Extracted from:

Practical Vim, Second Edition

Edit Text at the Speed of Thought

The Pragmatic Bookshelf

Raleigh, North Carolina

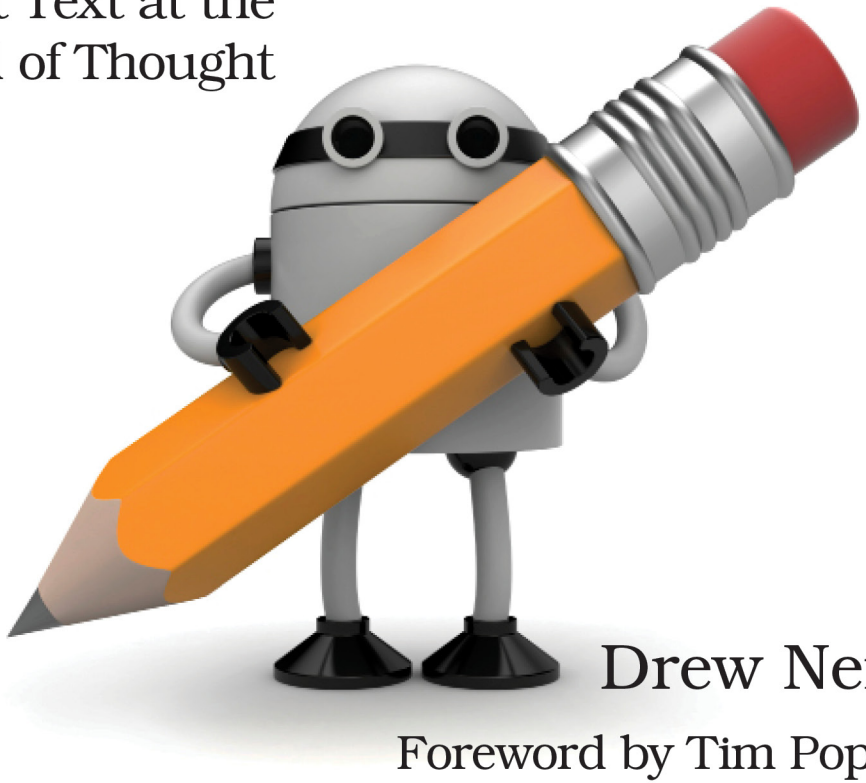
The Pragmatic Programmers

FOR Vim 8

Practical Vim

Second Edition

Edit Text at the Speed of Thought



Drew Neil

Foreword by Tim Pope

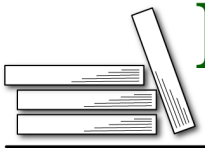
Practical Vim, Second Edition

Edit Text at the Speed of Thought

Drew Neil

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Katharine Dvorak (editor)
Potomac Indexing, LLC (index)
Cathleen Small (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

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

Printed in the United States of America.
ISBN-13: 978-1-68050-127-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—January 2017

Tip 56

Traverse the Jump List

Vim records our location before and after making a jump and provides a couple of commands for retracing our steps.

In web browsers, we're used to using the back button to return to pages that we visited earlier. Vim provides a similar feature by way of the *jump list*: the `<C-o>` command is like the back button, while the complementary `<C-i>` command is like the forward button. These commands allow us to traverse Vim's jump list, but what exactly is a jump?

Let's start by making this distinction: motions move around *within a file*, whereas jumps can move *between files* (although we'll soon see that some motions are also classified as jumps). We can inspect the contents of the jump list by running this command:

```
⇛ :jumps
❮  jump line  col file/text
     4    12    2 <recipe id="sec.jump.list">
     3   114    2 <recipe id="sec.change.list">
     2   169    2 <recipe id="sec.gf">
     1   290    2 <recipe id="sec.global.marks">
>
Press Enter or type command to continue
```

Any command that changes the active file for the current window can be described as a jump. In the jump list, Vim records the cursor location before and after running such a command. For example, if we run the :edit command to open a new file (see Tip 42, *Open a File by Its Filepath Using ':edit'*, on page ?), then we can use the `<C-o>` and `<C-i>` commands to jump back and forth between the two files.

Moving directly to a line number with `[count]G` counts as a jump, but moving up or down one line at a time does not. The sentence-wise and paragraph-wise motions are jumps, but the character-wise and word-wise motions are not. As a rule of thumb, we could say that long-range motions may be classified as a jump, but short-range motions are just motions.

