCStatusline()
  let cmd = "mpc status"
  let result = split(system(cmd), '\n')

  let status = len(result) == 3 ? result[2] : result[0]

  let [s:count, s:settings] =
        \ [len(split(system('mpc playlist'), '\n')),
        \ split(status, '   ')]

  let s:statusline = " "
        \ . s:settings[1] . " --- "
        \ . s:settings[2] . " ---%="
        \ . s:count . " songs "

  return s:statusline
endfunction
```

We'll again go through this one piece at a time.

```
function! GetMPCStatusline()
  let cmd = "mpc status"
  let result = split(system(cmd), '\n')
```

We start by getting the command output from mpc status, splitting its lines into List items. Since the number of lines will vary depending on whether a track is currently playing, we next check the length of the List, using len().

The status variable is going to contain a particular line from the mpc status output. To pick that line based on the length of the result List, we use a ternary operator:

```
let status = len(result) == 3 ? result[2] : result[0]
```

If the length is 3, there's a track playing, and mpc is reporting that in addition to the usual status information. In that case, we take the last item in the List: result[2]. Otherwise, we take the first (and only) item: result[0].

```
let [s:count, s:settings] =
      \ [len(split(system('mpc playlist'), '\n')),
      \ split(status, '   ')]
```

This is an interesting use of let: we're using it to assign multiple variables at once by giving it a List. The only catch when we do this is that we have to include as many variable names as there are List items.

The variables s:count and s:settings are using another variable scope: the *script scope*, which is denoted by the s: prefix. s:count contains the playlist track count, which we get from splitting the result of mpc playlist by newlines into a List and then checking the List length. The other variable, s:settings, contains

the output from that line of mpc status. We split it on every three spaces because that is how mpc separates the items of information we want.

```
let s:statusline = " "
      \ . s:settings[1] . " --- "
      \ . s:settings[2] . " ---%="
      \ . s:count . " songs "
```

Now we're getting to the actual statusline string. The s:settings variable contains the repeat and random settings; we start the statusline with those and then add the %= left-hand vs. right-hand separator. At the end, we put the track count, s:count.

```
  return s:statusline
endfunction
```

And our function ends by returning the s:statusline.

Now to use it! At the end of mpdv.vim, right under the buftype setting, set the statusline:

```
set buftype=nofile
setlocal statusline=%!GetMPCStatusline()
```

And then go ahead and try it. Run :call OpenMPC(), and behold, we have our own statusline!

We'll come back to our ftplugin file later on, in Chapter 6, *Commands and Mappings*, on page 53. Our next pursuit is related: now that we have a filetype, we have a chance to see how Vim handles syntax highlighting. Just as Vim lets us add support for our own filetypes, it lets us write our own syntax highlighting; in the next chapter we'll get into the syntax directory and begin doing just that.

CHAPTER 5

Highlighting Syntax

Syntax highlighting is part of Vim's filetype support for the wide variety of languages that it supports by default. It relies on *syntax files*, VimL script files that define the elements of languages and place them in standard categories so that Vim knows how to format and highlight code in those languages. When we come up with our own filetype, it's up to us to tell Vim how to highlight that filetype's syntax.

Our plugin project is getting close to where we'll put on the finishing touches. In this chapter we'll clean up the playlist track listing and create our own special syntax. Then we'll prettify it with syntax highlighting. We'll use the filetype we worked on in the previous chapter and see another facet of how a Vim filetype plugin works.

The Vim Syntax File

We know from *The Structure of a Vim Plugin*, on page 12, that syntax files are kept in the `syntax` directory. Create a `syntax` directory under the main plugin directory if you haven't done that yet. Then within it, create the file `mpdv.vim`.

Distinguishing Syntax Elements

We specify syntax elements by using the `:syntax` command. The command can define several different element types, but the ones we're most commonly going to use are `keyword`, `match`, and `region`.

`keyword`	Used to specify a keyword element or a list of keyword elements.
`match`	Used to specify a class of elements defined by a provided regular-expression pattern.
`region`	Used to specify an element defined by starting and ending regular-expression patterns.

Here are examples of all three:

```
syntax keyword langType     String Number Dictionary List
syntax match   langComment  /".*/
syntax region  langString   start=/'/ skip=/''/ end=/'/
```

The first arguments in these commands—langType, langComment, and langString—are the names of *syntax groups*. By convention, we start group names with the filetype for the language that the syntax file is for, so here lang would be a filetype. (This is also what :set filetype? would return.) The langType group would describe types in the language of the lang filetype.

After the group names come the elements. A keyword is a simple string, such as if or for. An element defined in a match uses a regular-expression pattern, which we delimit with / characters, and an element defined by a region includes everything between the characters that we specify as the start and the end.

In our region example, we're using another pattern, the optional skip, to define false-alarm patterns on which we *don't* want to end a match. The langString group defines a String as two single quote marks and everything between them. The skip pattern matches a *pair* of single quotes, so if we come across two consecutive single quotes, the string will go on until it finds another lone single quote, which will be the end. (If you recall from when we discussed the String type in *Functions, Types, and Variables*, on page 1, this is how single quotes are escaped in single-quoted VimL strings.)

Remember that most of VimL's vocabulary consists of Ex commands, like we run in the Vim command line. :syntax is no exception, which means that we can try out these examples by opening an empty Vim buffer, entering a line of our syntax, and then executing a :syntax command, like this:

```
This is a String: 'Hello!'

