Extracted from:

Practical Vim

Edit Text at the Speed of Thought

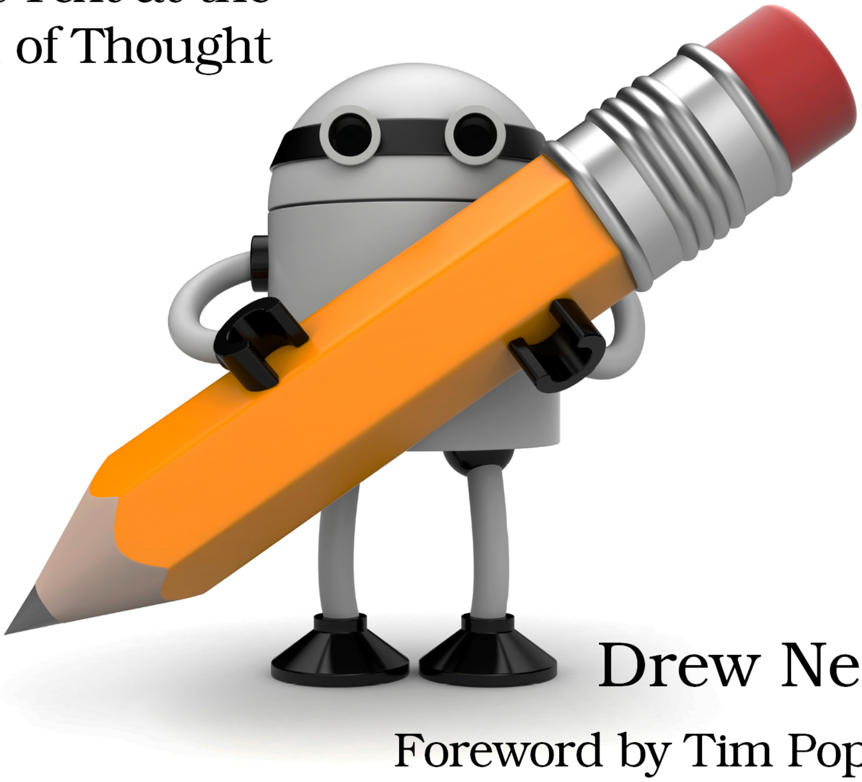
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Practical Vim

Edit Text at the Speed of Thought



Drew Neil

Foreword by Tim Pope

Edited by Kay Keppler

The team that produced this book includes:

Kay Keppler (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J. Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-98-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2012

Read Me

Practical Vim is for programmers who want to raise their game. You've heard it said that in the hands of an expert, Vim shreds text at the speed of thought. Reading this book is your next step toward that end.

Practical Vim is a fast track to Vim mastery. It won't hold you by the hand, but beginners can find the prerequisite knowledge by running through the Vim tutor, an interactive lesson distributed with Vim.[1] *Practical Vim* builds on this foundation by highlighting core concepts and demonstrating idiomatic usage.

Vim is highly configurable. However, customization is a personal thing, so I've tried to avoid recommending what should or should not go into your vimrc file. Instead, *Practical Vim* focuses on the core functionality of the editor—the stuff that's always there, whether you're working over SSH on a remote server or using a local instance of GVim, with plugins installed to add extra functionality. Master Vim's core, and you'll gain portable access to a text editing power tool.

How This Book Is Structured

Practical Vim is a recipe book. It's not designed to be read from start to finish. (I mean it! At the start of the next chapter, I'll advise you to skip it and jump straight to the action.) Each chapter is a collection of tips that are related by a theme, and each tip demonstrates a particular feature in action. Some tips are self-contained. Others depend upon material elsewhere in the book. Those tips are cross-referenced so you can find everything easily.

Practical Vim doesn't progress from novice to advanced level, but each individual chapter does. A less-experienced reader might prefer to make a first pass through the book, reading just the early tips in each chapter. A more advanced reader might choose to focus on the later tips or move around the book as needed.

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

A Note on the Examples

In Vim, there's always more than one way to complete any given task. For example, in Chapter 1, *The Vim Way*, on page ?, all of the problems are designed to illustrate an application of the dot command, but every one of them could also be solved using the `:substitute` command.

On seeing my solution, you might think to yourself, "Wouldn't it be quicker to do it *this way*?" And you may well be right! My solutions illustrate a particular technique. Look beyond their simplicity, and try to find a resemblance to the problems that you face daily. That's where these techniques will save you time.

Learn to Touch Type, Then Learn Vim

If you have to look down to find the keys on the keyboard, the benefits of learning Vim won't come fast. Learning to touch type is imperative.

