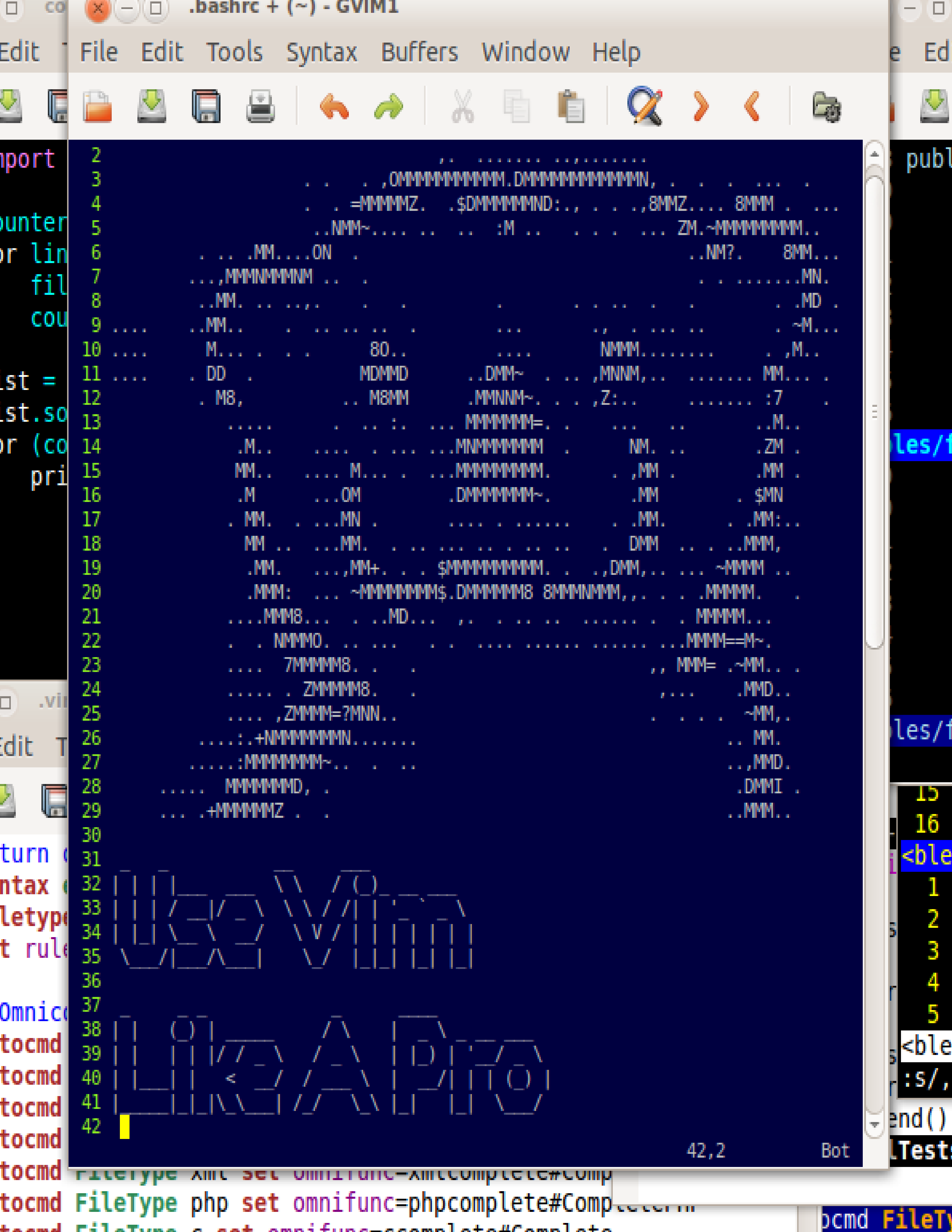

.bashrc + (~) - GVIM1
File Edit Tools Syntax Buffers Window Help
Use Vim
Like A Pro
42,2 Bot

Use Vim Like A Pro

Go from noob to pro

Tim Ottinger

This book is for sale at http://leanpub.com/VimLikeAPro

This version was published on 2014-06-22

Tweet This Book!

Please help Tim Ottinger by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I am ready to #UseVimLikeAPro

The suggested hashtag for this book is #vimLikeAPro.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#vimLikeAPro

Contents

Why Bother? (reasons)

There are many other editors. Several are excellent. There is no reason why you cannot use all of them.

You might want to learn vim for any of these reasons:

- With the sudden rise in Unix use (Linux and Mac OS X, in particular) the text editor known as *vim* (“vi improved”) has become ubiquitous
- *vim* has a small footprint in RAM and on the CPU. A given system can support a great many *vim* users at once.
- *vim* has a lot of “superpowers”, which make editing quite efficient.
- *vim* has “geek appeal”.
- *vim* has a very active user/developer community. It always has.
- Learning new stuff is good for your brain.

Why Write This Tutorial (approach)

Some other tutorials are very good, and google/yahoo/bing/whatever can help you find them all.

There are also some great books that have been written since I started this tutorial. Those books are far more comprehensive and have had a lot of investment from their publishers and editors.

This little tutorial has been around for a long time and has been strangely popular. I like to think it is because I have taken a slightly different approach.

I wrote this for the impatient developer.

There is a certain mental model that makes mastering *vim* much faster. I don’t know any other materials that use the same approach, or which teach as deeply in such a small space.

I’ve agonized and organized (and re-agonized, and reorganized) the tutorial for top-to-bottom learning, so that anyone who emerges from the other end of this tutorial will have professional-grade editing skills, probably better than many of their more experienced colleagues.

When you are done here, you may want to invest in a much more comprehensive book. I am keenly, painfully aware of how much material I have intentionally left out.

You’ll be pleased to know that I continue to look for things to leave out, or faster ways to shrink the content. Well, that is, other than this apologetic section.

I think this is one of the fastest ways to improve your use of *vim*, and a pretty good way to start using *vim* from scratch.

This work started out as a web page, for free. Now I am using leanpub because I like the formats and styling I can get with their system.

I stil have a "suggested price" of zero dollars. Free. Gratis.

Leanpub has a nice system that allows people to give me small monetary gifts if they want to. I have had some pizza and scotch money that I would not otherwise have had. Thank you for the kindness you have shown. It is more than I expected.

How should one use the tutorial? (usage)

Look at each subsection heading as the beginning of a separate lesson, and spend a little time with it before moving onward. Maybe spend a day with each bit of knowledge, and maybe a several days when the lesson is particularly meaty.

Don't be in a hurry. Don't rush your brain, lest you forget old things as fast as you learn new ones. Consider doing a few lessons a week. People have used *vim* for 10 years and still don't know half as much as you'll learn in the first major section; you can take a few months to work through this.

You can't learn *vim* without using *vim*, so you should have some text files (preferably open source program code) to work with. It is better yet if you are using *vim* at work. It also helps if you work with a partner who is also reading this tutorial, so that you can reinforce each other.

What can I do with this tutorial? (license)

This work is licensed under a Creative Commons Attribution 3.0 License[1].

Copy it, share it, paste it into your web page. Don't pretend it is your own stuff, and please give me some attribution. As a courtesy, if you find it worth distributing, I wouldn't mind getting a copy or a link. Just let me know[2].

[1]http://creativecommons.org/licenses/by/3.0/

[2]mailto:tottinge@gmail.com

Master The Basics

A little reassurance first.

Nobody knows all of *vim*. Nobody needs to know it all. You only need to know how to do your own work. The secret is to not settle for crummy ways of doing work.

vim has word completion, and undo, and shortcuts, and abbreviations, and keyboard customization, and macros, and scripts. You can turn this into *your* editor for *your* environment.

That is cool, but it may be reassuring to know that you can probably will not need to. You can be far more productive without touching any of deeply advanced features.

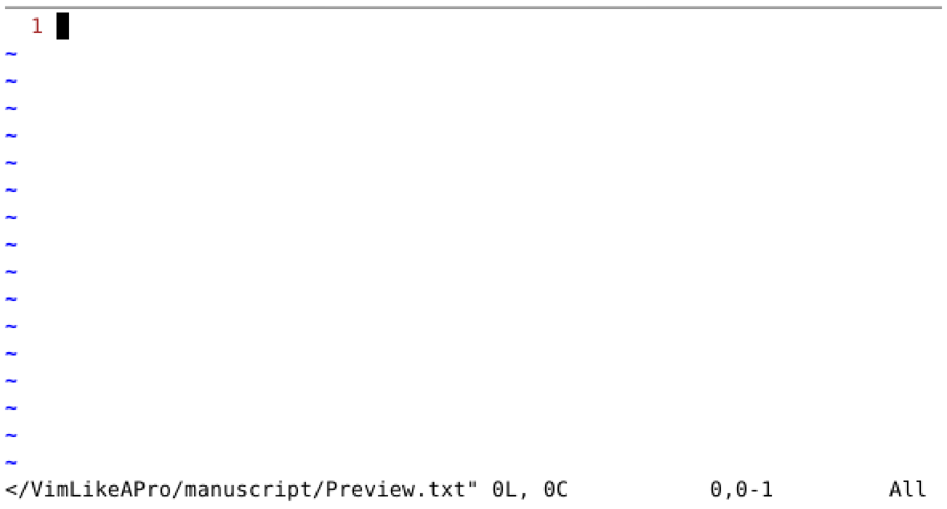
As Bram Moolenaar (*vim*'s primary author) says, the best way to learn *vim* is to use it and ask questions. This little tutorial is full of questions you might not have thought to ask. That's the main value I can give you.

vim has a built-in tutorial. You might want to try it, especially if you don't like my tutorial. All you have to do is type "vimtutor" at the command line. It is a very nice tutorial, and is rather complete (compared to mine, which is fairly nice but not very complete at all).