:syntax region langString start=/'/ skip=/''/ end=/'/
```

So enter that first line in a new empty buffer, execute the second line in Vim's command line, and…nothing happens. Why is that?

Linking Syntax Groups to Highlight Groups

What we just did in our langString example was define a syntax group—we told Vim how to distinguish a langString element from the surrounding code. What we did *not* do was tell Vim how to *highlight* the langString element. To do that, we must use the other key syntax-related command, :highlight.

With :highlight, we set the color and other formatting options that Vim uses to highlight syntax elements. The command takes an element and a set of

arguments specific to different terminal and GUI Vim configuration, since the various terminal and GUI versions of Vim have varying levels of support for the formatting options.

term — Used to specify the format used in *normal* terminals, especially those lacking color capabilities. Example: term=bold.

cterm — Used to specify the format or colors used in color-capable terminals; also relevant are ctermfg, or the color to use for text in a color terminal, as well as ctermbg, the background color to use in a color terminal. Example: cterm=bold ctermfg=blue ctermbg=white.

gui — Used to specify the format or colors used in GUI versions of Vim; also relevant are guifg, or the color to use for text in a GUI Vim window, and guibg, or the color to use for the background in a GUI Vim. Example: gui=underline guifg=darkBlue guibg=green.

The values we give to term, cterm, and gui are part of a set that includes, among others, italic, underline, and bold. The color value for ctermfg, ctermbg, guifg, and guibg can be a Vim color name, such as Blue or Green or a color number or RGB hexadecimal value. (There are complete lists of the color numbers that we can use; see :help cterm-colors and :help gui-colors.)

Another way to use :highlight is to have it define groups. These aren't *syntax* groups like we define with :syntax, but *highlight groups*. Highlight groups are classes of syntax to which we can apply colors and formatting options, using color schemes. Vim uses several highlight groups for things like the statusline, the last search match, and the divider between split windows. Similarly, there are commonly used groups that are used by convention for language constructs, such as comments, types, and operators. (See :help group-name.)

There are a couple of ways in which we can use :highlight to highlight syntax groups. One way is to set the color values for the syntax group directly:

```
highlight langString ctermfg=Blue guifg=#0000FF
```

Try running this in the command line on that new empty buffer—you should see the string that we entered turn blue. Yay!

The only problem with doing this for a language's syntax file is that it breaks the user's color scheme. Color scheme files are written to be portable: they set colors and other options for terminal and GUI Vim versions. Let's say that our user has searched Vim's website for color schemes and has installed a color scheme that contains this line:[1]

1. http://www.vim.org/scripts/script_search_results.php?script_type=color+scheme

```
highlight String  ctermfg=113 cterm=none guifg=#95e454 gui=italic
```

Most Vim color schemes will contain an equivalent to this line; it specifies colors for the String highlight group. In the case of a GUI Vim, it specifies italic type. Our :syntax command for langString, however, formats the langString group directly. Since langString is part of the lang syntax file, other color schemes won't format it—they format instead the general-purpose highlight group String. So to take advantage of other color scheme files, we have to *link* our syntax groups to the conventional highlight groups for which the color schemes are written.

This linking approach, then, is the other way to use :highlight to highlight syntax groups. Here's an example of how to do it, using a slightly modified line from the Groovy syntax file that ships with Vim:

```
highlight link groovyComment Comment
```

groovyComment is a syntax group that's defined in the groovy.vim syntax file. This line links it with Vim's Comment highlight group, so that now any color scheme can provide appropriate highlighting for comments in a Groovy file.

Formatting the Playlist

Our first step in making formatting improvements to our playlist will be to neatly align the items of each track. Currently, we simply display each track's items, separated by a / character, and we aren't paying any attention to each item's length.

Our pre-first step will be to move the playlist-fetching code to a separate function. In this new function we'll call mpc to get the playlist, divide up each track's items, format them all to display nicely, and then return the result to mpc#DisplayPlaylist().

We'll put this at the top of our autoload/mpc.vim file. Here's how it should start:

vsyntax/mpc/autoload/mpc.vim

```
function! mpc#GetPlaylist()
  let command = "mpc --format '%position% @%artist% @%album% @%title%' playlist"
  let [results, playlist] = [split(system(command), '\n'), []]
  let maxLengths = {'position': [], 'artist': [], 'album': []}
```

We begin by calling mpc, as we'd expect, but this time we're using a different format for the output:

```
mpc --format '%position% @%artist% @%album% @%title%' playlist
```

Each item making up a track in the playlist is separated by the string @. We'll need this later on; track titles and album names can contain spaces, so we can't use the space as a delimiter.

After we define the command variable, we define three others. results is the playlist from mpc, split into a List by newline characters. playlist, for now, is an empty List, and maxLengths is a Dictionary, with List entries for position, artist, and album. Let's see how this is used.

```
for item in results
  let song = split(item, " @")
  let [position, artist, album, title] = song

  call add(maxLengths['position'], len(position))
  call add(maxLengths['artist'], len(artist))
  call add(maxLengths['album'], len(album))
endfor
```

Here we use a for loop on results. We create a List called song to hold the split-up track items, and then we assign those items to the variables position, artist, album, and title. Then we add the length of each of these to the corresponding List in maxLengths.

```
call sort(maxLengths.position, "LargestNumber")
call sort(maxLengths.artist, "LargestNumber")
call sort(maxLengths.album, "LargestNumber")
```

Next, we call sort() on each List in maxLengths.

Brief digression: notice that we aren't giving sort() just the List to sort—we're also including "LargestNumber", which is the name of a custom function that will do the sorting. Normally, we would use sort() like so:

```
let scientists = ['Robert Whate', 'Bill Cook', 'Alfred Clark',
                \ 'Fred Stoner', 'Brad Noggin', 'Squirt']
echo sort(scientists)
" →  ['Alfred Clark', 'Bill Cook', 'Brad Noggin',
"     'Fred Stoner', 'Robert Whate', 'Squirt']
```

And for a List like the one in the example, this works perfectly, because sort() sorts on the *String* representations of the items in a List. But because it does that, we can't use this on a List comprising Number items:

```
let numbers = [4, 5, 15, 78, 9]
echo sort(numbers)              " → [15, 4, 5, 78, 9]
```

The LargestNumber() function, as you'll see when we add it, won't sort alphabetically, so it will avoid this problem. (Strange as it may seem, this custom function is actually the officially recommended solution for sorting numbers;

see :help sort().) But this concludes our digression. For now, add the next part of mpc#GetPlaylist():

vsyntax/mpc/autoload/mpc.vim
```
for item in results
  let song = split(item, " @")
  let [position, artist, album, title] = song

  if(maxLengths.position[-1] + 1 > len(position))
    let position = repeat(' ',
          \ maxLengths.position[-1] - len(position))
          \ . position
  endif
  let position .= ' '
  let artist .= repeat(' ', maxLengths['artist'][-1] + 2 - len(artist))
  let album .= repeat(' ', maxLengths['album'][-1] + 2 - len(album))

  call add(playlist,
        \ {'position': position, 'artist': artist,
        \  'album': album, 'title': title})
endfor
```

After sorting the maxLengths, we again loop through the results, this time using spaces to pad each of the values that makes up a track. position is right-aligned—we add padding to its beginning rather than to its ending—and the others are left-aligned.

To add the correct number of spaces, we use the function repeat(). It takes two arguments: a value to repeat, which in our case is a space character, and a number of times to repeat that value, which we calculate. We either prepend or append the spaces to the track values, and to get the number of spaces, we use the longest corresponding item from maxLengths minus the length of the current item.

At the loop's end we add a new Dictionary, containing the padded track items, to the playlist we defined at the start of the function.

