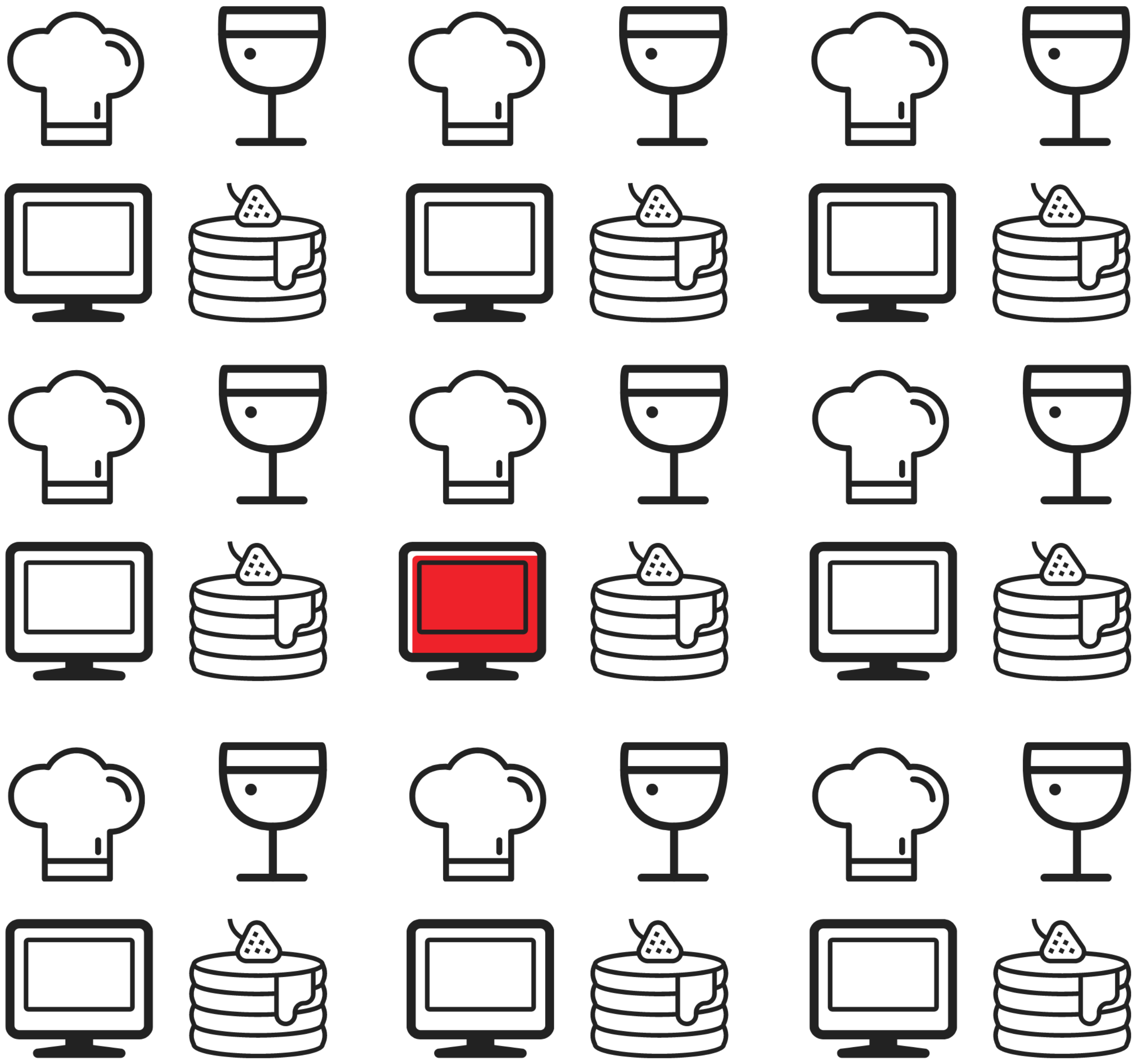

GOURMET VIM: A COOKBOOK

WITH 200+ PRACTICAL RECIPES

IGOR IRIANTO

Gourmet Vim Cookbook

A collection of 200+ Vim recipes

Igor Irianto

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

This version was published on 2021-04-10

Contents

First things

What is this book?

Text editing is an art.

When programming, we write re-usable codes. Why should it be any different with text editing?

The Gourmet Vim explores the art of efficient text editing using Vim, with 200+ practical examples. When learning a new skill, examples are indispensable, having numerous examples will solidify these concepts more effectively.

This book is full of tips and tricks on how to do common (and some uncommon) actions in Vim that you can practice immediately. It will give you a jump start to use Vim like a pro (even if you're already one, there is always something new that you can learn).

When you're done reading this, you'll be better at Vim than ever before.

Some naming conventions

Here I refer to commands starting with `:` as "command-line commands". Other literatures may refer to them as "Ex commands". If you wonder if they are different, they are the same thing.

Here I refer to the terminal commands when accessed from inside Vim as "external commands". You may hear them referred to on the internet as "filter command" or "bang operator". Again, these are all referring to the same thing.

Where are the Vimscripts?

Some tips in this book utilize Vimscript (Vim's built-in scripting language). However, you will find that there is no dedicated section for Vimscript. This is because this book is intended to be mainly for practical Vim uses.

Including a dedicated Vimscript section will expand the scope of this book exponentially because now we are dealing with a whole programming language. However, if you want to learn Vimscript basics, check out my other work Learn-Vim[1].

[1] https://github.com/iggredible/Learn-Vim

Why did you organize this book this way?

In the beginning, I planned to write a giant monolithic cookbook (I actually did). But it was hard to search for things. So I decided to split them into different categories.

The recipes are organized loosely under different categories. Expect some overlaps.

How to read this book?

You can read the recipes in any order. They are completely independent.

This is a practical book. To become good in Vim you need to develop your muscle memory, not head knowledge. You don't learn how to cook by only reading cookbooks. You learn how to cook by actually cooking foods.

You need to type along with every command you see in each recipe. Repeat them several times. Try different variations. Read the `:help` page for that command. Understand the bigger picture. Your goal is to be able to execute that command naturally and instinctively.