Finally, please consider GVIM. It will make your experience much more pleasant. If you only have *vim*, then you can still use it and learn, but GVIM has a much nicer look, lets you use your mouse and scroll wheel, and has menus and icons for those of you who are used to such things.

What does it look like?

It does not look like much. It was not built for beauty. *vim* uses the default terminal appearance. GVim adds menu bars and stuff, but *vim* looks like this:



```
  1 █
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
</VimLikeAPro/manuscript/Preview.txt" 0L, 0C          0,0-1         All
```

a screenshot of *vim* in action

As far as visible features, there is:

- the line number (I have line numbers turned on by default, you might not),
- a bunch of tildes (∼) marking empty lines
- a line of status at the bottom of the screen.

You can usually tell by the blank line markers that you are in *vim*.

The status line can be turned off, but in this case it shows:

- the rightmost ∼n∼ characters of the filename,
- the number of lines and characters in the file,
- the location of the cursor.
- where you are in the file ("All" because its all showing).

It is not exciting, and that is good.

Modality

The original vi was invented back when "green screen" ascii terminals were the UI innovation of the day (ask your dad about ascii terminals). There were not so many shift-like keys (shift, alt, ctrl, windows, fn) and there was no such thing as a pointing device. Pretend that there was only a "ctrl" key and a "shift" key, whether it is true or not.

Programming and all other computer use)was done with your eyes on the screen and two hands on the keyboard. Vi made it possible to do so **quickly**, because vi is a bit like a video game, where any little gesture on the keyboard causes something to happen.

If you are using *vim* and pressing Whatkeys causes either cool or unfortunate things to happen, you know you are in the `command mode`, which is the default state of the editor. Commands are assigned to the ordinary everyday keys like ‘p’ and ‘y’ and ‘g’, not chords like Control-Alt-Shift-Escape.

vim has combinations and sequences to get the special power-ups like navigating between functions in separate files and reformatting entire lists in the middle of a document, code completion, abbreviations, templates and the like but that is for later.

There has to also be a way to type text into a document, but most of the keys already have special meanings! The only reasonable option was for the developers to create an “insert `mode`” which would make the ‘a’ key type an ‘a’ character, just like a typewriter (ask your dad what a typewriter is). This is called “insert `mode`”. Not much happens in “insert `mode`” except normal, old, boring typing. You only want to use insert `mode` when you must do typing, but all the cool stuff happens in the normal (control) `mode`.

You will learn many convenient ways to get into insert `mode`, but for now you should know that the way out of insert `mode`, back to the video-game-like control `mode`, is to press the ESCAPE key.

Understanding that you have basically two `modes` of operation will make your stay in *vim* less confusing, and starts you on your way to *vim* guruhood.

Know the *vim* command pattern

Most of the time you will either get an immediate result from a keystroke, or you will type a command and a movement command (often repeating the same keystroke: the “double-jump”). When you start to learn the other bits and pieces (registers, repeats, etc) then you might think *vim* is inconsistent, and this is not so. The command pattern is rather consistent, but some parts are optional.

```
register repeats operation movement
```

item	**meaning**
`register`	Register name (optional, with default cut/paste register used if not otherwise specified)
`repeats`	Repeats (optional): 13
`operation`	Operation: y (for yank)
`movement`	Movement (depending on the operation): yy (repeated to take current line, a convention used in vi)

vim commands work with the pattern shown above. There are some commands that don’t use register and some that don’t take movement, but for the most part this is the way it goes.

A register is essencially a cut-n-paste buffer. In most editors you get only one. In *vim* you have too many, but you don't have to use them, so don't worry about it untl you get to the lesson on *registers*.

A repeat is a number of times you want to do something. If you don't type in a number, the default is 1.

An operation is a keystroke that tells *vim* to do something. These are mostly normal keypresses, and most operators do not require shifts or alts or controls.

Movement is a command that takes the cursor somewhere. There are a lot of them, because there are lots of ways you need to move. don't panic, though, because you can use the arrow keys if you really have to. There is a whole section of this tutor on moving around.

Lets try an example to clarify how the pattern works. If I want to copy 13 lines into my copy/paste register, I can skip specifying a register name, type 13 for a repeat count, press 'y' for yank, and then press one more 'y' as a movement command (meaning current line). That yanks 13 lines into the default cut-n-paste register. If I press 'p' (choosing to use no register name and no repeat, recognizing that put has no movement command), then those lines are pasted back into my document just after my current line.

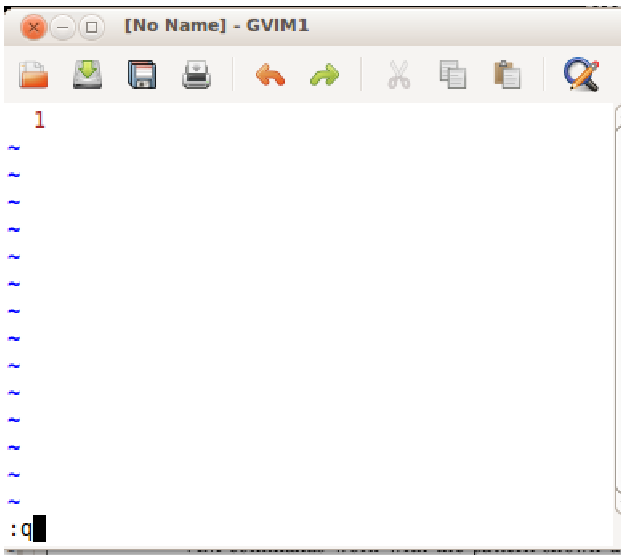
If you know this pattern, then you will know how to leverage everything else you learn about vi. Learn all the convenient ways to move, and you will know how to cut, paste, reformat, shift, and many other things you want to do.

GET OUT!

You should be able to get out of a *vim* session once you are in it. There are a few ways to do so. Try these:

What to Type	**What it does**
`:q`	Quit the current window (or editor if you're out of windows) if there are no unsaved changes.
`:q!`	Quit the current window even if there are unsaved changes.
`:qa`	Quit all windows unless there are unsaved changes.
`:qa!`	Quit all windows even if there are unsaved changes.
`:wq`	Save changes and quit the current window.
`ZZ`	Save changes and quit current window

When you type a colon, the cursor drops to the lower left corner of the screen. Later you will know why. For now, it is enough to know that it is supposed to do that, and that these :q commands will work. Notice that there is no : in front of ZZ.



how it looks when you quit

If you can't get out of *vim*, you should check to be sure the caps lock is OFF, and press the escape button. If it feels good, press it a couple of times. If it beeps, you know that you've escaped enough. Then these exit commands should work.

Mnemonics

Not all commands are mnemonic. They tried, but there are more than 26 things you might want to do in a text editor, and the distribution of letters means that not that many words start with a 'q' and happen to be meaningful in editing. However, many commands are mnemonic. There are commands for moving Forward, Back, a Word at a time, etc.

A great many are mnemonic if you know the jargon. Since "copy" and "cut" both start with "c", we have the vernacular of "yank" (for copy), "delete" (for cut), and "put" (for paste). Y, D, P. It seems a little funky but it is possible to remember these. Remember, eventually it becomes muscle memory, but the authors of VI and *vim* tried not to be arbitrary when it was totally up to them. Sometimes, there wasn't much of an option.

Invocation

Now that you know how to get out of *vim*, maybe it is time to learn how to get into *vim*. We typically start *vim* from the command line, though you may have menues or other ways.

There are a few ways you can start *vim*.

What to Type	What it does
`vim`	start with an empty window
`vim file.txt`	start with an file.txt loaded and ready to edit
`vim +23 file.txt`	start with an file.txt loaded and ready to edit at line 23.
`vimtutor`	Start in tutorial mode. This is a good idea.
`vimdiff oldfile.txt newfile.txt`	Start *vim* as a really fancy code merge tool.
`vimdiff .`	Start *vim* as a file explorer.

There is more, not shown. For now, knowing these will help you to get started. DO try out the vimtutor and the vimdiff. Some of these won't work until you set up a .vimrc, but that is explained later.

```
>vim Book.txt
```

starting vim

If you type `gvim` instead of `vim` (mvim on OS X) then you will get the gee-whiz, cool, gui version of *vim* (if it is installed). It has some extra powers. You'll typically like it better than the plain *vim*. It is like *vim* with chocolate icing. Everything we say about *vim* here is also true of GVIM, so you can use the same tutorial with either.

You don't have to edit one file at a time. You can start (g)*vim* with multiple filename arguments. When you do, there are a few options you can pass to get some fun additional effects. Of course, these are more fun after you learn how to work with split windows, so you can refer back to it later.