```
  return playlist
endfunction
```

And last of all, we return the playlist. That's a fairly straightforward process on which we won't spend much time.

Oh, right! Before we can use this, we need to add LargestNumber():

```
function! LargestNumber(no1, no2)
  return a:no1 == a:no2 ? 0 : a:no1 > a:no2 ? 1 : -1
endfunction
```

Simple enough. We take two numbers and return 0 if they're equal, 1 if the first number is larger, and -1 if the second number is greater.

Now we need to modify mpc#DisplayPlaylist() to make use of our new function. In mpc#DisplayPlaylist(), replace everything up to the opening if statement with the following highlighted lines:

```
function! mpc#DisplayPlaylist()
  let playlist = mpc#GetPlaylist()

  for track in playlist
    let output = track.position . " "
          \ . track.artist
          \ . track.album
          \ . track.title
    if(playlist[0].position == track.position)
      execute "normal! 1GdGI" . output
    else
      call append(line('$'), output)
    endif
  endfor
endfunction
```

Also, as you see above, make sure to replace track with output the two times that it occurs after the highlighted lines.

Our playlist's tracks are now formatted nicely. Wait to check that, though—it's now time to add highlighting.

Using conceal with Syntax Regions

For the playlist highlighting, we're going to use region syntax groups. We'll use special characters to delimit each item in a track, but we won't *show* those characters—they're just to help us with highlighting. The effect will be to use different colors for each item that makes up a track. To do this, we'll make use of a special feature of Vim's syntax highlighting: *conceal*.

conceal is actually an argument that we can give to the :syntax command; it tells Vim that it can hide (or conceal) an element when it comes across it. The related argument concealends does the same thing, but for the start and end characters of a region: when we use it, the *ends* become concealable, and the text between the ends doesn't. We're going to be using concealends.

The conceal functionality depends on two Vim options: conceallevel, which takes a number between 0 and 3, and concealcursor, which takes a string containing any of the letters n, v, i, and c. Each letter stands for a Vim mode. The numbers 0 through 3 tell Vim what to do with concealable syntax elements—for example,

if conceallevel is set to 0, Vim shows these elements, or if it's set to 3, it hides them entirely. Vim treats the current cursor line specially; if the current mode is included in concealcursor, then the line that the cursor is on is treated as conceallevel says, but otherwise it's shown. This makes it easier for us to edit concealable syntax items—we can set them to be shown when we move the cursor over them.

We can combine the concealends argument with one or both of two others, contains and matchgroup, to set separate highlighting for an element and its *ends*. contains refers to the text without the ends. In the case of a string delimited by quotes, that would be the string itself. matchgroup is a group name containing the ends, which would be the quotes in that string.

Say we wanted text to be shown in bold when we surrounded it by asterisks. We could use matchgroup in something like this:

```
syntax region mdBold matchgroup=boldEnds start=/*/ end=/*/ concealends
```

And then we could highlight the mdBold group like this:

```
highlight mdBold cterm=bold gui=bold
```

And then, if we set the conceallevel option correctly, we could write this:

```
This is *bold* text.
```

Vim would hide the asterisks and display the word *bold* in a bold font.

Specifying a New Syntax

Let's get back to mpc#GetPlaylist() now. We have an odd problem when coming up with a syntax for our playlist's text. We need to separate titles and names, but we also need to be able to tell which is which. To do that, we'll need to use more specific delimiting characters—titles and names can contain the space character, so we can't use that.

Here's our solution: For each track in the playlist, we'll append a couple of characters of the item type's name, along with our trusty @ character, to the beginning and ending of each item. If mpc gives us Jim Cole as the artist our playlist will include that as follows:

```
@arJim Colear@
```

We'll add two new functions to do this; one will *encode* track items in this syntax, and one will *decode* the syntax. Here they are; add them, and then we'll go over them:

vsyntax/mpc/autoload/mpc.vim
```
function! mpc#EncodeSong(item)
  let item = split(a:item, " @")
  let song = {'position': item[0],
        \ 'artist': '@ar' . item[1] . 'ar@',
        \ 'album': '@al' . item[2] . 'al@',
        \ 'title': '@ti' . item[3] . 'ti@'}
  return song
endfunction

function! mpc#DecodeSong(item)
  let line_items = split(substitute(a:item, ' \{2,}', ' ', 'g'), ' @')
  let song =  {'position': line_items[0],
        \ 'artist': line_items[1][2:-4],
        \ 'album': line_items[2][2:-4],
        \ 'title': line_items[3][2:-4]}
  return song
endfunction
```

First, mpc#EncodeSong() splits an item by the @ separator that we use when we get the playlist from mpc. It returns a Dictionary, with entries for each of the items that make up a track.

Then in mpc#DecodeSong(), we take a different approach. In this function we're dealing with items that mpc#GetPlaylist() has formatted as tracks, and they'll all have different amounts of padding between them. So we use substitute() to replace all occurrences of two or more spaces with a single space, and then we split() the result on the @ between each item. We return the result as a Dictionary.

Now we need to use these functions in mpc#GetPlaylist() before we return the playlist text. In that function, change the highlighted lines:

vsyntax.1/mpc/autoload/mpc.vim
```
function! mpc#GetPlaylist()
  let command = "mpc --format '%position% @%artist% @%album% @%title%' playlist"
  let [results, playlist] = [split(system(command), '\n'), []]
  let maxLengths = {'position': [], 'artist': [], 'album': []}

  for item in results
    call add(playlist, mpc#EncodeSong(item))
  endfor

  for track in playlist
    call add(maxLengths['position'], len(track.position))
    call add(maxLengths['artist'], len(track.artist))
    call add(maxLengths['album'], len(track.album))
  endfor
```