Learn the concepts, not the commands.

Navigation

Basic Navigations

Problem:
There are too many commands to remember for moving around in Vim.

Solution:
You don't need to learn all Vim navigation keys. You just need to remember a few important ones to move comfortably.

A few commands I suggest you to commit to memory if you are just starting out:

- `h/j/k/l` - left / down / up / right.
- `w/e/b` - forward to the start of the next word / forward to the end of the next word / backward to the beginning of the previous word.
- `{/}` - previous / next paragraph.
- `gg/G` - start of the file / end of the file.

That's all. Less than a dozen keys. You can learn more navigation keys as you progress.

Toggle Between Files

Problem:
You need to quickly return to the previous file.

Solution:
Use `Ctrl-^` to toggle between the current file and the previously edited file.

When you pass a number to `Ctlr-^`, Vim will go to that buffer number in your buffer list. If you have `5` buffers opened and you run:

- `5 Ctrl-^`, Vim will go to buffer number `5`.
- `2 Ctrl-^`, Vim will go to buffer number `2`.

You can also use `Ctrl-I` and `Ctrl-O` to quickly go to the next and previous cursor location in the jump list.

Vim remembers where you have jumped and stores that in a jump list (`:jumps`). `Ctrl-I` and `Ctrl-O` let you to navigate up and down that list. This is useful for tracing your steps.

A jump is typically a movement that moves your cursor for more than one line (`h/j/k/l` aren't jumps but `{/}` are).

Navigating Parentheses, Brackets, and Braces

Problem:
You are lost in a jungle of parentheses.

Solution:
Use `%` to go to a matching parentheses / brackets / braces (`( ) [ ] { }`).

If you have expressions like `(lambda (y) ((lambda (x) (* x y)) 3))`, you can quickly navigate to the other end of the parentheses with `%`.

Go to a Different Line

Problem:
You need to go to a different line number in the file.

Solution:
Vim has different line operators to go to different lines.

- You can go to line number n with `:n` or `nG`. If you need to go to line 10, run `:10` or `10G`.
- If you pass a count to `%`, it is now a line operator. `50%` jumps to the middle file. `67%` jumps to the 2/3 position of the file.

Same-Line Navigation

Problem:
You need to go to a different column in the same line.

Solution:

- Use `f` and `t` for a forward and backward in-line search. To go to the next “a” in the current line, run `fa`. To go until the next “b”, run `tb`. To repeat this search, use `;` to search forward and `,` to search backward.
- Use `0` and `$` to go to the beginning and the end of the line. You can also go to the first character of a wrapped line with `g0` and to the last character of a wrapped line with `g$`. To wrap lines, run `set wrap`.
- Use `^` and `g_` to go to the first non-blank and last non-blank of the line.
- Use `n|` to go to the column number n of the current line. `10|` takes you to the 10th column.

Screen Adjust

Problem:
You need to center your cursor to the center of the screen.

Solution:

- Use `H M L` to put the cursor at the top, middle, or lower part of the screen.
- Use `zb`, `zz`, and `zt` to move the current line to the bottom, center, or top of the screen.

Scroll

Problem:
You can't use mouse in Vim, hence you can't scroll.

Solution:
Vim lets you use `Ctrl-E / D / F` and `Ctrl-Y / U / B` to scroll up or down a line / half a screen / a whole screen.

Go to the Last...

Problem:
You just did an action and you wanted to go back to the location where you did that action.

Solution:
Vim remembers many actions you did. You can quickly go back to the location where you did that action with marks.

Vim marks can record positions. There are two different mark specificities: line marks (less accurate) and location marks (more accurate).

A line mark uses a single quote, like `'m` (where m is the mark symbol) and a location mark uses a single backtick, like "'m". A line mark will only take you to the correct row. A location mark will take you to the correct row and column. In the following examples, I will only use the line mark, but remember that for each line mark, there is a location mark counterpart.

Use `'[` or `']` to go to the start or end of a recently changed or yanked text.

If you recently ran `yj` on the following phrase:

```
I am a phrase
I am another phrase
```

Pressing `']` puts your cursor on the second line.

Use `'<` or `'>` to go to the start or end of a recently visually highlighted text.

If you recently ran `Vj` to highlight the current line and the line below on the following phrase:

```
I am a phrase
I am another phrase
```

Pressing `'>` puts your cursor on the second line.

Use `'.` to go to the recently modified line

If you recently inserted, modified, or deleted a text, pressing `'.` puts the cursor on that line.

Use `''` to quickly jump to the last position before your last jump. Use double backticks for the location mark counterpart.

Use `'0` to go to the last position before you exited Vim.

Go to the Filename Under the Cursor

Problem:
You need to jump to a file.

Solution:
When your cursor is on a filename, use `gf` to jump to that file.

If your cursor is on “./some_lib.py”, pressing `gf` will take you to the file `some_lib.py` relative to the current directory. If that file doesn’t exist, Vim will throw an error.

Go to the Place Where a Text Was Inserted and Enter the Insert Mode

Problem:
You recently wrote a text and you needed to add to that text.

Solution:
With `gi`, you can go to where the last text was inserted and automatically enter the insert mode.

This is useful when you just wrote something and you needed to add to that text. It only works within a file (if you recently wrote something on a different file, pressing `gi` won’t take you to the other file).

Search

Basic Search

Problem:
You need to do a basic keyword search in Vim.

Solution:
You can do a basic keyword search with `/keyword` to search forward and `?keyword` to search backward. You can either search for a literal word or use a regular expression.

To repeat the last search forward, use `n`. To repeat the last search backward, use `N`.

Starting Regex

Problem:
You want to start using a regular expression in your searches, but don't know where to start.

Solution:
You can get really far with only a little regular expression knowledge. These four should be enough to get started:

- `.` - any character.
- `n+` - one or more n.
- `n*` - zero or more n.
- `n{2,5}` - between two to five n.

You have to escape the `+` and `{ }` symbols or else Vim will treat them as a literal character.