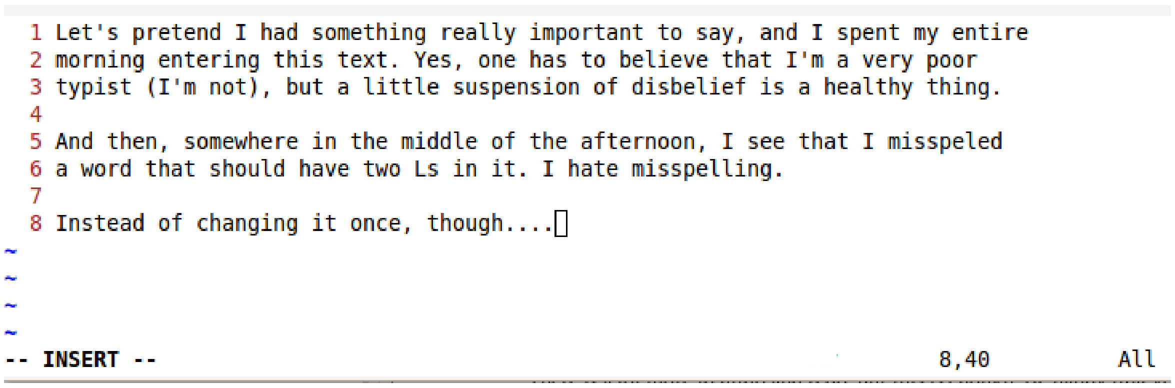
Option	What it does
`-o`	Open multiple files in horizontally tiled windows .
`-O`	Open multiple files in vertically tiled windows .
`-p`	Open multiple files in separate tabs (I hate this).

Don't panic. You have undo/redo

The command for undo is `u`. That is not too hard to remember, is it? A lot of *vim* commands are pretty mnemonic-friendly.

The redo would be the `r` key, but the `r` is used for "replace" (we'll talk later about this). We're stuck with `control-R` instead. Ah, well. You can't have everything.

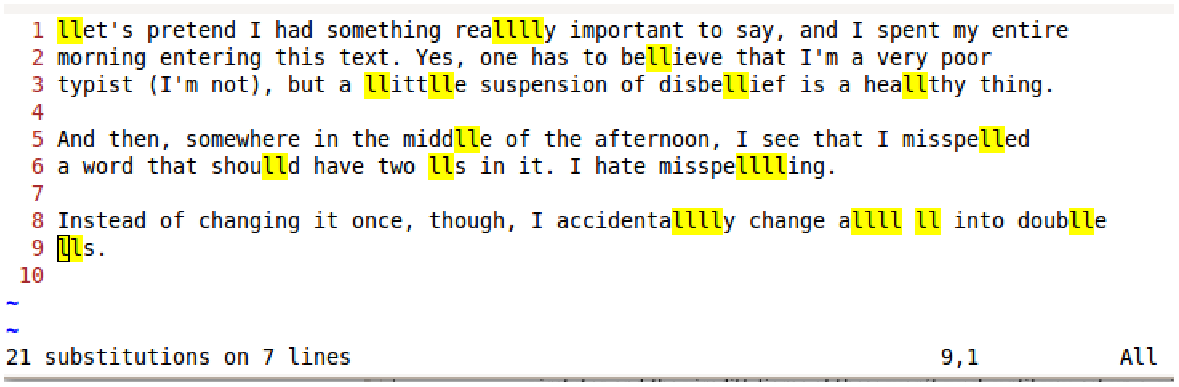
This is a good place for an example, so lets start with some precious, text that cost me a whole morning of text (well, a couple of minutes at least)



```
Let's pretend I had something really important to say, and I spent my entire
morning entering this text. Yes, one has to believe that I'm a very poor
typist (I'm not), but a little suspension of disbelief is a healthy thing.

And then, somewhere in the middle of the afternoon, I see that I misspeled
a word that should have two Ls in it. I hate misspelling.

Instead of changing it once, though....
~
~
~
~
-- INSERT --                                              8,40          All
```

before making the error

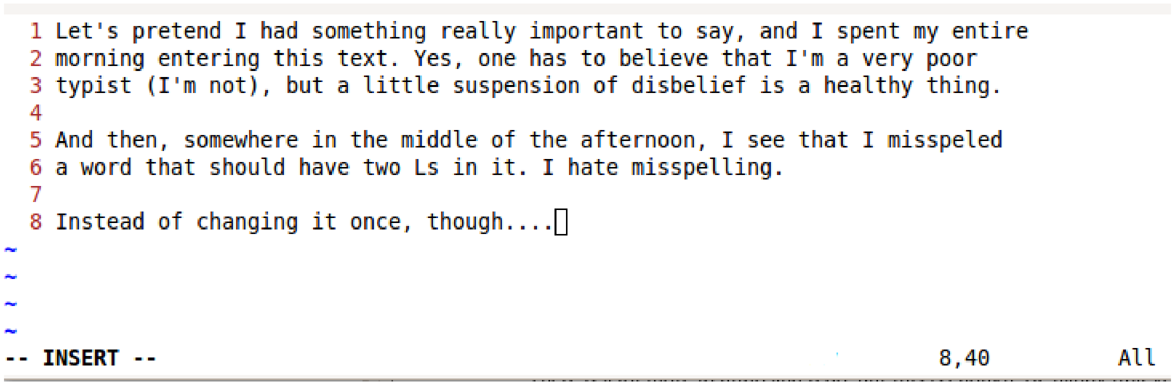
Ewww. Misspelling. Yuck, Lets change that L into a double-L.



```
llet's pretend I had something realllly important to say, and I spent my entire
morning entering this text. Yes, one has to bellieve that I'm a very poor
typist (I'm not), but a llittlle suspension of disbellief is a heallthy thing.

And then, somewhere in the middlle of the afternoon, I see that I misspelled
a word that shoulld have two lls in it. I hate misspellling.

Instead of changing it once, though, I accidentallllly change allll ll into doublle
lls.

~
~
21 substitutions on 7 lines                               9,1           All
```

after the error

Wow. That is far worse. As a pro *vim* user, I press the `u` button for undo.



```
Let's pretend I had something really important to say, and I spent my entire
morning entering this text. Yes, one has to believe that I'm a very poor
typist (I'm not), but a little suspension of disbelief is a healthy thing.

And then, somewhere in the middle of the afternoon, I see that I misspeled
a word that should have two Ls in it. I hate misspelling.

Instead of changing it once, though....
~
~
~
~
-- INSERT --                                              8,40          All
```

starting vim

There is a lot more to undo and redo, but this is enough. Be happy that you can revert changes, and un-revert them. *vim* isn't as powerless and unforgiving as you feared it might be, though you might still not like it very much. Just wait for that muscle memory to kick in.

If you get into a real mess, then exit the editor without saving.

If you are really afraid, or really cautious, then you should have version control for your text files. I recommend you start editing with junk files in a junk directory anyway, but when you are working on something important, you should not be afraid to make changes. Version control is a good security blanket and a useful backup strategy. Consider using Git or Mercurial, both of which are easy and powerful.

Move by context, not position

The poor soul who is using *vim* for the first time will be found pressing up and down arrows and executing key repeats, moving horribly inefficiently through any body of code. He will be scrolling or paging (btw: `\^f` moves forward one page, `\^b` moves backward one page) and searching with his poor eyeballs through piles of code. This poor soul is slow and clueless, and probably considers *vim* to be a really bad version of windows notepad instead of seeing it as the powerful tool it is.

By the way, the arrow keys do not always work for *vim*, but do not blame *vim*. It is actually an issue with the way your terminal is set up. *vim* can't tell that your arrow keys are arrow keys. If you have the problem, you have more research to do.

To use *vim* well, it is essential that you learn how to move well.

Do not search and scroll. Do not use your eyes to find text. They have computers for that now. Here are a handful of the most important movement commands. The best way to move is by searching:

What to Type	What it does
`/`	search forward: will prompt for a pattern
`?`	search backward: will prompt for a pattern
`n`	repeat last search (like dot for searches!)
`N`	repeat last search but in the opposite direction.
`tx`	Move "to" letter 'x' (any letter will do), stopping just before the 'x'. Handy for change/delete commands.
`fx`	"Find" letter 'x' (any letter will do), stopping **on** the letter 'x'. Also handy for change/delete commands

If you're not searching, at least consider jumping

What to Type	What it does
`gg`	Move to beginning of file
`G`	Move to end of file
`0`	Jump to the very start of the current line.
`w`	Move forward to the beginning of the next word.
`W`	Move forward to the beginning of the next space-terminated word (ignore punctuation).
`b`	Move backward to the beginning of the current word, or backward one word if already at start.
`B`	Move backward to the beginning of the current space-terminated word, ignoring punctuation.
`e`	Move to end of word, or to next word if already at end.
`E`	Move to end of space-terminated word, ignoring punctuation

The following commands are handy, and are even sensible and memorable if you know regex:

What to Type	What it does
`\^`	Jump to start of text on the current line. Far superior to leaning on left-arrow or h key.
`$`	Jump to end of the current line. Far superior to leaning on right-arrow or k key.

Here is some fancy movement

What to Type	What it does
`%`	move to matching brace, paren, etc
`}`	Move to end of paragraph (first empty line).
`{`	Move to start of paragraph.
`(`	Move to start of sentence (separator is both period and space).
`)`	Move to start of next sentence (separator is both period and space).
`''`	Move to location of your last edit in the current file.
`]]`	Move to next function (in c/java/c++/python)
`[[`	Move to previous function/class (in c/java/c++/python)

Finally, if you can't move by searching, jumping, etc, you can still move with the keyboard, so put your mouse down.

What to Type	What it does
`h`	move cursor to the left
`l`	move cursor to the right
`k`	move cursor up one line
`j`	move cursor down one line
`\^f`	move forward one page
`\^b`	move backward one page

You want to use the option `hls` (for "highlight search") in your vimrc. You will learn about that soon enough. In the short term you can type "`:set hls`" and press enter.