```
  call sort(maxLengths.position, "LargestNumber")
  call sort(maxLengths.artist, "LargestNumber")
  call sort(maxLengths.album, "LargestNumber")

  for track in playlist
    if(maxLengths.position[-1] + 1 > len(track.position))
      let track.position = repeat(' ',
            \ maxLengths.position[-1] - len(track.position))
            \ . track.position
    endif
    let track.position .= ' '
    let track.artist .= repeat(' ',
      \ maxLengths['artist'][-1] + 2 - len(track.artist))
    let track.album .= repeat(' ',
      \ maxLengths['album'][-1] + 2 - len(track.album))
  endfor

  return playlist
endfunction
```

This looks more complicated than it is. Let's go through it.

At the beginning, we used to create a List for each track, using split() on each item of the results from mpc. Now we instead call mpc#EncodeSong() on those items. This is how we populate the playlist.

Next, we add the items' lengths, via len(), from each track in the playlist to the right List in maxLengths. We sort each List using LargestNumber().

In our last loop, we pad each of the items in each track. This uses maxLengths and the repeat() function that we saw before.

To wrap this up, we need to write the :syntax and :highlight commands that will tell Vim what these items are and how to highlight them. Open the syntax file we created earlier, syntax/mpdv.vim. Add the following commands:

vsyntax.1/mpc/syntax/mpdv.vim

```
syntax region mpdArtist matchgroup=mpdArtistSyn start=/@ar/ end=/ar@/ concealends
syntax region mpdAlbum matchgroup=mpdAlbumSyn start=/@al/ end=/al@/ concealends
syntax region mpdTitle matchgroup=mpdTitleSyn start=/@ti/ end=/ti@/ concealends

highlight default mpdArtist ctermbg=234 ctermfg=lightgreen
                          \ guibg=#1c1c1c guifg=#5fff87
highlight default mpdAlbum ctermbg=234 ctermfg=lightblue
                          \ guibg=#1c1c1c guifg=#5fd7ff
highlight default mpdTitle ctermbg=234 ctermfg=lightmagenta
                          \ guibg=#1c1c1c guifg=#ffafff
```

We're defining three groups: one each for artist, album, and title. In each group, we set the matchgroup groups and start and end regular-expression patterns, we

set concealends. The key bit in each of these commands is of course the name of the syntax group, which comes right after the region keyword: the artist gets the group mpdArtist, album becomes mpdAlbum, and title is mpdTitle.

Now turn your attention to the final three lines: the :highlight commands. These syntax groups don't really fall into any category of programming-language constructs, so rather than linking them to the conventional highlight groups, we highlight the syntax groups directly. But notice the argument with which we're starting each :highlight command. When we include the default argument in a :highlight command, that command becomes the *default* way to highlight the group; in other words, it can be overwritten. If a user liked our colors overall but wanted the artist column to be displayed in red, he could add this line to his .vimrc:

```
highlight mpdArtist ctermfg=red guifg=#FF0000
```

And Vim would ignore our choice of color in favor of this.

The last step before we can see our highlighting in action is for us to go to the filetype file and make sure that Vim handles this syntax correctly. Open ftplugin/mpdv.vim and add these lines below the buftype and statusline settings:

vsyntax.1/mpc/ftplugin/mpdv.vim
```
setlocal conceallevel=3
setlocal concealcursor=nvic
```

This sets conceallevel to hide our regions' start and end patterns. We also set concealcursor so that it won't show the patterns in any of the four major modes: normal, visual, insert, and command. Remember that those patterns are in the groups that we put as the matchgroup of the regions. Because we set concealends on those regions, the pattern groups will now be hidden, and all we'll see will be the groups that each of those regions contains.

At long last, it's time to try this out. Open Vim and run :call OpenMPC() again. You should see the beautifully highlighted playlist, as shown in Figure 1, *The highlighted playlist*, on page 52.

We now have our playlist interface close to finished. You also know a bit more about syntax and color scheme files than you did at the start of the chapter—but you've gone long enough calling all of your plugin's functions by hand via the Vim command line. In the next chapter you'll learn about writing Vim commands and how you can add your own mappings that call those commands.

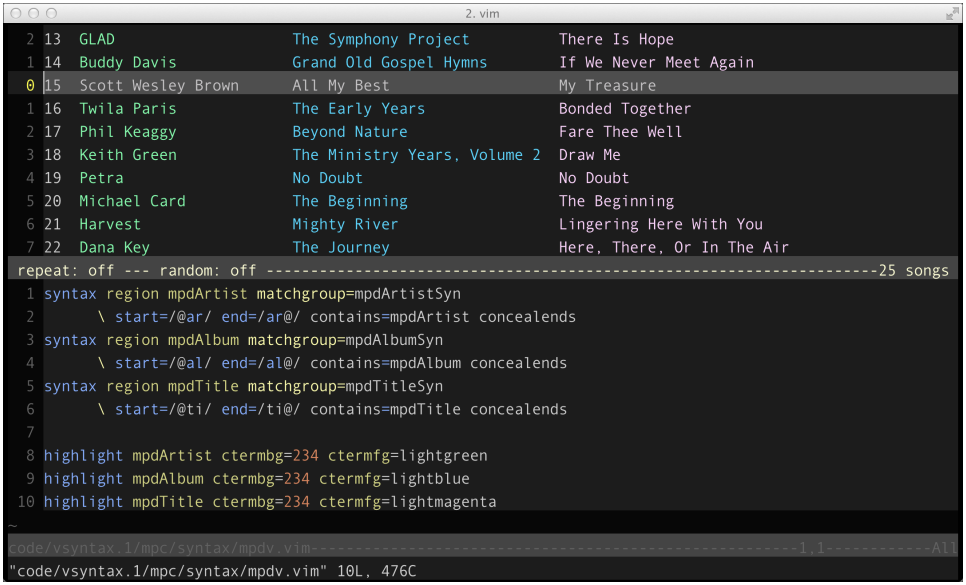


Figure 1—The highlighted playlist

CHAPTER 6

Commands and Mappings

We've been using Vim commands throughout this book. In fact, most of the code we've been writing in our script files could be executed on Vim's command line as a long series of commands. But as with functions, we can write our own commands with VimL—they're called *user commands*.

We can also map keys and key combinations. You might have mapped or remapped Vim functions before in your .vimrc file. Vim gives us a lot of choices for how we map keys—particularly when we include the mappings as part of our plugin—and even lets us give users an easy way to reconfigure those mappings to taste.

In this chapter we'll start by writing commands that call our mpc functions. Then we'll see how we can map keys to call our commands; we'll take advantage of the different mapping commands and their options, and we'll get a glimpse of just how versatile the mapping system is.

Writing User Commands

At this point you probably won't be shocked to learn how we define our own commands: by using a command! Specifically, we use :command to define commands. (Hmm—that certainly sounds redundant.)

User commands, like user functions, have to start with uppercase letters. And what's funny about user commands is that their job is simply to call *built-in* commands. If we want to define a command that calls a function—as we will be doing—we execute the :call command. Consider an example:

mappings/command.vim

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

This defines a user command called :Quote. Just like with :command itself, we can leave the colon (:) off the command that we're calling.

At Vim's command line, we'd run :Quote like we do any other Ex command:

```
:Quote

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

What if that function had an argument? We can try this on an alternate version of EchoQuote() that does:

```
function! EchoQuote(quote)
  echo a:quote
endfunction
```

To tell our command that it should expect arguments, we give it the nargs flag…with an *argument*. The argument specifies what number of arguments the command takes. By default it takes 0, and we can say so outright:

```
command -nargs=0 Quote   call EchoQuote()
```

But the new EchoQuote() has an argument, and obviously there are other ways to set nargs. If we set it to 1, the command will take 1 argument; if we set it to ?, it will take an optional argument, one or none. If we set it to *, it will take any number of arguments. We can also set it to + and require at least one argument.

Here's a :Quote command that takes one argument: the quote that EchoQuote() expects.

```
command -nargs=1 Quote   call EchoQuote(<args>)
```

In this command, we use the special code <args>. Vim replaces that with the argument we pass the command when we run it:

```
:Quote "I write, and you send me a fish."
" → I write, and you send me a fish.
```

Another take on that code is <q-args>, which *quotes* the command arguments when the command is called. If we used it on :Quote and then called :Quote "hi!" "bye!", then the value that EchoQuote() would get would be '"hi!" "bye!"'.

"I Command You To :PlaySong(3)…"

In *Retrieving the Text of a Line*, on page 27, we added the PlaySong() function, which lets our user play a song. Let's write a command for it. It will take the one argument—the line whose song to play—and run :call PlaySong() behind the scenes.

We'll put this in the ftplugin/mpdv.vim file because it's only for use with our .mpdv buffer where we show the playlist. At the end of that file, add this line:

mappings/mpc/ftplugin/mpdv.vim
```
command! -buffer PlaySelectedSong call mpc#PlaySong(line("."))
```

:command! is like function! here—it overwrites any previously declared command, like function! does with functions. Now technically this isn't a polite way to define commands. If we just wanted to be sure there was a :PlaySelectedSong command, Vim would let us use exists() to check for one—we could write it something like this:

mappings/command.vim
```
if(!exists(":PlaySelectedSong")
  command PlaySelectedSong call mpc#PlaySong(line("."))
endif
```

And then we'd leave our user's :PlaySelectedSong command intact, assuming the user had one. The thing is that in this case, the user probably doesn't—our plugin and its functions are pretty specialized. So we're going to go ahead and use :command!.

Aha—I subtly snuck a new argument in there! -buffer, when we include it in a command definition, makes the command buffer-local. With this in place, :PlaySelectedSong will be available only from within our playlist window's buffer.

Also look how the command calls mpc#PlaySong(): it uses the line() function that we first took advantage of back in Chapter 2, *A Real Live Plugin*, on page 15. We're using the dot file position to say *the current line*, and the effect is that when the command is called, mpc#PlaySong() gets called and passed the current line as an argument. This is good—it means that the command is saying, "Play the song I'm currently on."

Ooh. Maybe we should use a command to make it easier for our user to select a song in the first place. This one goes in the file under our plugin directory because we want to have it available throughout Vim. At the end of plugin/mpc.vim, add the following command, which calls OpenMPC():

mappings/mpc/plugin/mpc.vim
```
command! MpcBrowser call OpenMPC()
```

Some Toggling Functions (and Commands)

While we're adding commands, let's add some general-purpose mpc functions to the plugin.

The commands mpc toggle, mpc random, and mpc repeat actually all toggle things in mpc: playback, the setting of random, and the setting of repeat. We'll add a new function to autoload/mpc.vim for each one; it will change the setting and then echo the resulting mpc feedback to the user. Then we'll add Vim commands to call the functions.

Start with mpc#TogglePlayback():

mappings/mpc/autoload/mpc.vim
```
function! mpc#TogglePlayback()
  let command = 'mpc toggle'
  let result = split(system(command), '\n')[1]

  let message = '[mpc] '
  let message .= split(result, ' ')[0] == '[paused]' ? 'PAUSED' : 'PLAYING'
  echomsg message
endfunction
```

It's similar to the venerable mpc#GetPlaylist() in how it starts: it defines a command to call and then splits the result of calling the command. This function ends, though, by defining a message to send to the user. It begins with our usual [mpc] and then .= appends text to that—that's the . that we use for concatenating String values, but with a = on the end. Depending on what we got as output from mpc—whether the first item of result, turned into a List, is [paused]—we append either 'PAUSED' or 'PLAYING'.

You could just go ahead and try out that function, but first let's define a command for it. This one, like :MpcBrowser, will go under the plugin directory, so it will be available from anywhere and not just in our playlist window or buffer. Under the :MpcBrowser definition in plugin/mpc.vim, add this:

mappings/mpc/plugin/mpc.vim
```
command! TogglePlayback   call mpc#TogglePlayback()
```

With that in place, we can now toggle playback with the command, as shown in Figure 2, *Toggling playback with TogglePlayback*, on page 57.

The other two functions work similarly.

mappings/mpc/autoload/mpc.vim
```
function! mpc#ToggleRandom()
  let command = 'mpc random'
  let result = split(system(command), '\n')

  let status = len(result) == 3 ? result[2] : result[0]
  let message = split(status, '   ')[2] == 'random: off'
        \ ? '[mpc] RANDOM: OFF' : '[mpc] RANDOM: ON'
  echomsg message
endfunction
```