Vim traces its ancestry back to the classic Unix editors, `vi` and `ed` (see *On the Etymology of Vim (and Family)*, on page ?). These predate the mouse and all of the point-and-click interfaces that came with it. In Vim, everything can be done with the keyboard. For the touch typist, that means Vim does everything *faster*.

Read the Forgotten Manual

In *Practical Vim*, I demonstrate by showing examples rather than by describing them. That's not easy to do with the written word. To show the steps taken during an interactive editing session, I've adopted a simple notation that illustrates the keystrokes and the contents of a Vim buffer side by side.

If you're keen to jump to the action, you can safely skip this chapter for now. It describes each of the conventions used throughout *Practical Vim*, many of which you'll find to be self-explanatory. At some point, you'll probably come across a symbol and wonder what it stands for. When that happens, turn back and consult this chapter for the answer.

Get to Know Vim's Built-in Documentation

The best way to get to know Vim's documentation is by spending time in it. To help out, I've included "hyperlinks" for entries in Vim's documentation. For example, here's the "hyperlink" for the Vim tutor: `:h vimtutor` ⓘ.

The icon has a dual function. First, it serves as a signpost, drawing the eye to these helpful references. Second, if you're reading this on an electronic device that's connected to the Internet, you can click the icon and it will take you to the relevant entry in Vim's online documentation. In this sense, it truly is a hyperlink.

But what if you're reading the paper edition of the book? Not to worry. If you have an installation of Vim within reach, simply enter the command as it appears in front of the icon.

For example, type `:h vimtutor` (`:h` is an abbreviation for the `:help` command). Consider this a unique address for the documentation on vimtutor: a URL of sorts. In this sense, the help reference is a kind of hyperlink to Vim's built-in documentation.

Notation for Simulating Vim on the Page

Vim's modal interface sets it apart from most other text editors. To make a musical analogy, let's compare the Qwerty and piano keyboards. A pianist can pick out a melody by playing one note at a time or he or she can hold down several keys at once to sound a chord. In most text editors, keyboard shortcuts are triggered by pressing a key while holding down one or more modifier buttons, such as the control and command keys. This is the Qwerty equivalent of playing a chord on the piano keyboard.

Some of Vim's commands are also triggered by playing chords, but Normal mode commands are designed to be typed as a sequence of keystrokes. It's the Qwerty equivalent of playing a melody on the piano keyboard.

`Ctrl`-`s` is a common convention for representing chordal key commands. It means "Press the Control key and the `s` key at the same time." But this convention isn't well suited to describing Vim's modal command set. In this section, we'll meet the notation used throughout *Practical Vim* to illustrate Vim usage.

Playing Melodies

In Normal mode, we compose commands by typing one or more keystrokes in sequence. These commands appear as follows:

Notation	Meaning
`x`	Press `x` once
`dw`	In sequence, press `d`, then `w`
`dap`	In sequence, press `d`, `a`, then `p`

Most of these sequences involve two or three keystrokes, but some are longer. Deciphering the meaning of Vim's Normal mode command sequences can be challenging, but you'll get better at it with practice.

Playing Chords

When you see a keystroke such as `<C-p>`, it doesn't mean "Press `<`, then `C`, then `-`, and so on." The `<C-p>` notation is equivalent to `Ctrl`-`p`, which means "Press the `<Ctrl>` and `p` keys at the same time."

I didn't choose this notation without good reason. Vim's documentation uses it (:h key-notation ⓘ), and we can also use it in defining custom key mappings. Some of Vim's commands are formed by combining chords and keystrokes in sequence, and this notation handles them well. Consider these examples:

Notation	Meaning
`<C-n>`	Press `<Ctrl>` and `n` at the same time
`g<C-]>`	Press `g`, followed by `<Ctrl>` and `]` at the same time
`<C-r>0`	Press `<Ctrl>` and `r` at the same time, then `0`
`<C-w><C-=>`	Press `<Ctrl>` and `w` at the same time, then `<Ctrl>` and `=` at the same time

Placeholders

Many of Vim's commands require two or more keystrokes to be entered in sequence. Some commands must be followed by a particular kind of keystroke, while other commands can be followed by any key on the keyboard. I use curly braces to denote the set of valid keystrokes that can follow a command. Here are some examples:

Notation	Meaning
`f{char}`	Press `f`, followed by any other character
`` `{a-z} ``	Press `` ` ``, followed by any lowercase letter
`m{a-zA-Z}`	Press `m`, followed by any lowercase or uppercase letter
`d{motion}`	Press `d`, followed by any motion command
`<C-r>{register}`	Press `<Ctrl>` and `r` at the same time, followed by the address of a register

Showing Special Keys

Some keys are called by name. This table shows a selection of them:

Notation	Meaning
`<Esc>`	Press the Escape key
`<CR>`	Press the carriage return key (also known as `<Enter>`)
`<Ctrl>`	Press the Control key
`<Tab>`	Press the Tab key
`<Shift>`	Press the Shift key
`<S-Tab>`	Press the `<Shift>` and `<Tab>` keys at the same time
`<Up>`	Press the up arrow key
`<Down>`	Press the down arrow key
`␣`	Press the space bar

Note that the space bar is represented as `␣`. This could be combined with the `f{char}` command to form `f␣`.