This table summarizes a selection of jumps:

Command	Effect
`[count]G`	Jump to line number

Command	Effect
/pattern<CR> /?pattern<CR> / n / N	Jump to next/previous occurrence of pattern
%	Jump to matching parenthesis
(/)	Jump to start of previous/next sentence
{ / }	Jump to start of previous/next paragraph
H / M / L	Jump to top/middle/bottom of screen
gf	Jump to file name under the cursor
<C-]>	Jump to definition of keyword under the cursor
'{mark} / `{mark}	Jump to a mark

The <C-o> and <C-i> commands themselves are never treated as a motion. This means that we can't use them to extend the reach of a Visual mode selection, nor can we use them in Operator-Pending mode. I tend to think of the jump list as a breadcrumb trail that makes it easy to retrace my steps through the files that I've visited during the course of an editing session.

Vim can maintain multiple jump lists at the same time. In fact, each separate window has its own jump list. If we're using split windows or multiple tab pages, then the <C-o> and <C-i> commands will always be scoped to the jump list of the active window.

Beware of Mapping the Tab Key

Try pressing <C-i> in Insert mode, and you should find that it has the same effect as pressing the <Tab> key. That's because Vim sees <C-i> and <Tab> as the same thing.

Beware that if you attempt to create a mapping for the <Tab> key, it will also be triggered when you press <C-i> (and vice versa). That may not seem like a problem, but consider this: if you map the <Tab> key to something else, it will *overwrite* the default behavior of the <C-i> command. Think carefully about whether that's a worthwhile trade-off. The jump list is much less useful if you can only traverse it in one direction.

Tip 57

Traverse the Change List

Vim records the location of our cursor after each change we make to a document. Traversing this change list is simple and can be the quickest way to get where we want to go.

Have you ever used the undo command followed immediately by redo? The two commands cancel each other out, but they have the side effect of placing the cursor on the most recent change. That could be useful if we wanted to jump back to the part of the document that we edited most recently. It's a hack, but `u<C-r>` gets us there.

It turns out that Vim maintains a list of the modifications we make to each buffer during the course of an editing session. It's called the *change list* (see :h changelist), and we can inspect its contents by running the following:

```
⇛ :changes
❮ change line  col text
      3     1    8 Line one
      2     2    7 Line two
      1     3    9 Line three
  >
  Press ENTER or type command to continue
```

This example output shows that Vim records the line and column number for each change. Using the `g;` and `g,` commands, we can traverse backward and forward through the change list. As a memory aid for `g;` and `g,`, it may help to remember that the `;` and `,` commands can be used to repeat or reverse the `f{char}` command (see Tip 50,*Find by Character*, on page ?).

To jump back to the most recent modification in the document, we press `g;`. That places the cursor back on the line and column where it ended up after the previous edit. The result is the same as if we had pressed `u<C-r>`, except that we don't make any transitory changes to the document.

Marks for the Last Change

Vim automatically creates a couple of marks that complement the change list. The `` `. `` mark always references the position of the last change (:h `.), while the `` `^ `` mark tracks the position of the cursor the last time that Insert mode was stopped (:h `^).

In most scenarios, jumping to the `` `. `` mark has the same effect as the `g;` command. Whereas the mark can only refer to the position of the most recent change, the change list stores multiple locations. We can press `g;` again and again, and each time it takes us to a location that was recorded earlier in the change list. The `` `. ``, on the other hand, will always take us to the last item in the change list.

The `` `^ `` mark references the last *insertion*, which is slightly more specific than the last *change*. If we leave Insert mode and then scroll around the document, we can quickly carry on where we left off by pressing `gi` (:h gi). In a single

move, that uses the `^ mark to restore the cursor position and then switches back into Insert mode. It's a great little time saver!

Vim maintains a change list for each individual buffer in an editing session. By contrast, a separate jump list is created for each window.

Tip 58

Jump to the Filename Under the Cursor

Vim treats filenames in our document as a kind of hyperlink. When configured properly, we can use the gf command to go to the filename under the cursor.