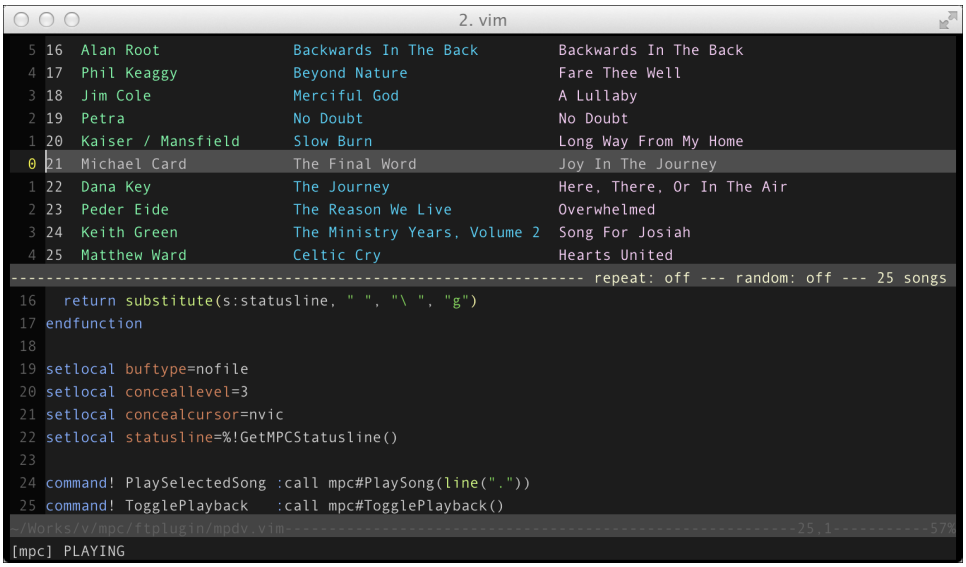


Figure 2—Toggling playback with TogglePlayback

```
function! mpc#ToggleRepeat()
  let command = 'mpc repeat'
  let result = split(system(command), '\n')

  let status = len(result) == 3 ? result[2] : result[0]
  let message = split(status, '   ')[1] == 'repeat: off'
        \ ? '[mpc] REPEAT: OFF' : '[mpc] REPEAT: ON'
  echomsg message
endfunction
```

This time, before we get a message to echo to the user, we check the length of the output—if there's a song playing, there will be three lines of output, but otherwise there'll be just one. We take whichever line has the mpc status, including the settings of random and repeat, and then we check items on *those* lines, again split() on spaces. If the status text for a setting says it is off, we send the user a message to say so. If not, we tell the user that the setting is on.

These are the :ToggleRandom and :ToggleRepeat commands. Add them right below the command that calls mpc#PlaySong():

mappings/mpc/ftplugin/mpdv.vim

```
command! -buffer ToggleRandom     call mpc#ToggleRandom()
command! -buffer ToggleRepeat     call mpc#ToggleRepeat()
```

Adding Mappings

Back in *Writing Text to a Buffer*, on page 18, you saw how we can call normal-mode commands from a script. Those are commands like we enter all the time while using Vim—`dd` to delete a line, `j` to move down a line, `p` to paste, and so forth.

We've been adding some helpful commands to our plugin, but before we get this out to an actual user, we'll also want to add mappings to those commands, for something closer to the ultimate in ease of use. We want to be able to say something along the lines of, "Hit `Ctrl-x` to play the song." Saying, "Run `:PlaySong(3)` `Enter`" doesn't have quite the same ring to it, does it?

Some of these mappings are going to be usable in our plugin only—and they won't just be specific to our plugin, but they're only going to work in our `mpc` playlist buffer. For the rest of the windows our user might have opened, business will go on as usual.

Let's first see how mappings work in Vim. We'll go through the commands and see a bit of how they're used, and then we'll add the mappings we want for our plugin.

Modes, Mappings, and Recursive Mappings

The most basic command we can use to write a mapping is `:map`.

mappings/map.vim
```
map o   0 " (OK -- maybe don't try that one)
```

But Vim lets us write six different kinds of mappings: they're mode-specific. Among others, we can write mappings that kick in only when we're in normal mode, visual mode, or insert mode.

Mode-specific mappings get their own commands:

mappings/map.vim
```
" Map m to run Ctrl-d (half-page-down) in normal mode
nmap m   <c-d>

" Map ' to type the XML 'right single quote' entity in insert mode
imap '   &#8217;
```

All of these commands come in alternate versions that include the phrase `nore`, *no-recursive*, after their mode's initial. When we define a mapping using those versions, it is not a *recursive mapping*. By default when we use a standard mapping command such as `:nmap` and map a key to something that *includes* that key, we've written a recursive mapping.

So, why are recursive mappings an issue? Let's see if we can find out with an example. Open a new buffer and run the following :nmap command:

mappings/map.vim
```
nmap o  oHello! <esc>A
```

Now hit the o key, which in Vim normally means to drop down one new line and enter insert mode. You should see Vim drop down a new line, enter Hello! followed by a space, and then leave you in insert mode at the end of the line.

Now suppose we wanted to have o drop us down *two* new lines—we could simply change our mapping to execute o twice. Try this one out now. Be ready to hit Ctrl-c:

mappings/map.vim
```
nmap o  o<esc>oHello! <esc>A
```

Did Vim ever make it to that second Esc? No indeed: we mapped o to execute o, Esc, execute o again, and—*oops!*—execute o, Esc, and so on until we stopped it.

Now edit the mapping to look like this and try it out:

mappings/map.vim
```
nnoremap o  o<esc>oHello! <esc>A
```

Much better!

In our first mapping, Vim used the default function of o the first time, and then on the second occurrence it used our own mapping. This is why that particular mapping never got to Esc. In this one, any time that we refer to o in the mapped keys, Vim ignores the current mapping that's remapping that key and interprets it as it normally does.

When to Use nore

The accidental recursive mapping that we just demonstrated may not seem like that dangerous of a threat. If we put adequate thought into the functions we call and the keys we map in our mappings, we might never have an issue with starting infinite loops through a mapping.

Here's the thing, though: mappings are possibly the most commonly modified part of Vim's interface. A user who never makes any more customizations to the editor could easily accumulate a wide-ranging collection of useful mappings or remappings, whether coming across them on GitHub or devising them on his own. And these mappings could easily conflict with ones our plugin introduces. There are only so many keys on a keyboard, after all.

Because we can't predict what mappings a user might be using, the safest practice in packaging VimL for distribution is to always use the nore variants of the mapping commands. (The exception is, of course, when we actually *want* a recursive mapping.) This not only protects us and our user from unpredictable collisions, but also reassures us: whatever we do or might later add down the line, we're not going to re-invoke a command sequence if we're using a nore mapping command.

Arguments: Making Mappings Buffer-Local (and Quiet)

We can write mappings that run Ex commands—including our own commands. Since our mappings specify keys to be entered, including : in the right-hand side of a mapping makes us enter command mode, and from there we can enter a command name and execute it:

mappings/map.vim
```
nnoremap v  :vsplit<cr>
```

This mapping redefines v to run :vsplit, which opens a new vertical split window. (Because v already has a perfectly useful function, I don't recommend actually doing this; this is just an example.) As you can see, we have to use <cr> to run the command in place of actually hitting Enter.

