The VimL Primer

Edit Like a Pro with Vim Plugins and Scripts

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

The VimL Primer

Edit Like a Pro with Vim Plugins and Scripts

Benjamin Klein

Edited by Lynn Beighley and Fahmida Y. Rashid

The VimL Primer

Edit Like a Pro with Vim Plugins and Scripts

Benjamin Klein

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

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

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

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

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 ?, 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 ?, 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]

```
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.

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

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.