Let's demonstrate with the jumps directory, from the source files distributed with this book. It contains the following directory tree:

```
practical_vim.rb
practical_vim/
  core.rb
  jumps.rb
  more.rb
  motions.rb
```

In the shell, we'll start by changing to the jumps directory and then launching Vim. For this demonstration, I recommend using the -u NONE -N flags to ensure that Vim starts up without loading any plugins:

```
⇛ $ cd code/jumps
⇛ $ vim -u NONE -N practical_vim.rb
```

The practical_vim.rb file does nothing more than load the contents of the core.rb and more.rb files:

jumps/practical_vim.rb
```
require 'practical_vim/core'
require 'practical_vim/more'
```

Wouldn't it be useful if we could quickly inspect the contents of the file specified by the require directive? That's what Vim's gf command is for. Think of it as *go to file* (:h gf).

Let's try it out. We'll start by placing our cursor somewhere inside the 'practical_vim/core' string (for example, pressing fp would get us there quickly). If we try using the gf command now, we get this error: "E447: Can't find file 'practical_vim/core' in path."

Vim tries to open a file called practical_vim/core and reports that it doesn't exist, but there is a file called practical_vim/core.rb (note the file extension). Somehow we need to instruct Vim to modify the filepath under the cursor by appending the .rb file extension before attempting to open it. We can do this with the 'suffixesadd' option.

Specify a File Extension

The 'suffixesadd' option allows us to specify one or more file extensions, which Vim will attempt to use when looking up a filename with the `gf` command (:h 'suffixesadd'ⓘ). We can set it up by running this command:

```
⇛ :set suffixesadd+=.rb
```

Now when we use the `gf` command, Vim jumps directly to the filepath under the cursor. Try using it to open more.rb. In that file, you'll find a couple of other require declarations. Pick one, and open it up using the `gf` command.

Each time we use the `gf` command, Vim adds a record to the jump list, so we can always go back to where we came from using the `<C-o>` command (see Tip 56,*Traverse the Jump List*, on page 5). In this case, pressing `<C-o>` the first time would take us back to more.rb, and pressing it a second time would take us back to practical_vim.rb.

Specify the Directories to Look Inside

In this example, each of the files referenced with the require statement was located relative to the working directory. But what if we referenced functionality that was provided by a third-party library, such as a rubygem?

That's where the 'path' option comes in (:h 'path'ⓘ). We can configure this to reference a comma-separated list of directories. When we use the `gf` command, Vim checks each of the directories listed in 'path' to see if it contains a filename that matches the text under the cursor. The 'path' setting is also used by the :find command, which we covered in Tip 43,*Open a File by Its Filename Using ':find'*, on page ?.

We can inspect the value of the path by running this command:

```
⇛ :set path?
❮ path=.,/usr/include,,
```

In this context, the . stands for the directory of the current file, whereas the empty string (delimited by two adjacent commas) stands for the working directory. The default settings work fine for this simple example, but for a

larger project we would want to configure the 'path' setting to include a few more directories.

For example, it would be useful if the 'path' included the directories for all rubygems used in a Ruby project. Then we could use the `gf` command to open up the modules referenced by any require statements. For an automated solution, check out Tim Pope's bundler.vim plugin,[1] which uses the project Gemfile to populate the 'path' setting.

Discussion

In the setup for this tip, I recommended launching Vim with plugins disabled. That's because Vim is usually distributed with a Ruby file-type plugin, which handles the setup of 'suffixesadd' and 'path' options for us. If you do a lot of work with Ruby, I recommend getting the latest version of the file-type plugin from github because it's actively maintained.[2]

The 'suffixesadd' and 'path' options can be set locally for each buffer, so they can be configured in different ways for different file types. Vim is distributed with file-type plugins for many languages besides Ruby, so in practice you won't often have to set these options yourself. Even so, it's worth understanding how the `gf` command works. It makes each filepath in our document behave like a hyperlink, which makes it easier to navigate through a codebase.

The `<C-]>` command has a similar role. It also requires a bit of setup (as discussed in Tip 103,*Configure Vim to Work with ctags*, on page ?), but when it's correctly configured, it allows us to jump from any method invocation directly to the place where it was defined. Skip ahead to Tip 104,*Navigate Keyword Definitions with Vim's Tag Navigation Commands*, on page ?, for a demonstration.

While the jump list and change list are like breadcrumb trails that allow us to retrace our steps, the `gf` and `<C-]>` commands provide wormholes that transport us from one part of our codebase to another.

1. https://github.com/tpope/vim-bundler
2. https://github.com/vim-ruby/vim-ruby