Now look at the command line. If you just entered w, you'll see that it left the :vsplit command there. This is no different from how the command line acts when we run a command ourselves outside of a script, but if we want to hide the command when we run it, we can use <silent>.

mappings/map.vim
```
nnoremap <silent> v  :vsplit<cr>
```

Now when we hit v, we get a new vertical window without cluttering up the command line.

That's one argument. Another one, <buffer>, lets us make our mappings work only in the mpc window, in that window's buffer, like -buffer does for commands. It has to be the first argument we use, and when we use it, it makes the mapping buffer-specific:

mappings/map.vim
```
nnoremap <buffer> <silent> v  :vsplit<cr>
```

If this mapping were in the ftplugin/mpdv.vim file, it would be defined in the mpc window's buffer whenever that buffer is opened. It would therefore work only in the mpc buffer.

That would be a good start for our plugin's mappings—we want them to work only in that one window. But that's not all we can do when we add mappings in a plugin.

Localizing Mappings

Because our mappings are going to be specific to our plugin, what are some ways in which we can distinguish them from the mappings a user might already have? And I did say that we were going to see how a user could reconfigure our mappings to his own liking. Let's see that.

Mapping <SID> Functions

Remember those two variables back in *Constructing a Statusline*, on page 38, that were using that special scope? That was the *script* scope, and we were using that for variables—we had s:count and s:settings. We can use the script scope for function definitions, too:

mappings/map.vim
```
function! s:ColorfulCuteAnimals()
  let animals = ["Phil", "Tom", "Barb", "Bob", "Stacy", "Peary", "Mark",
              \ "Michael"]
  for a in animals
    echom a . " is a tiny animal."
  endfor
endfunction
```

Now, the whole point of script-local variables or functions is that they're available only in the script itself. And for script-local variables, there's no way we can get ahold of them outside of this file, but for functions, we can add a mapping to the file:

mappings/map.vim
```
nnoremap <leader>a   :call <SID>ColorfulCuteAnimals()<cr>
```

We can't just use s: in the mapping. We have to use a special Vim code, <SID>, to access the function. When Vim comes across <SID>, it replaces it with the *script ID*, a random number that acts as a special identifier for just that script. We can see this in action by removing that closing <cr> from the mapping and then running the mapping:

mappings/map.vim
```
nnoremap <leader>a   :call <SID>ColorfulCuteAnimals()

" When we run the mapping, the command line is populated with something like:

:call <SNR>45_ColorfulCuteAnimals()
```

The <SID> trick works only if the mapping is in the same file as the function, but if the two are in the same script, this is a way for us to keep our functions in script scope and allow for them to have mappings. This approach can even be combined with the next one we'll look at, so as to allow users to write their own mappings to script-local functions.

Using <plug>

Vim gives us a way to create mappings to keys that can't be typed, or rather, to *key codes* that can't be typed. <plug> is a special key for which we can write a mapping. For example, we can create a mapping from <plug> to one of our plugin's commands:

mappings/map.vim
```
nnoremap <silent> <buffer> <plug>MpcPlayselectedsong   :PlaySelectedSong<cr>
```

This defines something for our user to map to: <plug>PlaySelectedSong. It's mainly useful in a case where we want our user to be able to write his own mappings for our commands. For example, with our mapping here, our user could add this to his own .vimrc:

mappings/map.vim
```
nmap <leader>p   <plug>MpcPlayselectedsong
```

We name the <plug> mapping by the script name Mpc and command name Playselectedsong. By convention, only the first letter of the script name and the first letter of the command name are uppercase—thus the odd capitalization.

We're going to use one <plug> mapping for our plugin. We'll have :TogglePlayback be a <plug> mapping so that our user can map that command to whatever key or key combination he wants. The other mappings will be buffer-specific.

In ftplugin/mpdv.vim, add these lines:

mappings/mpc/ftplugin/mpdv.vim
```
nnoremap <silent>            <plug>MpcToggleplayback   :TogglePlayback<cr>
nnoremap <silent> <buffer>   <c-x>                     :PlaySelectedSong<cr>
nnoremap <silent> <buffer>   <c-a>                     :ToggleRandom<cr>
nnoremap <silent> <buffer>   <c-e>                     :ToggleRepeat<cr>
```

We've mapped the last three commands to Ctrl sequences. Toggling playback is one function that our user might very well want to use outside of the plugin window, and because :TogglePlayback is *not* a <buffer> command, it will work from anywhere within Vim. In our playlist window, and *only* in our playlist window, the user should be able to hit Ctrl-x to play the song that the cursor is on. To turn repeat and random on and off, he can use Ctrl-e and Ctrl-a…and

what about :TogglePlayback? Let's just add a default mapping of our own to <plug>MpcToggleplayback. We'll use <leader>p.

This is where we learn about the hasmapto() function. Vim provides it to help with the issue of conflicting mappings between plugin setups and user setups. Using this function, we can include a check when we define a new mapping, and make it defined only if the user doesn't already have a mapping in place that's mapped to the same thing:

mappings/mpc/ftplugin/mpdv.vim

```
if !hasmapto("<plug>MpcToggleplayback")
  nmap <leader>p   <plug>MpcToggleplayback
endif
```

In Conclusion

We've just completed the main user interface to version 1.0 of a functioning Vim plugin. That brings us to the end of our plugin project, which means the end of our investigation into VimL!

We've gotten into a variety of Vim-scripting aspects—from coding basic functions to breaking them out into autoloaded files, from writing autocommands to adding a syntax file to writing our own user commands. VimL the language is deeply intertwined with Vim the editor, and as you go on in writing VimL, you can pick any one of these aspects to study and find plenty more to learn about it.

These days we have many good Vim-related websites available dispensing tips, tricks, and general editing wisdom. Even when they're not specifically focused on VimL, there's always more to learn about scripting Vim from digging around on these. There are also plenty of major established Vim plugins available, and once you have a foundational understanding of VimL, the source code for a Vim plugin can be an education in itself.

So, I hope you've enjoyed this introduction. Now go on to great Vim-scripting endeavors! In Appendix 1, *Some Resources*, on page 65, I've listed a few of the sites and plugin repositories that I've found helpful or that have instructional source. Happy Vimming!