Help is on its way.

There is an online help mechanism in *vim*. You should know how to use it.

Type `:help` and you will get a split window with help text in it. You can move around with the arrow keys, or with any of the *vim* movement commands you will learn.

You can always enter funky keys by pressing ˆv first, and then the keystroke. This is most useful in help. You can type `:help \^v\^t` to get help for the keystroke ˆt. By convention you can usually get what you want by typing `:help CTRL-T` also. Do not underestimate how handy this is.

Most distributions of *vim* will install a program called `vimtutor`. This program will teach you to use *vim*. It will do so by using *vim*. It is a handy piece of work (props to the author!).

Help has links. If you see one you like, you can move the cursor to the link (lets not just beat on the arrow keys, here!) and press ˆ]. Yeah, it is an odd and arbitrary-looking command. That will not only navigate to the link, but also push it on a stack. If you want to go back, you can press ˆt (yes, also pretty arbitrary) to pop the current link off the stack and return to the previous location in the help. The commands `\^]` and `\^t` aren't very memorable, but we'll use them for code navigation later, so learning them is not a total waste of mental energy.

Shifted letters and DEATH BY CAPS!

For a number of commands, shift will either reverse the direction of a command (so N is the opposite of n, see next bullet) or will modify how the command works. When moving forward by one word at a time (pressing `w`), one may press `W` to move forward by one word but with W the editor will consider punctuation to be part of the word. The same is true when moving backward with `b` or `B`.

Because a shifted letter may mean something very different from the same letter unshifted, you must be very careful not to turn on the capslock! Sometimes a poor unwary soul will accidentally hit the capslock. When he intends to move left with 'j', he instead joins the current line with the next. Many other unwanted edits can take place as his fingers make a quick strafing run for some complex edit. It is ugly.

If you encounter DEATH BY CAPS, you should turn off the capslock, and then try pressing 'u' repeatedly to get rid of unwanted edits. If you feel that it is a lost cause, press ":e!" followed by pressing the enter key. That will reload the file from disk, abandoning all changes. It is a troublesome thing that will eventually happen to you. Some people turn off their capslock key entirely for this reason.

Quoting Your Regex Metacharacters

If you do not know what a regex is, skip this section. For those who understand what a regex is, and who realize that the "/" command takes a regex rather than just normal text, this will be important. For the rest of you, it will seem totally out of place and should be skipped for now.

You should know how to use regular expressions, because a few tricks in regex will make your whole Unix/Linux/Mac experience a little better. It is too large a topic to expose fully here, but you might try looking at on of the good references or [3]tutorials[4] elsewhere on the web.

The main thing to remember is that *vim* will side with convenience when it comes to regex. Since you search a lot, *vim* will assume that `/+` means that you want to search for the nearest + character. As a result, all the metacharacters have to be quoted with the backslash ("") character. It is sometimes a pain, but if you really want to find a plus sign followed by a left-parenthesis, it is very easy.

Insert, Overwrite, Change

In *vim* you have a variety of ways to start entering text, as mentioned above in the section on Modality.

You are normally in `command mode`. When you type certain keys, you are placed in insert mode or overtype mode. In insert mode, the text you type goes before the cursor position, and everything after the cursor is pushed to the right or to the next line.

In ' overtype mode' your keystrokes are input, just as they are in insert mode, but instead of inserting the keystrokes *vim* will replace the next character in the document with the character you type. You get to overtype mode by pressing an overtype key command while in command mode.

In `ex mode` you are typing a string of commands to run into a little window at the bottom of the screen. We'll talk about this later on, because it is powerful stuff. It is also a little cryptic, so we will wait. You get into ex mode by typing ":" in command mode.

You always return to command mode from overtype, insert, or command mode by pressing escape. That is one handy key.

[3]http://www.geocities.com/volontir/%3E
[4]http://larc.ee.nthu.edu.tw/~cthuang/vim/files/vim-regex/vim-regex.htm

What to Type	What it does
`i`	insert before the current cursor position
`I`	insert at the beginning of the current line. Far better than pressing ˆ and then i.
`a`	insert after the current cursor position
`A`	insert/append at the end of the current line. Far better than pressing $ and then i.
`r`	retype just the character under the cursor
`R`	Enter overtype (replace) mode, where you destructively retype everything until you press ESC.
`s`	(substitute) delete the character (letter, number, punctuation, space, etc) under the cursor, and enter insert mode
`c`	the 'change' (retype) command. *Follow with a movement command.* `cw` is a favorite, as is `cc`
`C`	Like 'c', but for the entire line.
`o`	insert in a new line below the current line
`O`	insert in a new line above the current line
`:`	Enter command mode (for the advanced student)
`!`	Enter shell filter mode (for the very advanced student)

Consider the value of the `c` command. If you use it with the ‘`t or  f commands, it becomes very powerful. If you were at the C at the beginning of the previous sentence, you could type `ct. `and retype the whole first sentence, preserving the period. The same is true with other commands, such as the `d‘ for delete. The movement commands add a lot of power to the change command, and that is one reason why it is important to learn to move well.

NEVER PARK IN INSERT MODE.

vim is set up to do more navigating and editing than typing. It rewards you for working in the same way, mostly in control mode with spurts of time in insert mode.

If you try to use *vim* as a weak form of notepad, modality and navigation will ensure that you are never really efficient. If you want to sail, you have to get in the boat, and if you want to get good at *vim*, you need to get good in command mode.

So, if you are stopping to think, hit `<esc>`. If you aren't in the middle of text typing, you should be in command mode. If you are wanting to move up or down a line, or to some other place, hit

Gain Efficiency

The surest way to tell a *vim* noob is by listening to them type. You will hear the constant tap-tap-tap. They tap or hold down the ‘down’ key so that the page scrolls, stopping to read from time to time. They tap-tap-tap the space or right-arrow to move past the text, then tap-tap-tap the backspace, and then type in the new text.

Noobs live in the insert mode, as if vim were merely some primitive version of Notepad. That way is ever-so-slightly better than nothing, I guess.

If the noob is going to make the same change again, he repeats the process in each place they want to change.

Already you can insert at the beginning or end of any line, search and replace text with * and `#`. You search by content, not by position. You will not get a headache in 5 minutes of programming.

You are no longer the noob. But neither are you the master.

In the hands of a master, the code dances and flashes and changes at a mere flinch of the hands. Where does this magic come from?

Vim has had many years to evolve very effective patterns, many exposed here. It is time for you to move from competent to amazing.

The Double-Jump

This is a small trick that makes a big difference.

By convention (“usually”) pressing a command twice will tell it to operate on the current line. If you want to yank (copy) the current line, press `yy`. If you want to delete the current line, press `dd`. This is a pretty consistent convention, down to the special case of “save and exit” being `ZZ`. Doing operations on the entire current line is very common, and it made sense to make it convenient.

What to type	What it means
cc	change (retype) the current line
yy	copy (yank) the current line
dd	delete the current line
ZZ	save and exit the current file
gg	go to the top of the current file

Happiness is a good .vimrc

When *vim* starts up, it reads your personal settings before it does anything interesting. You should create and edit a file named `.vimrc` in your home directory.

There is very fine magic in *vim*. However, it often comes without the magic turned on. Command line completion, color syntax highlighting, the file explorer, and many other features are “missing” unless you turn them on in `~/.vimrc`.

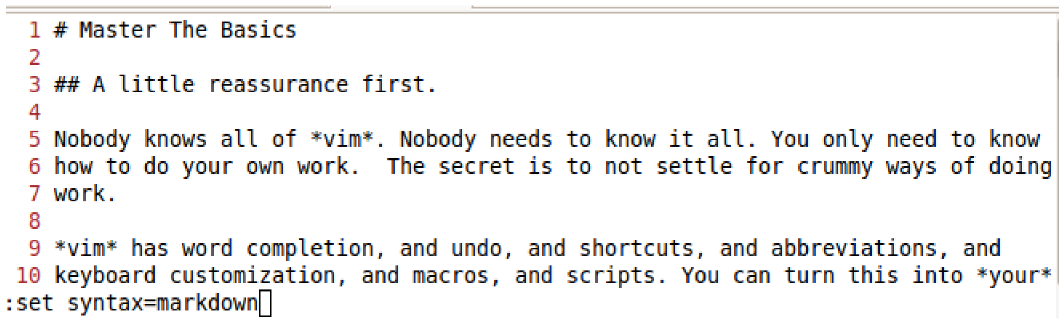
Start by creating a one-line `.vimrc` that does nothing but turn syntax on. Then exit and restart vim.

```
vim .vimrc
```

On restart, *vim* will automatically recognize the syntax of the .vimrc and highlight it accordingly. This helps you to recognize and correct syntax errors. *Vim* automagically recognizes the formats of *many* files.

Sometimes you may have markdown in a text file. In those cases, *vim* will guess wrongly. It will correctly see that your file is text, and use no highlight marking at all. This can happen for other reasons in other files.