- `/d.g` matches "dog", "dig", "dug".
- `/a\+` matches "a", "aa", "aaaaaaaaaa", etc.
- `/a*` matches "", "a", "aaaaa", etc.
- `/z\{2,5\}` matches "zz", "zzz", "zzzz", and "zzzzz"

Stop Searching After the Last Match

Problem:
You want to stop searching when you reach the last match instead of looping back to the top of the file.

Solution:
This default feature to loop the search matches is enabled by the `'wrapscan'` option. To stop it, use the `'nowrapscan'` option (`set nowrapscan`).

Suppose that in the current document there were 3 matches. When you reached the third match, Vim will automatically cycle back to the first match. Sometimes you want the search to stop after the last match.

With `'nowrapscan'` on, Vim now will stop searching after the 3rd match.

Show the Search Count

Problem:
You want to know how many keyword matches is in the file.

Solution:
To display the match number, add this in your vimrc:

```
set shortmess-=S
```

Now when you search (`/keyword`), Vim will display the number of matches, like `[3/5]`.

Moving the Cursor Below or Above the Match

Problem:
You need to offset the cursor when searching by a few lines.

Solution:
To offset it by n lines, use `/pattern/n`. Your cursor will land on the nth line below the keyword match. If you run `/donut/3`, your cursor will go to 3 lines below the "donut" match. This is the same as running `/donut/+3`.

To go to the third line above the match, run `/donut/-3`.

Moving the Cursor to the End of the Match

Problem:
The cursor always lands at the first character of the match and you need to offset it to the right end of the match.

Solution:
Use the `e` offset modifier when searching. If you run `/donut/e`, the cursor will no longer be on the "d" of "donut", but will be on the "t" of "donut".

To offset the offset, add numbers to the argument:

- `/donut/e-2` to offset it `2` characters to the left of the end match (cursor will be on the letter "n" of "donut").
- `/donut/e+3` to offset it `3` characters to the right of the end match (cursor will be `3` characters after "donut").

Search for a Character Enclosed in a Collection

Problem:
You want to search for either "A", "B", or "C".

Solution:
Use Vim collection (`[]`) to search for any enclosed character. Run `/[ABC]` to match either "A", "B", or "C".

You can use a collection for a sequence of characters too.

- `/[A-Z]` to search for any uppercase letter.
- `/[0-9]` to search for any number.
- `/[A-Za-z]` to search for any lowercase or uppercase letter.

Search for a Keyword at the Start or at the End of the Line

Problem:
You are searching for "Donut" but it has to be at the start of the line.

Solution:

- To search for "Donut" at the start of the line, use `/^Donut`.
- To search for "donut" at the end of the line, use `/donut$`.

Search for a Line Containing a Starting, Middle, and Ending Keyword

Problem:
You need to search for a line that starts with "strawberry", has "banana" somewhere in the middle, and ends with "chocolate". You don't care what goes in-between.

Solution:
Use `/^strawberry.*banana.*chocolate$`.

- `^strawberry` means that the line starts with "strawberry".
- `.*` means zero or more of any character. `.*banana.*` means that anything goes before and after "banana".
- `chocolate$` means that the line ends with "chocolate".

Greedy vs Non-Greedy Search Patterns

Problem:
Sometimes you need to search for the longest pattern and sometimes you need to search for the shortest pattern.

Solution:
There are two types of search patterns: greedy and non-greedy.

- Greedy matches the longest pattern. `.*` is an example of greedy.
- Non-greedy matches the shortest pattern. `\{-}` is an example of non-greedy.

If you have a string "chocolatedonutdonut", running `/ch.*donut` matches the entire string "chocolatedonutdonut" (longest match) while running `/ch\{-}donut` matches "chocolatedonut" (shortest match).

Searching for a Keyword Separated by End-of-Line

Problem:
You need to find keywords that are separated by end-of-line.

Solution:
The `\_.` pattern searches for the subsequent pattern with end-of-line. It is best used with `\{-}` or `*` (non-greedy or greedy) modifiers.

Given the text:

```
strawberry
blueberry
chocolate
pancakes
and more
pancakes
```

- `/strawberry\_.\{-}pancakes` matches (non-greedy) “strawberry”, followed by anything, including newlines, then the first “pancakes” on line 4.
- `/strawberry\_.*pancakes` matches (greedy) “strawberry” to the last “pancakes” on line 6.

Search for Either Or

Problem:
You need to match for either “pancakes” or “waffles”.

Solution:
`/pancakes\|waffles` searches for either “pancakes” or “waffles”.

Avoid Typing Forward-Slashes

Problem:
When you are searching for a path or an URL, like `/some/very/long/path`, it is a hassle having to escape the forward slashes.

Solution:
Instead of searching with `/` and having to escape every single `/` in the pattern like `/\/some\/very\/long\/path`, use the `?` search operator instead. `?/some/very/long/path`.

Note that `?` will search backward, but you can just use `N` to search “forward”. This saves you from having to type escapes.

Marking the Start and End of a Match

Problem:
You want to match for a part of a longer pattern.

Solution:
Use the `\zs` (start) and `\ze` (end) to mark the start and end of a match.

Given the following text:

```
sugardonut
chocodonut
chocolate
```

Explanation:

- To match the "donut" in "chocodonut", run `/choco\zsdonut`.
- To match the "choco" in "chocodonut", run `/choco\zedonut`.
- To match "do" in "sugardonut", run `/sugar\zsdo\zenut`.

Search for the Nth Occurrence in a Line

Problem:
You need to search for every 2nd occurrence on each line.

Solution:
Running `/\(.\{-}\zsdonut\)\{2\}` will match every second "donut" occurrences (with very magic: `/\v(.{-}\zsdonut){2}`)

```
Chocolate donut, blueberry donut, raspberry donut
```

Search for Any Text Surrounded by a Particular Pattern

Problem:
You need to search for any text, any length, that is surrounded by a pair of double-quotes.

Solution:
To search for any text that is surrounded by a pair of double-quotes, use `/"[^"]*"`.

This pattern is useful to search for a body of text surrounded by a character.

- To search for any text surrounded by a pair of single quotes, use `/'[^']*'`.
- To search for any text surrounded by a pair of commas, use `/,[^,]*'`.