Switching Modes Midcommand

When operating Vim, it's common to switch from Normal to Insert mode and back again. Each keystroke could mean something different, depending on which mode is active. I've used an alternative style to represent keystrokes entered in Insert mode, which makes it easy to differentiate them from Normal mode keystrokes.

Consider this example: `cw`replacement<Esc>. The Normal mode `cw` command deletes to the end of the current word and switches to Insert mode. Then we type the word "replacement" in Insert mode and press `<Esc>` to switch back to Normal mode again.

The Normal mode styling is also used for Visual mode keystrokes, while the Insert mode styling can indicate keystrokes entered in Command-Line mode and Replace mode. Which mode is active should be clear from context.

Interacting with the Command Line

In some tips we'll execute a command line, either in the shell or from inside Vim. This is what it looks like when we execute the grep command in the shell:

```
⇒ $ grep -n Waldo *
```

And this is how it looks when we execute Vim's built-in :grep command:

```
⇒ :grep Waldo *
```

In *Practical Vim*, the $ symbol indicates that a command line is to be executed in an external shell, whereas the : prompt indicates that the command line is to be executed internally from Command-Line mode. Occasionally we'll see other prompts, including these:

Prompt	Meaning
$	Enter the command line in an external shell
:	Use Command-Line mode to execute an Ex command
/	Use Command-Line mode to perform a forward search
?	Use Command-Line mode to perform a backward search
=	Use Command-Line mode to evaluate a Vim script expression

Any time you see an Ex command listed inline, such as :write, you can assume that the <CR> key is pressed to execute the command. Nothing happens otherwise, so you can consider <CR> to be implicit.

By contrast, Vim's search command allows us to preview the first match before pressing <CR> (see Tip 81, on page ?). When you see a search command listed inline, such as /pattern<CR>, the <CR> keystroke is listed explicitly. If the <CR> is omitted, that's intentional, and it means you shouldn't press the Enter key just yet.

Showing the Cursor Position in a Buffer

When showing the contents of a buffer, it's useful to be able to indicate where the cursor is positioned. In this example, you should see that the cursor is placed on the first letter of the word "One":

```
One two three
```

When we make a change that involves several steps, the contents of the buffer pass through intermediate states. To illustrate the process, I use a table showing the commands executed in the left column and the contents of the buffer in the right column. Here's a simple example:

Keystrokes	Buffer Contents
{start}	`One two three`
dw	`two three`

In row 2 we run the dw command to delete the word under the cursor. We can see how the buffer looks immediately after running this command by looking at the contents of the buffer in the same row.

Highlighting Search Matches

When demonstrating Vim's search command, it's helpful to be able to highlight any matches that occur in the buffer. In this example, searching for the string "the" causes four occurrences of the pattern to be highlighted:

Keystrokes	Buffer Contents

{start}	`the problem with these new recruits is that`
	`they don't keep their boots clean.`

/the<CR>	`the problem with these new recruits is that`
	`they don't keep their boots clean.`

Skip ahead to Tip 80, on page ?, to find out how to enable search highlighting in Vim.

Selecting Text in Visual Mode

Visual mode allows us to select text in the buffer and then operate on the selection. In this example, we use the `it` text object to select the contents of the `<a>` tag:

Keystrokes	Buffer Contents
{start}	`<a href="http://pragprog.com/dnvim/">Practical Vim</a>`
`vit`	`<a href="http://pragprog.com/dnvim/">Practical Vim</a>`

Note that the styling for a Visual selection is the same for highlighted search matches. When you see this style, it should be clear from context whether it represents a search match or a Visual selection.

Downloading the Examples

The examples in *Practical Vim* usually begin by showing the contents of a file *before* we change it. These code listings include the file path:

macros/incremental.txt
```
partridge in a pear tree
turtle doves
French hens
calling birds
golden rings
```

Each time you see a file listed with its file path in this manner, it means that you can download the example. I recommend that you open the file in Vim and try out the exercises for yourself. It's the best way to learn!

To follow along, download all the examples and source code from the Pragmatic Bookshelf.[1] If you're reading on an electronic device that's connected to the Internet, you can also fetch each file one by one by clicking on the filename. Try it with the example above.