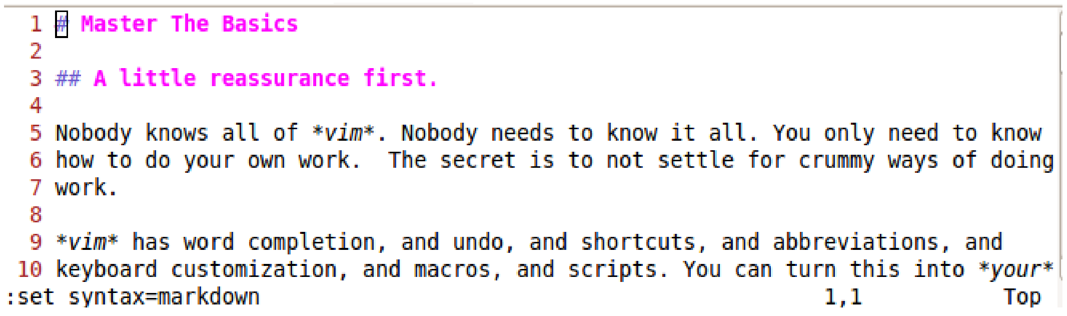
In such a case, you can enter a command:



```
# Master The Basics

## A little reassurance first.

Nobody knows all of *vim*. Nobody needs to know it all. You only need to know
how to do your own work.  The secret is to not settle for crummy ways of doing
work.

*vim* has word completion, and undo, and shortcuts, and abbreviations, and
keyboard customization, and macros, and scripts. You can turn this into *your*
:set syntax=markdown
```

image of a user setting the syntax

Once the syntax selection is corrected, *vim* will help with syntax highlighting as best it can.



```
# Master The Basics

## A little reassurance first.

Nobody knows all of *vim*. Nobody needs to know it all. You only need to know
how to do your own work.  The secret is to not settle for crummy ways of doing
work.

*vim* has word completion, and undo, and shortcuts, and abbreviations, and
keyboard customization, and macros, and scripts. You can turn this into *your*
:set syntax=markdown                                            1,1         Top
```

image with syntax coloring and highlighting

There is a very nice guide to the various settings in *vim*, and even an interactive display so that you can turn them on and off.

Guides to Vim Configuration
`:options`
`:browse options`
`:browse set`

In this window, you can browse through all the available options, and can even set their current settings. Short help messages are associated with each, and you can hit the enter button on any short help to see the longer help text. If you press the enter key on an option, it will toggle that option or set a new value.

Let's start adding lines to the .vimrc one-at-a-time. Notice what changes, how it helps you edit.

What to Type	**What It Means**
`syntax enable`	turn on all the magic, including Explorer and syntax highlighting
`set showmode`	Show me when I'm in insert/overtype mode
`set showcmd`	When a command is in progress, show it in the status bar
`set wildmenu`	magic for completion at the : command line.
`set ruler`	turn on the "ruler" (status info) at the bottom of the screen.
`runtime ftplugin/man.vim`	Turn on man pages (type :Man)
`set autoindent`	indent in a smart way, instead of returning to the left margin all the time
`set expandtab`	expand tabs to spaces
`set nowrap`	Don't wrap text (makes windows ugly)
`set hlsearch`	Highlight all matches in text when you search
`set showmatch`	Show matches for braces, parens, etc.
`set ignorecase`	do case-insensitive searching
`set smartcase`	When a search phrase has uppercase, don't be case insensitive
`set path=.,..,/usr/include/**,/usr/share/**`	Tell the editor where to search for files
`set spelllang=en_us`	when I want spell-checking, I want it to be english

If you type these incorrectly, then you will see helpful error messages when you start *vim* next time. To help eliminate fear and nervousness, change some options to be wrong and then exit and reload vim.

If that's too tedious, you can use the command: :source ∼/.vimrc

The source command also can be abbreviated as “so”: :so ~/.vimrc

This will cause the .vimrc file to be re-read and you can see your syntax errors.

If it feels safer, copy your .vimrc to a file called “temp.vim” and use “:so temp.vim” to test it until you know it is right. When you like it, move/copy/append it to your .vimrc

Getting rid of things.

Up until now, you could go into insert mode and backspace over text, but that has the tell-tale “tap tap tap” pattern. Let’s learn a more efficient way.

You can get rid of the character under the cursor by pressing `x`. For one character, it’s fine. If you want to delete 10 characters, you can save effort by typing `10x` (remember the “command pattern?”).

It can be pretty handy, but you could very quickly get tired of counting how many times you want to press x. I know I would.

The more flexible `delete` command is very simple. It is the letter ‘d’ for “delete”. It is one of the lucky mnemonic commands. It also supports the standard command pattern presented early in the tutorial.

Let’s explore what that means.

You can use *d* with a movement command. The delete command also supports the double-jump (*dd*).

You can delete the current line by typing “dd”, or you can delete the current line and the one under it by typing d followed by the ‘j’ or ‘down arrow’.

Here are more examples (not an exhaustive list):

Keystrokes	Behavior
d}	Delete to end of paragraph
dG	Delete to end of file
dtJ	Delete up to capital J

All commands that take a movement command will work this way (including ‘c’). Every movement command you learn increases your power to copy, delete, and retype. This added power is why it is essential that you learn to move well in *vim*.

Delete will also take a *repeat* count, so you can type `23dd` to delete 23 lines starting with the current line. This can be handy.

But what about the ‘register’ part of the command? We will talk about those later.

Use the Dot.

Edits actions in *vim* are recorded. Say that you just deleted a line (by typing `dd`). The editor knows you deleted a line. You can repeat the edit (that is, delete another line) by pressing the period key ("."). You can even apply the standard pattern and give a register, repeat, and then a dot (the dot knows the command and movement). This is particularly handy if the command you last used was `cw`. It will repeat the replace operation on the text under the cursor.

Because the dot command repeats the last edit you did, it is one of the most powerful keys on the keyboard. You should learn to rely on it. It is one of the most wonderful things *vim* gives you.

Use the Star

The star is a great command, especially if you have the option `hlsearch` turned on in your .vimrc file. It will move to the next use of the word under the cursor. In doing so, it will highlight all uses of the word under the cursor.

What to Type	What it does
*	Move to next instance of word under cursor.
#	Move to previous instance of word under cursor.

Keep text in front of your face

There are commands for moving the location where the current line appears on the display. The *vim* folks were running out of letters I think, so they attached these commands to the z key.

What to Type	What it does
zt	move current line to top of page
zz	move current line to middle of page
zb	move current line to bottom of page

You also can do much to keep reference code in front of your face if you use split windows. Lets assume your are in `code.cpp`, and want to look at `code.h` for a while.

What to Type	What it does
:split code.h	splits window horizontally and loads code.h in a new window
:vsplit code.h	splits window vertically and loads code.h in a new window

Once you have split windows, you'll want to know how to move between them. Here is a small set of commands (all bound up in ^w sequences) that will help you move about. You can always close any window (even a split one) with the `:q` or `ZZ` tricks (from “GET OUT”, far above).

What to Type	**What it does**
$^{\text{W j or}}$W leftarrow	Move to next window to the left
$^{\text{W l or}}$W rightarrow	Move to next window to the right
$^{\text{W k or}}$W uparrow	Move to window above current window
$^{\text{W j or}}$W downarrow	Move to window below current window
ˆW c	Close current window
ˆW o	Close all windows except the current window

Check out `:help CTRL-W` for more information about window control and movement.

Take cut-n-paste to the next level

Registers

In most editors you get a single cut-n-paste buffer. When you use the cut or copy command, you lose whatever is in the buffer. As a result you end up zipping back and forth in a file, cutting from one place, and pasting in another. If you are lucky you can split the window and go back and forth between tiles, but it's a lot of manual labor and an exercise in hand-eye coordination as you seek, cursor, mark, cut, seek, cursor, paste your way to authoring nirvana.

It probably took a couple of minutes to get sick of that.

In *vim*, they have a different answer. Sadly they have different terminology, too. Instead of editing buffers, we have "registers". Same concept, different term (the word "buffer" means something else in *vim*).

A *vim* register is a like the copy-and-paste buffer you have used in lesser editing tools. When you delete, the deleted text is saved in a default buffer (like a "cut" command). You can paste it back into the document by pressing the `p` (mnemonic: put or paste) key.

The delete key can use a named register. A register name is specified by typing a double-qoute character, followed by the name of a register. The register name is a lower-case or upper-case letter (case is significant).

You may rearrange multiple bits of text by cutting them into different registers and pasting them into different places (or files) by using a register-specific paste.

Pasting also takes a register specification, which is always a double-quote followed by a letter, followed by the 'p' for paste.

Here is a non-exhaustive set of examples:

What to Type	What it does
x	cut the character under the cursor to the default buffer
p	paste whatever is in the default buffer
xp	cut the current character and paste it back to the right (transpose)
yy	Cut the current line into the default register
"ayy	Cut the current line into register *a*
"ap	paste from register *a* into the current document
22"ap	paste from register *q* 22 times

The registers are (from the *vim* documents, available via `:help registers`):

Register Name	What it does
a-z	yanked text replaces current content of register
A-Z	yanked text appends to the current content of the register
”	The unnamed or default register
+	The system default register (the normal cut/paste one)
*	Select/drop registers
_	The black hole – essentially /dev/null, used to avoid wiping out register “ (the unnamed register)

There are also a few other special-purpose registers which I leave for your exploration in the help system, such as the small delete register and the numbered ones. You can use *vim* for years without knowing these.

vim expects you to prefix the register name with a double-quote character. This is how it knows the difference between the command y and register y. Some day you will for get, and be surprised. Try it in a junk file to see how the error feels and looks.

So, y is the yank command, and “y is the y register. If you type “y *vim* will wait for you to complete the standard pattern before taking action.