APPENDIX 1

Some Resources

Websites

Vim Tips Wiki . . . . . http://vim.wikia.com
The Vim Tips Wiki probably has a tip on how to do anything Vim-related that you could ever want to do, if you can just find it. Two articles of particular interest to new VimLers are at http://vim.wikia.com/wiki/Write_your_own_Vim_function and http://vim.wikia.com/wiki/How_to_write_a_plugin.

Vim FAQ . . . . . http://vimdoc.sourceforge.net/htmldoc/vimfaq.html
Common questions and answers relating to Vim in general. See Section 25 in particular. (It's on "Vim Script Writing.")

Vim Weekly . . . . . http://www.vimweekly.com
An email newsletter: five Vim tips sent weekly. You can browse past issues on the site, and they're full of Vim (and by implication, VimL) goodness.

Vim Wiki . . . . . http://vim-wiki.mawercer.de
A wiki by Marc Weber that aspires to "become the greatest resource about Vim by telling you about the most useful plugins, workflows and settings."

usevim . . . . . http://usevim.com
A blog by Alex Young, with Vim-related links and commentary going back to 2012. Going over the archives is an especially good way to discover helpful scripts and plugins, thanks to the "Script Roundup" and "Plugin Roundup" series.

Wholly Unbalanced Parentheses . . . . . http://of-vim-and-vigor.blogspot.com
"Occasionally coherent observations" by Barry Arthur—often VimL-related.

Plugins

For Plugin Development

ingo-library . . . . . http://www.vim.org/scripts/script.php?script_id=4433
A compilation by Ingo Karkat of utility functions used in his plugins. There's a lot in here, and it's definitely worth looking into; see also his other plugins, which are listed at http://www.vim.org/account/profile.php?user_id=9713 (at "Script Contributions").

genutils . . . . . http://www.vim.org/scripts/script.php?script_id=197
A set of "[miscellaneous] utility functions" for script authors, collected by Hari Krishna Dara.

Vimball . . . . . http://www.vim.org/scripts/script.php?script_id=1502
The archiving utility written by Charles Campbell (Dr. Chip). You can use it to package a plugin as a "vimball," a single archive that can be opened in Vim and then sourced to install the plugin files correctly.

For Plugin Usage

As I suggested back in *The Structure of a Vim Plugin*, on page 12, creative Vim users have come up with a few systems that simplify the processes of installing, updating, and removing plugins. Here are a few of the most common.

Pathogen . . . . . http://www.vim.org/scripts/script.php?script_id=2332
The "poor man's package manager" that started it all, written by the famed Tim Pope. It's also on GitHub; see the README at https://github.com/tpope/vim-pathogen for splendid instructions.

Vundle . . . . . https://github.com/gmarik/Vundle.vim
Inspired by Pathogen. This is what I use; both are working with Git, but Vundle provides commands for the basic things, such as installing and updating plugins, and automates most other things.

NeoBundle . . . . . https://github.com/Shougo/neobundle.vim
Inspired by Vundle! It's advertised as being "good for plugin power users," so if you want a plugin manager that can do all the things, this is probably what you want.

See Also (for Inspiration)

rails.vim . . . . . https://github.com/tpope/vim-rails/
The Vim plugin for Rails, by Tim Pope—highly worth digging into (even if you, like me, prefer Grails).

vim-airline . . . . . https://github.com/bling/vim-airline
Bailey Ling's outstandingly popular Vim statusline solution. Pay particular attention to its use of autoload.

ctrlp.vim . . . . . https://github.com/kien/ctrlp.vim
A fuzzy-finding plugin that, unlike some of the others, is written in pure VimL.

ack.vim . . . . . https://github.com/mileszs/ack.vim
Miles Sterrett's Vim plugin for Ack, the "tool like grep, optimized for programmers." See its help file and use of the `ftplugin` directory.

Netrw . . . . . http://www.vim.org/scripts/script.php?script_id=1075
The standard plugin for network-involving reading and writing of files, by Charles Campbell—an excellent example of working with the underlying operating system. It's bundled as a vimball; you can also view the source directly at the GitHub mirror, https://github.com/vim-scripts/netrw.vim.

Seek . . . . . https://github.com/goldfeld/vim-seek
A plugin by Vic Goldfeld that uses `s` to jump to a part of the current line (the same as `f`, but Seek's motion takes two characters). This is a great example of using VimL to add a Vim motion.

Bibliography

[Nei12] Drew Neil. *Practical Vim: Edit Text at the Speed of Thought*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2012.

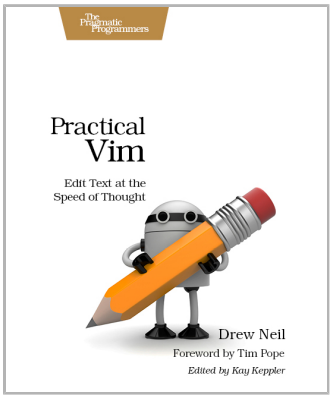
Long Live the Command Line!

Use tmux and Vim for incredible mouse-free productivity.

Practical Vim

Vim is a fast and efficient text editor that will make you a faster and more efficient developer. It's available on almost every OS—if you master the techniques in this book, you'll never need another text editor. In more than 100 Vim tips, you'll quickly learn the editor's core functionality and tackle your trickiest editing and writing tasks.

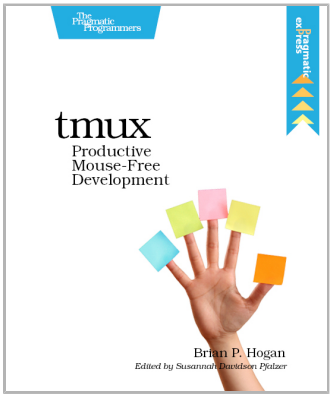
Drew Neil
(346 pages) ISBN: 9781934356982. $29
https://pragprog.com/book/dnvim



tmux

Your mouse is slowing you down. The time you spend context switching between your editor and your consoles eats away at your productivity. Take control of your environment with tmux, a terminal multiplexer that you can tailor to your workflow. Learn how to customize, script, and leverage tmux's unique abilities and keep your fingers on your keyboard's home row.

Brian P. Hogan
(88 pages) ISBN: 9781934356968. $16.25
https://pragprog.com/book/bhtmux



The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
https://pragprog.com/book/bkviml
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
https://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
https://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
https://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/bkviml*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764