The collection syntax, `[]`, when you add a caret (`^`) means character negation. `[^"]*` means zero or more any non-double-quote character. `[^']` means any non-single-quote character.

Repeat the Last Search

Problem:
You want to repeat the last search.

Solution:
To repeat the last search, use `//` or `??`. You can also run an empty search `/<Enter>` or `?<Enter>` to use the last search.

Force a Case-Sensitive Search

Problem:
You have `ignorecase` and `smartcase` options and you need to specifically search for a lowercase word.

Solution:
Adding `\C` to the keyword forces a case-sensitive search.

This is useful if you have `ignorecase` and `smartcase` settings because Vim will do a case insensitive search if you do `/donut`. However, `/donut\C` will explicitly match for “donut” and not “Donut” or “DONUT”.

Optional Search

Problem:
You need to search for optional phrases in a word.

Solution:
The `\=` modifier makes the preceding character optional.

If you have the string “hero”, “heroes”, and “heros”, `/hero\(es\)\=` will match “hero” and “heroes” (the suffix “es” is optional). While `/heros\=` will match “hero” and “heros” (the suffix “s” is optional).

Search a Phrase in Multiple Files With Vimgrep

Problem:
You need to search for “food” inside multiple files.

Solution:
Vim has a built-in grep to search in files named `:vim` (you read that right, this command is called vim, short for `:vimgrep`).

To search for "food" in all Javascript files recursively, run `:vimgrep /food/ **/*.js`. To search for "food" in all Javascript files in the current directory, run `:vimgrep /food/ *.js`.

Vimgrep uses quickfix (`:h quickfix`). Use `:copen` to display the results.

Alternatively, you can also use the terminal grep program. Vim has a `:grep` command that uses whatever command is assigned to the `'grepprg'` option (by default, it uses the terminal grep). To search for "donut" in the current directory, run `:grep "donut" . -R`.

Adding Matches to an Existing Quickfix List

Problem:
You did a search with vimgrep and you need to add more items to the result.

Solution:
Vimgrep has an append version: `:vimgrepa`.

If you run `:vimgrep /breakfast/ **/*.js` followed by `:vimgrep /lunch/ **/*.js`, the second vimgrep search overwrites the first one. That's not good.

To add the search results, first run `:vimgrep /breakfast/**/*.js`, then run `:vimgrepa /lunch/ **/*.js`. Now you have a list of both "breakfast" and "lunch" searches.

Alias for Digits and Words

Problem:
Typing `[0-9]` is too verbose. Is there a shorter alias?

Solution:
Vim has predefined characters to help with search. `\d` is an alias for digits `[0-9]`. To search for a digit followed by "foo", run `/\dfoo`. This will match "1foo", "2foo".

Similarly, `\w` is an alias for a "word" character `[0-9A-Za-z_]`. To search for the number `0` followed by any two word characters like "0ab" or "0a1" or "0_1", run `/0\w\w`.

More Predefined Characters

Problem:
You wonder if there are more useful aliases like `\d` and `\w`.

Solution:
The next 5 useful character aliases (in my opinion), are:

```
\s    Whitespace character (space and tab)
\S    Non-whitespace character (everything except space and tab)
\w    Word character [0-9A-Za-z_]
\l    Lowercase alphas [a-z]
\u    Uppercase character [A-Z]
```

If you want to see the whole list, check out `:h /character-classes`.

Quickly Search for the Word Under the Cursor

Problem:
You want to quickly search for the word your cursor is on.

Solution:
Use `*` and `#` to search for the word under the cursor forward and backward. If your cursor is on the word "donut", if you press `*`, Vim does `/\<donut\>`. `\<` and `\>` means the beginning and end of a word. It matches "chocolate donut" but not "chocolatedonut".

To match the latter, use `g*` and `g#`. These commands are similar to `*` and `#`, but instead of treating the word under the cursor as a whole word, Vim treats it as a string pattern. If your cursor is on the word "donut" and you press `g*`, Vim does `/donut`. It matches "glazed donut", "donuts", "donutman", etc.

Finding a File

Problem:
You need to search and open a file.

Solution:
At the very core, you can open a file in Vim with `:e yourfile.txt`. Vim also has the `:find` command. When you open a file with either `:e` or `:find`, you can use `<Tab>` to autocomplete the directory and/or file name. Try it yourself: go to the directory with several files and press `:e` then `<Tab>`. You'll find that Vim tries to autocomplete the name.

You may wonder, what's the difference between `:edit` and `:find`? The difference is that `:find` finds file in a path while edit doesn't.

To check what your paths are, run:

```
:set path?
```

By default, yours probably look like this: `path=.,/usr/include,,`. The big picture here is that you can edit your path so you can find files faster.

Suppose that this is your project structure:

```
app/
  assets/
  controllers/
    application_controller.rb
    comments_controller.rb
    users_controller.rb
    ...
```

The above is an example of a Rails project structure. To go to the `users_controller.rb` from the root directory, you have to go through several directories and pressing a considerable amount of tabs. You have to do something like `:find a<Tab>c<Tab>u<Tab>` before finally getting to `app/controllers/users_controller.rb`.

Let's create a shortcut for the `controllers/` directory by adding the controller path to the current `'path'`. Run:

```
:set path+=app/controllers/
```

Now when you type `:find u<Tab>`, Vim now searches inside `app/controllers/` for files starting with `u`. You've created a path shortcut for autocompletion! You can update the path with frequently-visited paths.

Search and Replace in Multiple Files

Problem:
You need to substitute texts in multiple files.

Solution:
You can use `:vimgrep` or `:grep` combined with `cfdo` to do multiple-file substitution.

Let's say you want to replace all "pancake" with "waffle" inside all files.

First, find all the instances of that word with `:grep "pancake" . -R` (you can also run `:vimgrep /pancake/ **`). Then run `:cfdo %s/pancake/waffle/g | update`.

Command-Line Mode

Command Suggestion

Problem:
You want to see all the relevant commands.

Solution:
While you are in the command-line mode, press `Ctrl-D` for a list of relevant commands.