Examples of increasing power/complexity:

What to Type	What it does
dd	yank the current line into the default, unnamed register (“” or quote-quote)
“add	delete the current line into register a
“y$	Yank from the current character to the end of the line into register y
“byy	Yank the current line into register b
“c24dd	Literally *Into register c, 24 times delete the current line*. That’s complex to read, maybe it’s easier to just say *delete the next 24 lines into the c register*

I’m sure that `"c24dd` seems a little crazy, but think how you would do the same work if you were using notepad or the like. This is 6 keystrokes, and only one of them shifted, and you never had to leave the home row to grab a mouse.

It would be an extremely efficient way to cut 24 lines into a named register **if** you happened to know that you had 24 lines. If you didn’t know that, the work of counting the lines would more than destroy the efficiency. That makes this a pretty academic example, and opens the door to visual marking of text for copy/cut, etc.

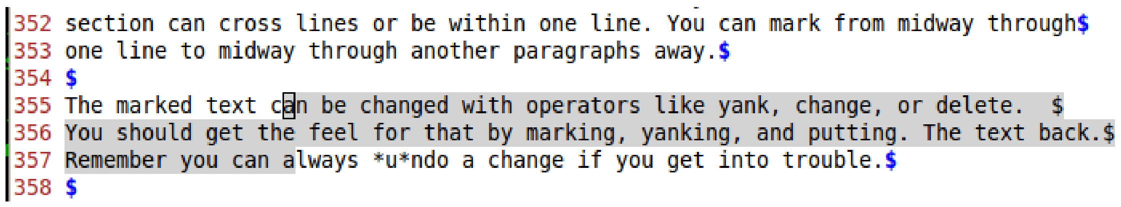
You may forget what you have in your registers. If you type `:registers` (from command mode, of course) *vim* will present you with a list of your registers and their content. Handy tip courtesy of Chris Freeman.

Marking

vim has non-visual marking and it has visual marking. Chances are, you are interested in visual marking for cut-n-paste (yank-n-put) purposes, so let's look at that.

What to Type	What it does
v	mark character-wise
V	mark line-wise
ˆv	column-wise marking
gv	Remark the area last marked.

The command for *v*isual marking is `v` (another mnemonic!). You can press `v`, and then cursor or search to the end of the text you want to mark. The marked section can cross lines or be within one line. You can mark from midway through one line to midway through another paragraphs away.



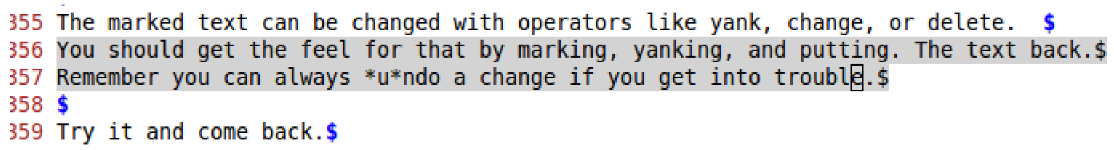
```
section can cross lines or be within one line. You can mark from midway through$
one line to midway through another paragraphs away.$
$
The marked text can be changed with operators like yank, change, or delete.  $
You should get the feel for that by marking, yanking, and putting. The text back.$
Remember you can always *u*ndo a change if you get into trouble.$
$
```

Marking in normal mode

The marked text can be changed with operators like yank, change, or delete.
You should get the feel for that by marking, yanking, and putting. The text back. Remember you can always *u*ndo a change if you get into trouble.

Try it and come back.

Sometimes you want to mark entire lines at a time. For this, *vim* uses the shifted (uppercase) `V`. It works just like the lower-case V but always selects a whole line at a time. Of course a second press of `V` will cancel this mode.

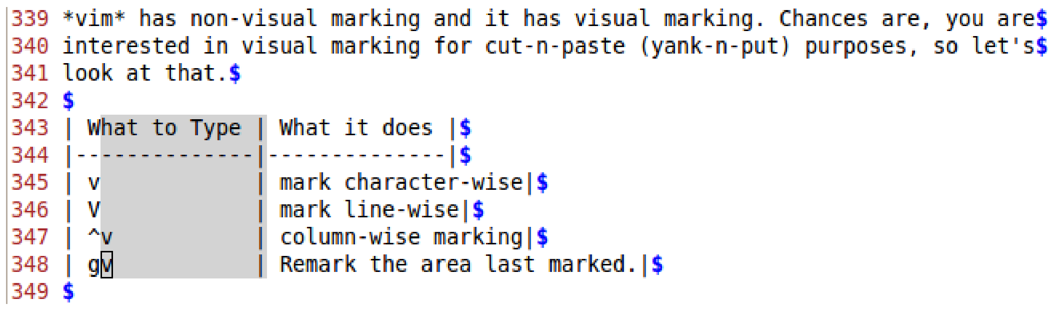


```
The marked text can be changed with operators like yank, change, or delete.  $
You should get the feel for that by marking, yanking, and putting. The text back.$
Remember you can always *u*ndo a change if you get into trouble.$
$
Try it and come back.$
```

Marking in lines mode

Other times, you might want to mark a rectangle instead of whole lines or contiguous characters.

For rectangle (blockwise) marking, *vim* uses the control character 'ˆV'. Yanking and pasting rectangular regions is a cool feature.



```
*vim* has non-visual marking and it has visual marking. Chances are, you are$
interested in visual marking for cut-n-paste (yank-n-put) purposes, so let's$
look at that.$
$
| What to Type | What it does |$
|--------------|--------------|$
| v            | mark character-wise|$
| V            | mark line-wise|$
| ^v           | column-wise marking|$
| gv           | Remark the area last marked.|$
$
```

Marking in rectangle mode

> Notice that ˆV has an entirely different behavior in insert mode. Do not let that confuse you. Just be sure you're in command mode when you start marking.

A cool feature is that you can start marking with `v`, then press `V` to switch to line mode, or press `^V to switch to rectangle selection.`

Once you leave the visual marking mode, the area is no longer marked. the *vim* help tells us that we can go back into visual mode with the same marking mode and marked area by typing `gv`. I have been playing with it. It's handy.

I have been marking the entire document (`ggVG`) and then yanking it to the machine's cut-n-paste buffer (`"+y`) and switching to my blog editor. In the blog editor (not vi) I do the standard ctrl-a ctrl-ˆv to paste my document in. Now I can save a few keystrokes by using `gv` rather than `ggVG` before pasting (after the first time).

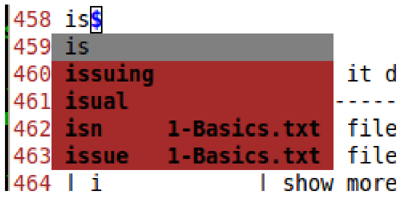
You can do much more than simple yank and put. You can use the `r` command and another letter like "X" , and change every character in the marked area to an "X". You can use the marked are for ex commands (which we have not talked about). There is rather a lot of power here, but we'll end the marking lesson here for now.

Completion

Feel free to use long names and big words, because *vim* has completion. It's not *intellisense*, mind you, but it will finish your words for you. Type enough of a word to be unique, and (without leaving insert mode) press ˆn. If the word you're looking for is in any of the loaded files (or buffers) then *vim* will present its best guess. If it is not the one you want, press ˆn again until either you find your word, or you run out of choices. You can also use ˆp to go back to a previous selection.

What to Type	What it does
ˆn	In insert mode, complete a word (forward to through choice list)
ˆp	In insert mode, complete a word (backward through choice list)

In newer version of *gvim* (graphical version) a selection box will pop up, and you will pick your word by either typing a little more so it really is unique or else by using arrow keys.

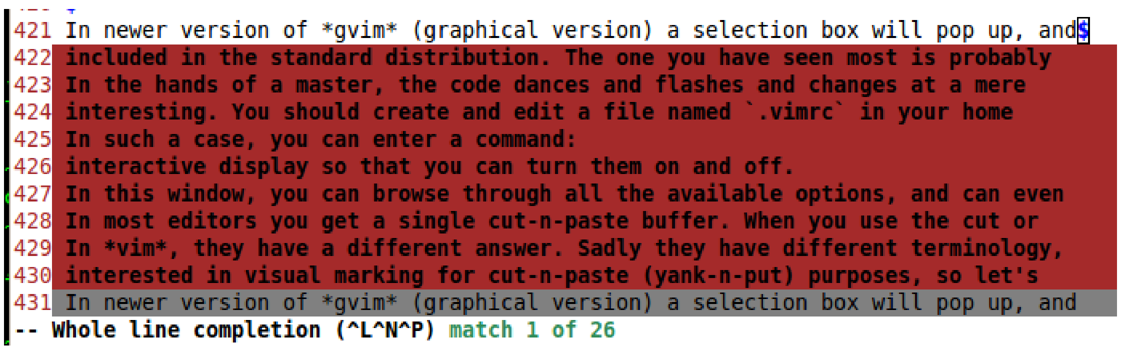


```
is
is
issuing                it d
isual                  -----
isn      1-Basics.txt  file
issue    1-Basics.txt  file
| i                 |  show more
```

the completion list

> Notice in the picture that ‘isn’ and ‘issue’ come from a different file, named 1-Basics.txt. Often noobs will edit one file, exit, then edit another. If you have related files loaded, completion is much more powerful. Noobs don’t know that because they’ve not learned about completion yet.