Use Vim's Factory Settings

Vim is highly configurable. If you don't like the defaults, then you can change them. That's a good thing, but it could cause confusion if you follow the examples in this book using a customized version of Vim. You may find that some things don't work for you the way that they are described in the text. If you suspect that your customizations are causing interference, here's a quick test. Try quitting Vim and then launching it with these options:

```
⇒ $ vim -u NONE -N
```

1. http://pragprog.com/titles/dnvim/source_code

The -u NONE flag tells Vim not to source your vimrc on startup. That way, your customizations won't be applied and plugins will be disabled. When Vim starts up without loading a vimrc file, it reverts to vi compatible mode, which causes many useful features to be disabled. The -N flag prevents this by setting the 'nocompatible' option.

For most examples in *Practical Vim*, the vim -u NONE -N trick should guarantee that you get the same experience as described, but there are a couple of exceptions. Some of Vim's built-in features are implemented with Vim script, which means that they will only work when plugins are enabled. This file contains the absolute minimum configuration that is required to activate Vim's built-in plugins:

essential.vim
```
set nocompatible
filetype plugin on
```

When launching Vim, you can use this file instead of your vimrc by running the following:

```
⇒ $ vim -u code/essential.vim
```

You'll have to adjust the code/essential.vim path accordingly. With Vim's built-in plugins enabled, you'll be able to use features such as netrw (Tip 43, on page ?) and omni-completion (Tip 117, on page ?), as well as many others. I consider Vim's factory settings to mean built-in plugins enabled and vi compatibility disabled.

Look out for subsections titled "Preparation" at the top of a tip. To follow along with the material in these tips, you'll need to configure Vim accordingly. If you start up with Vim's factory settings and then apply the customizations on the fly, you should be able to reproduce the steps from these tips without any problems.

If you're still having problems, see Section 6, *On Vim Versions*, on page xiv.

On the Role of Vim Script

Vim script enables us to add new functionality to Vim or to change existing functionality. It's a complete scripting language in itself and a subject worthy of a book of its own. *Practical Vim* is not that book.

But we won't steer clear of the subject entirely. Vim script is always just below the surface, ready to do our bidding. We'll see a few examples of how it can be used for everyday tasks in Tip 16, on page ?; Tip 70, on page ?; Tip 94, on page ?; and Tip 95, on page ?.

Practical Vim shows you how to get by with Vim's core functionality. In other words, no third-party plugins assumed. I've made an exception for Tip 86, on page ?, and Tip 96, on page ?. In each case, the plugin I've recommended adds a feature that I find indispensable. And in each case, the plugin requires very little code—less than ten lines of Vim script. Both examples demonstrate how easily Vim's functionality can be extended. The implementation of visual-star.vim and Qargs.vim is presented inline without explanation. This should give you an idea of what Vim script looks like and what you can accomplish with it. If it piques your interest, then so much the better.

On Vim Versions

All examples in *Practical Vim* were tested on the latest version of Vim, which was 7.3 at the time of writing. That said, most examples should work fine on any 7.x release, and many of the features discussed are also available in 6.x.

Some of Vim's functionality can be disabled during compilation. For example, when configuring the build, we could provide the --with-features=tiny option, which would disable all but the most fundamental features (there are also feature sets labeled small, normal, big, and huge). You can browse the feature list by looking up :h +feature-list ⓘ.

If you find that you're missing a feature discussed in this book, you might be using a minimal Vim build. Check whether or not the feature is available to you with the :version command:

```
⇒ :version
❮ VIM - Vi IMproved 7.3 (2010 Aug 15, compiled Sep 18 2011 16:00:17)
  Huge version with MacVim GUI.  Features included (+) or not (-):
  +arabic +autocmd +balloon_eval +browse +builtin_terms +byte_offset
  +cindent +clientserver +clipboard +cmdline_compl +cmdline_hist
  +cmdline_info +comments
  ...
```

On a modern computer, there's no reason to use anything less than Vim's huge feature set!

Vim in the Terminal or Vim with a GUI? You Choose!

Traditionally, Vim runs inside of the terminal, with no graphical user interface (GUI). We could say instead that Vim has a TUI: a textual user interface. If you spend most of your day at the command line, this will feel natural.

If you're accustomed to using a GUI-based text editor, then GVim (or MacVim on OS X) will provide a helpful bridge into the world of Vim (see :h gui ⓘ). GVim supports more fonts and more colors for syntax highlighting. Also, you can use the mouse. And some of the conventions of the operating system are honored. For example, in MacVim you can interact with the system clipboard using `Cmd`-`X` and `Cmd`-`V`, save a document with `Cmd`-`S`, or close a window with `Cmd`-`W`. Use these while you find your bearings, but be aware that there's always a better way.

For the purposes of this book, it doesn't matter whether you run Vim in the terminal or as GVim. We'll focus on core commands that work just as well in either. We'll learn how to do things *the Vim way*.