If you type `:h fo`, then press `Ctrl-D`, Vim shows all relevant keywords containing the phrase "fo".

Get the Word Under the Cursor

Problem:
You want to get the word under the cursor to use it in your command.

Solution:
While in the command-line mode, you can get:

- The filename under the cursor with `Ctrl-R Ctrl-F`.
- The word under the cursor with `Ctrl-R Ctrl-W`.
- The current line with `Ctrl-R Ctrl-L`.
- The current file name with `Ctrl-R %`.

Faster Delete

Problem:
Deleting one character at a time in the command-line mode takes forever.

Solution:
If you make a mistake typing a command-line command, you can delete faster with:

```
Ctrl-H    Delete one character
Ctrl-W    Delete one word
Ctrl-U    Delete the entire line
```

These are the same shortcuts as the insert mode.

Are There More Commands?

Problem:
You want to learn what other commands Vim has.

Solution:
To get a list of all command-line commands, check out `:h ex-cmd-index`.

File

Buffers, Windows, Tabs

Problem:
You heard that Vim uses buffers, windows, and tabs, but you're not sure what they mean.

Solution:
Buffers, windows, and tabs are Vim's approach to file and layout management. Many text editors only use two abstractions: windows and tabs.

So what is this buffer? A buffer is an in-memory space for text. When you open a file in Vim, Vim opens a buffer to store that file data. When you open `3` files in Vim, you have `3` buffers opened.

By the way, make sure you have the `set hidden` option in Vimrc. Without it, whenever you switch buffers and if your current buffer has unsaved changes, Vim complains and prompts you to save the file. This can slow you down.

A window is a viewport on a buffer. From the outside, Vim windows look like most text editors' windows. You can have multiple windows opened and arranged vertically and / or horizontally. Multiple windows can point to the same buffer. Remember, each buffer represents each opened file. You can have two windows opened and with both pointing to the same buffer, `file1.txt`. You can have only one window open but you actually have `3` opened buffers, `file1.txt`, `file2.txt`, and `file3.txt`. A window is what you use to see buffers. A buffer is the actual file.

A tab is a collection of windows. Think of it like a layout for windows. In modern text editors, if you close a tab, the file in that tab goes away. Similarly, in a web browser, if you close a web page tab, that page is gone. In Vim, a tab does not represent an open file (remember, a buffer represents the actual file). You can close a tab with a window displaying `file1.txt`, but that file is still opened in the buffer behind the scene.

In short:

- A buffer is an in-memory space of a file.
- A window is a viewport on a buffer.
- Two windows can point to the same buffer or two windows can point to two different buffers.
- A tab is a layout for windows.

Two Different Write Commands With Similar Syntax

Problem:
You get two very different results running `:w !food` and `:w! food`. What's the difference?

Solution:
`:w !food` and `:w! food` are actually two different commands (although their syntaxes are very similar).

- `:w !food` executes the external command `food` (if if doesn't exist, it throws an error) with your buffer content as the STDIN.
- `:w! food` saves the current buffer as `food`.

Save a File That Requires a Root Permission

Problem:
You need to save a file but you need to do it with a root permission

Solution:
Save the file that requires a root permission with `:w !sudo tee %`. This uses similar write syntax as above. Recall that `:w !cmd` executes the terminal command `cmd`. In this case, the command is `sudo tee`.

`%` in Vim represents the current file.

Time Travel

Problem:
You need to restore your file state to the state it was 15 minutes ago.

Solution:
You can go back to an earlier text state with `:earlier`. To go back to the text state 15 minutes ago, run `:earlier 15m`. This command accepts other units: `s` (seconds), `h` (hours), `d` (days), and `f` (number of saves).

To go to a later state, there is a `:later` counterpart (I wish I could just run `:later 8h` and be done with my work, but that won't work…).

Converting the Current File to an HTML

Problem:
You want to convert the current file to a HTML, a la Vim.

Solution:
Vim has a `:%TOhtml` to convert the HTML version of the current document. It will generate a new file in the current directory `your_original_file_name.txt.html`.

Open an URL Content

Problem:
You need to edit the content of a web page.

Solution:
You can fetch the content of an URL directly `vim https://www.google.com/`.

Save a Partial File

Problem:
You need to save only parts of a file.

Solution:
You can save a partial file with `:{range1},{range2}w filename`. To save lines 5 and 10 as `partial_-stuff.txt`, run `:5,10w partial_stuff.txt`.

Show the Buffers List

Problem:
You need to see which buffers are open.

Solution:
Vim has the commands `:ls`, `:buffers`, and `:files` to show the buffers list.

Why use the buffer? Buffer is Vim's approach to quickly go to different files quickly. Most text editors have only two abstractions: windows and tabs. Vim has three: buffers, windows, and tabs.

Vim saves all the currently opened files in memory, in buffers. If you have `file1.txt`, `file2.txt`, and `file3.txt` opened, you have 3 buffers.

There are different approaches to jump to a particular buffer:

- Numerical approach: to go to buffer #9, run `:buf 9`.
- Filename approach: to go to `file2.txt` buffer, run `:buf file2.txt`.
- `:bnext` and `:bprevious` go to the next and previous buffer in the list.
- `:brewind` goes to the first buffer in the buffer list (`:blast` goes to the last buffer - having a blast yet :D?).

Finally, use `:bd file1.txt` to remove `file1.txt` from the buffer list.

Buffer is a clever approach to file management when editing with Vim. It creates for you a list of shortcut files. I strongly recommend that you utilize buffers more in your daily Vim editing!

Vim Tabs

Problem:
You need to open a different tab.

Solution:
Vim has a tab system to contain your window layouts. All buffers are shared between different tabs.

Some useful tab navigations are:

```
:tabnew file.txt    Open file.txt in a new tab
:tabclose           Close the current tab
:tabnext            Go to next tab
:tabprevious        Go to previous tab
:tablast            Go to last tab
:tabfirst           Go to first tab
```

By the way if you want to start Vim with multiple tabs, run this from the terminal:

```
vim -p file1.txt file2.txt file3.txt
```

Save the Current File in a Different Directory

Problem:
You need to save the current file in a different directory with the same name.