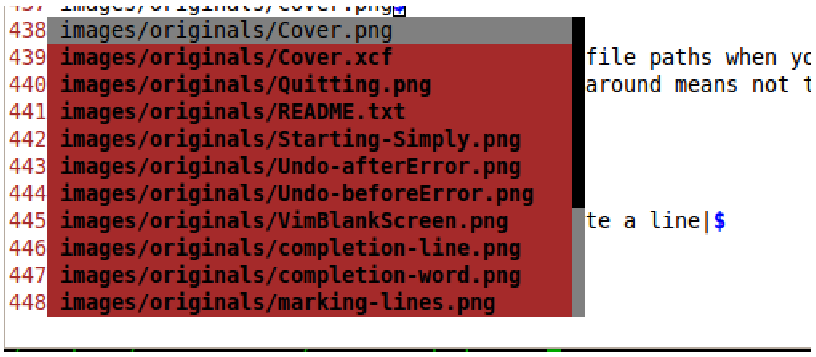
There is a more comprehensive “whole line completion” mechanism availabe to you also. You can press ^{x}l to enter a special completion mode. You cycle through choices with $^{n \text{ for next and}}$p for previous, or with arrows (if your *vim* supports them). Again, if you are using *gvim* you will get a popup window with choices. There are times this is more useful than doing cut-and-paste the old-fashioned way.



```
In newer version of *gvim* (graphical version) a selection box will pop up, and
included in the standard distribution. The one you have seen most is probably
In the hands of a master, the code dances and flashes and changes at a mere
interesting. You should create and edit a file named `.vimrc` in your home
In such a case, you can enter a command:
interactive display so that you can turn them on and off.
In this window, you can browse through all the available options, and can even
In most editors you get a single cut-n-paste buffer. When you use the cut or
In *vim*, they have a different answer. Sadly they have different terminology,
interested in visual marking for cut-n-paste (yank-n-put) purposes, so let's
In newer version of *gvim* (graphical version) a selection box will pop up, and
-- Whole line completion (^L^N^P) match 1 of 26
```

the line completion list

Less well known, there is a filename completion mechanism, accessed with ˆxˆf.



```
images/originals/Cover.png
images/originals/Cover.xcf          file paths when yc
images/originals/Quitting.png       around means not t
images/originals/README.txt
images/originals/Starting-Simply.png
images/originals/Undo-afterError.png
images/originals/Undo-beforeError.png
images/originals/VimBlankScreen.png te a line|$
images/originals/completion-line.png
images/originals/completion-word.png
images/originals/marking-lines.png
```

file name completion

Do not cut-and-paste individual names or file paths when you can have VIM type those for you. It's handy, and having it around means not typing filenames during a normal day.

What to Type	What it does
ˆxˆl	In insert mode, complete a line
ˆn	Get next choice
ˆp	Get previous choice

When you have your selection, just keep typing. Any key other than a selection key (up/down/ˆn/ˆp) will be accepted as new text as is normal in insert mode. This is a little counter-intuitive because you are accustomed to hitting enter or tab to accept the entry.

There are a number of other special commands which are only available in insert mode.

Since you have word-completion and line-completion, you have no excuse for writing short and cryptic variable names. Very long, meaningful names are quite feasible and not tedious at all.

The Explorer

You can edit directories. Give it a shot. There is help available, and you can get more information on the screen by pressing `i`. This is a kind of "poor man's midnight commander", or maybe a reasonable substitute for the windows explorer. It's quite handy, and highly recommended. This only works if "syntax enable" is in your .gvimrc file.

What to Type	What it does
o	Open file in a (horizontal) split window
v	Open file in a (vertical) split window
i	show more info
s	sort by column under the cursor
r	sort in reverse order
D	delete file
d	make new directory
enter	Open file in current window.

Indenting and unindenting

Forget using the tab key.

Too many tools use 8-character tabs, which is the standard. if you use tabs, even if you change the tabstop parameter in your code, a lot of programs will display or print your code incorrectly. Tabbing is dead, shifting is king.

So I recommend you set your tabstops to 8 in your .vimrc (`set tabstop=8`) and set your shiftwidth to the desired level (`set shiftwidth=4`). No, rather than recommend, I demand you go and add those two commands to your .vimrc right now. I'll wait. Really… go do it..

Command	**What it does**
set tabstop=8	Use industry standard 8-char tabs
set shiftwidth=4	Use standard 4-char indentation
set shiftround	Indent/Dedent to nearest 4-char boundary
set autoindent	Automatically indent when adding a new line

You need to also have autoindent turned on, so you do not have to manually space or indent every line. Autoindent is so handy, I included it as a necessary feature in the .vimrc section. If you followed the tutorial, you will have it turned on already. Not having it on is stupid. You really want it.

In CONTROL mode:

What to Type	**What it does**
<	left-shift (requires a movement cmd, works on whole lines)
>	right-shift (requires a movement cmd, works on whole lines)

If you want to move a paragraph to the left, then `<}` is your command. For shifting three lines right, it would be `3>>`. The shift commands follow the standard *vim* command pattern (hence the term "standard"). They do not use a buffer.

In INSERT or OVERTYPE mode:

What to Type	**What it does**
ˆT	Indent
ˆD	Dedent/unindent

Spelling

vim can also check your spelling. You can enter the command `:set spell` to turn on spelling checker. You can also set the dictionary and other options, but `:help spell` will tell you all about it.

I do not recommend turning this on normally. A lot of the things you will edit will contain stuff other than the dictionary's list of English words, and that can get to be annoying. I prefer to turn it on and off with `:set spell` and `:set nospell`.

The earnest student can learn to turn this on and off via special scripts that are run whenever a file is loaded. The less interested can skip it.

Little hints

There are some handy commands for showing you information in the status line, or in a scrolling display. When you need a reminder, but do not need to navigate to some part of source, it can be handy to use these.

What to Type	What it does
[i	show first line containing word under the cursor
[I	show every line containing word under the cursor
:g/pattern/	show every line matching the regular expression pattern

Shell Filtering

If you were working at the command line, you would know how to use sort, and filter with grep, maybe how to do various tasks with perl or awk. Those programs are all filters. They read the standard input and they write to standard output.

When you are *vim*, however, you may want to do the same things. It is sure a pain to save the part of a file you want to sort, escape to the command line, sort the piece of the file to a new file, and then load the sorted file fragment into the space in the editor where that piece of unsorted text used to be.

What you need to know is that all that work is unnecessary. If you wanted to sort a paragraph, and your cursor were at the start of the paragraph, all you have to type is `!}sort` and the magic is done.

vim is written to use filters directly. Not only is this handy for using all those great Linux/Unix filters, but also because you can write your own. Any filter-type program you write is now part of you editor as well as your command-line environment. That is a major bit of editing leverage. It is exciting stuff if you are a command-line guru already.

!! *command*	pass current line only through filter
!} *command*	pass area from current line through end of paragraph through filter
!G *command*	pass area from current line through end of file through filter
:%! *command*	pass the entire file through filter

Code Reformatting

You can reformat code or text a number of different ways. One is using shell filtering:

%!astyle	Restyle the entire file with astyle (a nice reformatting program).
%!indent	Restyle the entire file with indent (a nice, older program).

Another is using the `gq` command, which re-does the line wrapping, and which has intelligence for wrapping comments correctly.

gqq	Re-wrap the current line (a double-jump!)
gqj	Re-wrap the current line and the line following
gq}	Re-wrap lines from the current line to the end of the paragraph.

You can also retab a file. Retabbing converts tab stops to spaces, and ensures indentation is correct for each. It is done by setting your `tabstop` variable to the correct indent level, then setting `expandtab`, and finally by issuing the `:retab` command. It would be far too much work if I didn't have expandtab and tabstop set normally. Typically, I set tabstop and retab, and then save. That's a sequence I can map to a keyboard command, or can save as a macro.

You can also have the editor wrap your text as you type, and preserve your indentation. This is all done via the `linebreak`, `textwidth`, and `autoindent` settings, which you can easily explore with the `help` facility.

QuickFix mode is your friend

vim can run your makefile and take you to each variable in turn. If you have unit tests set up to run as part of the build, and the unit test framework produces messages in a compatible format, you will be guided through the failed tests just as if they were compile errors. Likewise, any style-checking tool you use may be treated likewise if it has a compatible format.

If you are doing Test-Driven Development, this is a critical feature. with quickfix mode, you can find the rhythm that you're looking for. You can even assign the :mak command to a keystroke (see :help map) so that you do not have to type :make. *vim* is a kind of agile editor in that regard. In a separate paper, I'll detail my *vim* settings for TDD.

Basic quickfix commands

:make	Run the makefile specified by the `makefile` variable
:cw	Show the compile error window if there are compile errors.
:cn	Go to the next compile error.
:cp	Go to the previous compile error.

As always, look at `:help quickfix` to learn more about this valuable mode of work.

Manual page access

In the .vimrc section I recommended that you turn on the Man feature. since you followed those instructions, you can now access man pages from *vim*

`:Man 5 crontab` shows you the crontab man page in a split window. Your cursor will be in the help window where you can navigate as you would with tags, using $^{\text{] to go to a tag, and}}$t to return. When you are done, type :q or ZZ to quit the window.

If you are looking for a man page for something in your file, you do not have to type the colon and the word man. You can type the *leader* character (by default “”) and the capital K and *vim* will find the man page and display it in a split window. By the way, if you change the leader character, you will of course have to adjust these instructions. This is very handy when you are working with scripts or the Linux/Unix C API.

:Man *subject*	Get manpage for *subject*
K	Get manpage for word currently under the cursor

This feature is more valuable if you ensure that you install all of the man pages for the programming tools and libraries you use. Or at least that you urge your systems admin to do it for you. If you work in perl, and you do not have all the perl man pages, you will lose out on this fine feature of *vim*.

Ctags lets you navigate like a pro.

I heartily recommend exuberant ctags as the tag program for almost any language. It will quickly span your code and create a ‘tags’ file, which tells *vim* all it needs to know to find a symbol in your source. The tag file gives a file and also a regular expression for finding the line you need. It does a very fine job.

!ctags -R * Run ctags (better to do this in your makefile)

ˆ]	Jump to the definition of the term (class/method/var) under the cursor
ˆt	Pop the browsing stack, return to previous location

Bookmarks.

vim allows you to set a bookmark on a line, and jump from one bookmark to another.

mx	Put bookmark ‘x’ at the current line.
‘x	Jump to mark ‘x’.

You can use any letter for a bookmark, however there is a difference between a lower-case letter and an upper-case letter.

- The lower-case letters set a file-specific bookmark, so that ‘a in one file will take you to a different place than ‘a in another file.
- Uppercase letters set global bookmarks, so that jumping to ‘A will take you to the line you marked in the file where you marked it. This is very handy, but is also sometimes not what you want, because it loads the marked file in the current window.

Pasting in Insert Mode

When in insert mode, you’re not just stuck with typing characters and doing line completion. There are other commands, and one of them is the ˆr command which will read data from a register and type it for you.

If you last deleted the word `falsify` and are typing in some part of the document, you can type ˆr, followed by the default register named “ (doublequote) and the editor will paste the word “falsify” into the text and continue onward in insert mode.

This is particularly helpful when doing something like `cw`, because the change command will delete the current word (loading the buffer) and then enter insert mode. So say I place my cursor on the word falsify above:

What to type	What it does
cw	deletes the current word to register “, and enter insert mode
<\b>	enters the text (we’re in insert mode). This text is the beginning of the html tag for bold text.
ˆr	start the paste-while-in insert mode
”	paste from register ‘ (“falsify”) into the current location in the file.
</\b>	enter the closing tag fror bold
ESC	Return to command mode.
u	Removes the <\b> tag from “falsify”!

Be warned, the ‘.’ command doesn’t see that you used the ˆR command, so if you move to the next word and hit ‘.’, *vim* will change that word to be be the bold-tagged “falsify”. If you want to bold a bunch of different words, you should learn how to record and playback macros (:help q).

Abbreviate!

One of the easiest ways to customize your editor is with abbreviations. For instance, of the most commonly mistyped python lines is the famous “main” invocation:

```
            if __name__ == "__main__":
```

I would like to type the word “pymain” and have the editor replace it with the invocation above. Easy to do:

> :ab pymain if **name** == “**main**”:

Now when I type any non-alphabetic character after the word pymain, it is expanded automatically. All that *vim* needs is an “ab” command and a whole word to expand in insert mode. The expansion is immediate and automatic, there is no hotkey by which you request the expansion. As a result, it will happen when you do not want it to happen. Every time I type pymain, I get the expansion listed above, even if it is an accident. I actually have to type the word wrong exit insert mode and then go back to correct it, because I can’t safely type it at all.

I can add this line to my .vimrc, as long as I leave out the leading colon. My vimrc has a number of abbreviations in it currently, because I choose my abbreviations carefully.

I find that I sometimes type ‘teh’ when I mean ‘the’. This is easy to fix.

> :ab teh the

I never type teh intentionally, so it is a good abbreviation candidate.

You will want to use this feature carefully, so that you do not end up getting unwanted expansions, but it is quite nice if you have common misspellings or long sequences of code that you would otherwise have to type far too often.

Record and playback macros.

In the help system, this is referred to as “complex-repeat”.

You enter macro recording mode by pressing the command ‘q’, followed by a register into which the macro will be stored. You can use any of the alphabetic keys (upper or lower case), and any of the digits. Of course you can have all 26 lower case, all uppercase 26 and all digits assigned to macros at one time if you like.

Every keystroke you type will be recorded until you press the ‘q’ key again.

To replay a macro, you use the @ key, followed by the register name.

Once you have replayed a macro, the undo key will see that macro as a single action. It’s very handy, since a macro can make changes in many lines found throughout the file.

vim remembers what macro you last played, and can repeat it with the double-jump. The double-jump would be “@@”.

The dot command will see it as a single action as well. That’s very cool, because the above lesson becomes much more useful. It works something like this:

qa	Start recording the macro to register ‘a’
cw	deletes the current word to register “, and enter insert mode
	enters the text (we’re in insert mode). This text is the beginning of the html tag for bold text.
ˆr	start the paste-while-in insert mode
”	paste from register ‘ (“falsify”) into the current location in the file.
</b>	enter the closing tag fror bold
ESC	Return to command mode.
q	Stop recording
W	Move one word to the right.
@@	replay the macro, wrapping the word under the cursor
W	Move one more word to the right.
.	replay the macro again, wrapping the word under the cursor

The macro you recorded is just text in a register. You can paste it into a document, edit it to improve its operation, yank it back into the register, etc. Macros provide a nice way to simplify complex edits.

Try “`:help q`” to see more about macro usage. I didn’t tell it all.

Mapping Keys

Whatever you can do by hand, or with a macro, you can also do with key mapping. All a key mapping does is assign a macro to a keystroke. Here is an example:

```
" Move between files in a long list map <F3> :prev<CR> map <F4>
:next<CR>
```

Learn more about mapping via `:help map`.

Colors and Stuff

vim is amazingly customizable, including the colors it uses. Some people make their *vim* themes available on the internet, and a number of color themes are included in the standard distribution. The one you have seen most is probably the one called “default”. If you are in a separate color scheme, you can see it via the command:

:echo g:colors_name

Try some of the existing color schemes like delek, darkblue, desert, koehler, elflord, peachpuff, or slate. You can always return to default, or just exit and reload the editor. The command you need is:

colorscheme delek

For ‘delek’, substitue any scheme you like. You can see all the schemes in the explorer mode by typing:

```
:e $VIM/vim70/colors
```

(assuming you are using vim 7.0. You may have to adjust for verion numbers).

The colorschemes are built from commands that set individual elements such as the foreground and background of the status line, You can learn an awful lot by reading one or two of the colors files.

The color commands start with “hi” (short for highlight), then the kind of thing to color (called a groupname), and then a string of colors to use for plain terminals (ask your dad) denoted as “term”, color terminals denoted as “cterm”, and guis (*gvim*) denoted as “gui”.

Here are a few settings I like for *gvim*:

```
hi LineNr guibg=lightgray guifg=black
hi StatusLine guifg=yellow guifg=darkblue
hi NonText guibg=darkgray
hi ToDo guifg=DarkRed
```

These will color the nontext area that comes past the end of text, the number column on the left of the screen (if you do “set nu”), and colors the active status line (title bar) differently from the status line for inactive tiles/windows.

For more on coloring and theming, you should consult the built-in help (`:help hi`) or perhaps some other more weighty and complete guide to *vim*

Exploiting the path.

There is a special *vim* variable called `path` which will help you to find files which are referenced by the files you will be editing. This feature is specifically useful if you are editing C program files and you point the path variable to the /usr/include/* directories.

The value you provide for path is a comma-separated list of paths where files can be found. This allows you to point to the standard include, and your project includes, and any other dirs you find useful.

To use this feature, place the cursor on the name of a file, and (while in normal/control mode) type `gf` or `\^wf`. The file will be loaded into the current window.

What to Type	**What it does**
set path=.,.**,/usr/include/**,/usr/share/**	Probably excessive, but sets the path to find just about anything. May take a long time.

What to Type	What it does
gf	Goto File: Get the file whose name is under the cursor
ˆWf	Window File: Same as `gf` except opens the file in a new window
:e#	Return to the previous window

I find the window version more useful generally, but I find the non-window version so much easier to type that I will use it instead. I wish that the file navigation would add to the tag stack, so that ˆt would return you to the previous file, but no. The workaround is to set a bookmark as a capital letter so that you can do a return to it from another file. I know that sounds a little awful, and it is a *little* awful, but it works.

Still, for C/C++ programmers, the combination of K and ˆwf allows a lot of file navigation, and the bookmarks are handy for getting in/out of header files and the like.

The path is also used by the name completion (ˆn) system to find the files in which it will search for word completions.

The Alternate File

There are commands in vim which work on "the alternate file", which is usually the file you edited previously.

Above, I listed the `:e#` command which edits the alternate file. That is the long way to type that command, though it is useful to know that `#` means the alternate file when typing commands.

The quick way to type the same command is `Ctrl-^` (control-hat or control-caret). It immediately jumps to the alternate file. This is handy more often than one might think, especially if one is doing test-driven development).

Sadly, the cursor doesn't jump to the window containing the alternate file, but rather brings that file's buffer to the current window. It is still handy, if not ideal.

Alternative Keystrokes

zz	:wq	Save file and quit
ctrl-ˆ	:e#	Switch to alternate file

Epilog

If you have followed the tutorial this far, you have a good start. There is much more to learn, and much further to go.

As a graduate of the Vim Like A Pro school of editing, you must uphold the standards.

- Do Not Park In Insert
- Avoid Death By Caps
- Don’t mindlessly tap-tap-tap
- Make your work easier
- Learn always!
- Remember there are other tools. Use them, too.