Solution:
To quickly save the current file in a different directory with the same name, run `:w a/different/directory/%` (recall that `%` is Vim's special variable for the current file).

Open the Man Page From Vim

Problem:
You need to quickly search the man page definition for the word under the cursor.

Solution:
Use `K` to quickly open up the man page for the word under the cursor.

Quitting Vim

Problem:
You can’t exit Vim.

Solution:
This is a classic Vim question. The truth is, there are many ways to exit Vim.

The standard ways:

```
:q
:wq
:q!
:qa
:qa!
```

The “I know commands that most people don’t” ways:

```
:x
:xa
:quita
:exit
:conf wqa
```

The normal mode way:

```
ZZ
ZQ
```

The “I enjoy pain and suffering” way:

```
:!ps ax | grep vim | grep -v grep | awk '{print $1}' | xargs kill -9
```

Finally, the brute force way:

```
Unplug your computer. Bzzt.
```

Viewing Recent Files

Problem:
You need to go to a recently edited file.

Solution:
Vim has `:oldfiles` to display, chronologically, the files you’ve edited in the past.

Combine this with `:browse` to browse and select one of the recently opened files (`:browse oldfiles`).

Changing Directory

Problem:
You need to change the current directory.

Solution:
You can change the directory from Vim with `:cd {path}`.

- You can go to the previous directory with `:cd -`.
- You can go to a different directory with `:cd some/other/dir/`.

You can also print the current directory name with `:pwd`.

Executing a Command When Opening a File

Problem:
You need to modify the file you are about to open with some Vim commands.

Solution:
You can pass a command when opening a file with `:edit` with `+{cmd}`.

To substitute all "toast" with "bagel" when you open `breakfast.txt`, run `:edit +%s/toast/bagel/g breakfast.txt`.

Get the Current File Info

Problem:
You need to get the current file name to the clipboard.

Solution:
You can copy the current filename to the clipboard with `:let @*=expand("%")`

Explanation:

- `@*` is one of the clipboard registers.
- `=expand("%")` gets the current file name

If you use this often, add this mapping in vimrc:

```
nnoremap <Leader>cf :let @*=expand("%")<CR>
```

You can add various modifiers to the `%` current file pattern. Some useful ones are:

```
%:p (full path)
%:. (relative to current directory)
%:h (head of file)
%:t (tail of file)
%:e (extension of file)
```

Count Words

Problem:
You need to count the words in the current file.

Solution:
You can count words with `g Ctrl-G`. It will return something like:

```
Col 1 of 0; Line 141 of 157; Word 748 of 774; Byte 4489 of 4976
```

You can also use visual selection to specify the text block. Highlight a block of text, then press `g Ctrl-G`.

```
Selected 5 of 293 Lines; 70 of 1884 Words; 359 of 10928 Bytes
```

Alternatively, you can count the occurrence of a specific keyword with `:s` command and the `gn` flags:

- To count characters: `:%s/./&/gn`
- To count empty lines: `:%s/^$/&/gn`
- To count a specific word, "donut": `:%s/\<donut\>/&/gn`

Delete a File

Problem:
You need to delete a file.

Solution:
You can delete files from Vim with the `delete()` function. Run either `:call delete(expand('%'))` or `:call delete(@%)` to delete the current file.

However, this still leaves the deleted file on the current buffer. Add `:bdelete!` to delete the current buffer.

For a more complete command:

```
:call delete(@%) | bdelete!
```

More Concise Way to Save and Exit

Problem:
`:wq` to save and exit is a hassle to type. Is there a shorter way to do this?

Solution:
Instead of `:wq` to save file, use `:x` to type 33% less.

The `:x` command will:

- Save if there are unsaved changes then close the buffer, or
- Close the buffer if there are no unsaved changes.

Suspend Vim

Problem:
You need to suspend Vim.

Solution:
You can suspend Vim with `:stop` or `:suspend`. You can also press `Ctrl-Z` at any time from inside Vim.

To return, run `fg` from the terminal.

Compiling

Problem:
You need to compile from Vim.

Solution:
Vim has a `:make` command that is similar to the `make` program. In fact, the `:make` program uses whatever command is defined by the `'makeprg'`, which by default is using the `make` command.

If you have a `Makefile`:

```
all:
        echo "build, run"
build:
        g++ hello.cpp -o hello
run:
        ./hello
```

Running `:make build` will run the build command and running `:make` will run all.

You can change it to run any command. You can make it to run typescript, ruby, g++, anything. For example, to change it to run `g++` on the current file, run `:set makeprg=g++\ %`.

Persist Local Configuration

Problem:
You need to persist some settings in Vim so they will still be there when you re-open Vim.

Solution:
You can use Vim View to persist Vim settings so when you close and re-open Vim, they will still be there. Think of View as a configuration "snapshot".

Before you exit Vim, your cursor was probably on a specific row and column. Maybe you had some folds and created local mappings. When you take a View of the current file, Vim takes a snapshot of these settings.

The attributes that Vim remembers can be viewed (pun intended) inside the `'viewoptions'` setting. Run `set viewoptions?` to see what Vim currently has. For example, if you want to add to the `'viewoptions'` the `localoptions` attribute, run `:set viewoptions+=localoptions`.

To save a view, run `:mkview`. Vim saves a view in the path specified by `'viewdir'` (to see it, run `:set viewdir?`).

The next time you open that file again, load your View with `:loadview` and you should have the old settings back!

If you want to automate the View creation when you leave a buffer, add this in Vimrc:

```
autocmd BufWinLeave *.txt mkview
```

To automate loading View when you open a buffer, add this in Vimrc:

```
autocmd BufWinEnter *.txt silent loadview
```

To see all attributes that a `viewoptions` can save, check out `:h 'viewoptions'`.

Saving a Session

Problem:
You want to preserve the window layout the next time you re-open Vim.

Solution:
Sometimes you leave Vim with multiple windows opened, folds created, and buffers opened, and you want to preserve them so when you return, everything looks just like it did before you left.

You can do that with Vim Session. A Session is similar to View, except it has a broader scope. A View doesn't save your layout but a Session does.

Session saves whatever attributes are defined in `'sessionoptions'` (`set sessionoptions?` to see it). You can add, subtract, or update it. To remove the `terminal` attribute for example, run `:set sessionoptions-=terminal`. To see all the attributes the `'sessionoptions'` supports, check out `:h 'sessionoptions'`.

To save a session, run `:mksession`. This will create a file `Session.vim` in the project root file. By the way, if you want to save the session somewhere else, you can pass it a new path argument like `:mksession ~/some/other/place.vim`.

To open Vim using a previously saved session, run `vim -S Session.vim` or you can run `:source Session.vim` from Vim.

History

Jumping Around Files Quickly

Problem:
You need to quickly jump to places that you have been to.

Solution:
Some movements in Vim count as jumps. Vim remembers these jumps and stores them in the jump list.

To see the jump list, use `:jumps`. The jump list shows all the places you've jumped to recently. You can travel up and down the jump list with `Ctrl-O` and `Ctrl-I`.

Vim considers these commands as jump commands:

```
'       Go to the marked line
`       Go to the marked position
G       Go to the line
/       Search forward
?       Search backward
n       Repeat the last search, same direction
N       Repeat the last search, opposite direction
%       Find match
(       Go to the last sentence
)       Go to the next sentence
{       Go to the last paragraph
}       Go to the next paragraph
L       Go to the the last line of displayed window
M       Go to the middle line of displayed window
H       Go to the top line of displayed window
[[      Go to the previous section
]]      Go to the next section
:s      Substitute
:tag    Jump to tag definition
```

Try this: move around with jump movements, then press `Ctrl-O` and `Ctrl-I` a few times. `Ctrl-O` and `Ctrl-I` are two of my favorite commands.

Trace Edited Texts

Problem:
You need to trace some texts you recently edited.

Solution:
Vim tracks the locations of your recently edited texts. You can view them with `:changes`.

You can go up and down this list with `g;` and `g,`. Vim makes it easy to trace recently changed texts.

Search History

Problem:
You need to search the previous searches.

Solution:
Sometimes you need to re-search a keyword but you can't quite remember what the keyword was exactly. `q/` and `g?` can help you. They can show the history of previous searches.

Typing `q/` or `q?` show all the `/` or `?` searches you've done. It will open a new window that you can navigate around. If you press `<Enter>` on one of the keywords, Vim will do a search using that word.

Command-Line History

Problem:
You need to run a command similar to a command you executed before.

Solution:
`q:` shows the history of the command-line commands.

If you press `q:`, Vim will take you to a history window with a list of command-line commands you have used. Regular Vim navigation works here. If you press `<Enter>` on a command, Vim will execute that command.

If in the past you ran `:s/strawberryhoneypancakes/kale/g` and you want to run `:s/strawberryhoneywaffles/kale/g`, instead of retyping the commands all over, why not find that command from the command history, modify it slightly, and then execute it?

Press `q:`, find that command in history, then edit it. Once you're done, run it.

The great thing about these histories (`q:`, `q/`, `q?`) is that you can use Vim navigation, making navigation in this mode very easy.

Another way to enter the command-line history is while you're in the command-line mode (`:`), press `Ctrl-F`. This also works with `/` and `?`. During the `/` or `?` search, type `Ctrl-F` to open the search history window.

Vim Histories

Problem:
You want to know what other things Vim store.

Solution:
Vim stores 5 different histories

```
:his c or :       - command-line history (q:)
:his s or / or ?  - search history (q/ or q?)
:his e or =       - expression history
:his i or @       - input history
:his d or >       - debug history
```

Insert Mode

Entering Insert Mode

Problem:
You need to go to the insert mode.

Solution:
There are many ways to enter the insert mode. The most common ones are:

```
i    Insert text before the cursor
I    Insert text before the first non-blank character of the line
a    Append text after the cursor
A    Append text at the end of line
o    Starts a new line below the cursor and insert text
O    Starts a new line above the cursor and insert text
```

If you're new to Vim, don't memorize them all. Start with `i` first. Then commit `a` and `o` to memory. Finally, you can commit their uppercase counterparts.

Different Ways to Exit Insert Mode

Problem:
Your keyboard doesn't have an `<Esc>` key and you need to quickly exit insert mode.

Solution:
Normally (pun intended) you can exit the insert mode with `<Esc>`. However, not all keyboards have the `<Esc>` key (like tablet keyboards). You can use either `Ctrl-[` or `Ctrl-C` to escape in addition to `<Esc>`.

Alternatively, you can create a mapping to escape while in the insert mode:

```
inoremap jk <Esc>
```

I find that the `<Esc>` button is too far, so I map the `<Caps-lock>` button to `<Esc>`. If you search for Bill Joy's ADM-3A keyboard (the Vi creator), you will see that his `<Esc>` key is not located on the far top left like modern keyboards but to the left of `q` key. This is why I think it makes sense to map `<Caps lock>` to `<Esc>`.

Paste a Recently Yanked Text

Problem:
You need to paste a previously yanked text.

Solution:
If you need to paste a text while in the insert mode, you could first quit the insert mode then paste it with `p`, but that is mode-switching. A better way to paste text while in insert mode is to use `Ctrl-R 0`. Doing `Ctrl-R` while in the insert mode lets you to insert the content of a register. `0` is the yank register.

You can insert from any other registers as well. If you need to paste the content from the register, run `Ctrl-R a`.

Performing a Quick Calculation

Problem:
You need to do a quick calculation and insert the calculation result.

Solution:
To do a quick calculation while in the insert mode, you can use `Ctrl-R` to use the expression register (=).

For example: `Ctrl-R =1+1<Enter>`.

Autocompletion

Problem:
You need to autocomplete a word.

Solution:
The insert mode has built-in autocompletions.

While in the Insert mode, use `Ctrl-P` for word completion. By pressing `Ctrl-P` in the insert mode, Vim searches backward for existing words that start with the text under the cursor (press `Ctrl-N` for forward search).

There are other auto-completions. Some of the common ones are:

- `Ctrl-X Ctrl-L` for a line autocompletion.
- `Ctrl-X Ctrl-F` for a filename autocompletion.
- `Ctrl-X Ctrl-K` for a dictionary autocompletion. Note: this method requires you to set the `'dictionary'` first. Assuming that there is a dictionary in `/usr/share/dict/words` (your computer should have some sort of dictionary path somewhere), add that path into the dictionary using `set dictionary=/usr/share/dict/words`.

For more, check out `:h ins-completion`.

Executing a Normal Mode Command While in the Insert Mode

Problem:
You need to execute a normal-mode command without exiting insert mode.

Solution:
You can execute a normal mode command while in the insert mode with `Ctrl-O`.

For example:

- `Ctrl-O zz` to center the screen.
- `Ctrl-O /food` to jump to the next “food”.

Larger Deletes

Problem:
You need to delete text chunks.

Solution:

- Press `Ctrl-U` to delete the entire line.
- Press `Ctrl-W` to delete the word before the cursor.

Meanwhile, `D` in normal mode deletes to the end of the line, but there is no insert mode command that does that, so why not create one?

Add the following mapping:

```
inoremap <C-d> <C-O>D
```

Now when you press `Ctrl-D` in Insert mode, Vim deletes to the end of the line.

Scrolling While in the Insert Mode

Problem:
You need to scroll from insert mode.

Solution:
Press `Ctrl-X Ctrl-E` and `Ctrl-X Ctrl-Y` to scroll down and up.

Insert Special Characters

Problem:
You need to enter special characters like the Π.

Solution:
Use Vim digraph. To enter a greek Pi, in insert mode, press `Ctrl-K P*`.

How did I know to use `P*`? I looked it up the digraph table (`:h digraph-table`). You can enter all sorts of characters like Φ Ω Λ Θ Π etc.

Layout

Indent and Un-indent Lines

Problem:
You need to indent and unindent lines.

Solution:
The `<` and `>` operators can indent and un-indent lines.

- To indent the next `3` lines, run `>2j`.
- To indent to the end of the file, run `>G`.
- `>>` and `<<` to indent and un-indent the current line.

Fix the Indentation of the Entire File

Problem:
The entire file's indentation is messy.

Solution:
`={motion}` can correct the indent of the lines indicated by the motion.

For example, `=G` fixes the indentation of the current line to the end of the file. A common practice is to use `gg=G` to correct the indentation of the entire file.

Note that the `=` operator uses the program defined in the `'equalprg'` setting, which by default is empty (when it is empty, Vim uses `'indent'`).

Redraw the Screen

Problem:
Occasionally Vim will display a stagnant screen after running a certain scripts.

Solution:
If Vim is showing a stagnant or "buggy" display, you may need to refresh it by either running `:redraw!` or pressing `Ctrl-L`.

Folding Lines

Problem:
You want to fold rows of lines.

Solution:
I don't like to fold my clothes but I like to fold rows of lines.

Why would you ever fold anything?

When you read a file, there are often many irrelevant texts that obstruct you from understanding what that file does in the big picture. To hide this unnecessary noise, use Vim fold.

Vim supports 6 different folds:

1. Manual
2. Indent
3. Expression
4. Syntax
5. Diff
6. Marker

The manual fold lets you to fold rows manually (you can't fold characters, just lines). The operator is `zf` and it works just like any operation: it accepts a motion. If you have the text:

```
Line one
Line two
```

With your cursor on "Line one", run `zfj` to fold both lines. Folded lines look something like:

```
+-- 2 lines: Line one -----
```

To open a folded text, use `zo`. To close a fold, use `zc`. You can also use the `:fold` command-line. My two favorite fold methods are `zR` to open all folds and `zM` to close all folds. The manual fold is useful when you want to do quick, impromptu folds.

To use the indent fold method, change the `'foldmethod'` option to indent: `set foldmethod=indent`. Vim will fold indented lines according to the `'shiftwidth'` option.

If you have the text:

```
One
  Two
  Two again
```

The latter 2 lines, having 2 spaces of indentation, will be folded. This will only work if you have `'shiftwidth'` value of 2 btw. The indent fold is useful when dealing with texts that use indent structure, like Python or YAML files.

The expression fold lets you define an expression to determine folding. You need to first tell Vim to use the expression fold with `set foldmethod=expr`, then Vim will use whatever expression you define for `'foldexpr'` option.

For example, if you want to fold all items starting with “p”:

```
donut
pancake
pop-tarts
protein bar
salmon
scrambled eggs
```

Run `:set foldexpr=getline(v:lnum)[0]==\\"p\\"`. I won’t go into Vimscript deeply here, but the `'foldexpr'` expression will return true only if the first character of the current line starts with “p”.

The syntax fold can fold lines based on syntax options. To use it, first set the fold method with `:set foldmethod=syntax`. Again, I won’t cover how syntax definitions work in Vim here, but in general, if you use syntax plugins like vim-polyglot[2], the syntax fold should work right out of the box. Vim uses the syntax definition to determine what to fold.

The diff fold works immediately when you do a vimdiff. If you run `vimdiff file1.txt file2.txt`, Vim automatically sets the foldmethod to diff and folds all the identical lines.

The marker fold is used to fold the lines marked by a pattern. You can set it up with `:set foldmethod=marker`. By default, Vim uses `{{{, }}}` patterns as fold identifiers. Vim will fold any rows between the identifier.

If you have:

[2]https://github.com/sheerun/vim-polyglot

```
Hello

{{{
world
vim
}}}
```

When Vim sees `{{{` and `}}}`, those lines will be folded. You can change the indicators by updating the `'foldmarker'` option values. To change it to `<<<` and `>>>`, run `:set foldmarker=<<<,>>>`.

Cursor Visual Helper

Problem:
You need to draw a straight line across the file.

Solution:
Use `:set cuc` (`'cursorcolumn'`) to display a column that spans across the entire screen. Try it!

To turn it off, run `:set nocuc`. If the cursor column has a hard-to-see highlight, you can change the highlight to white by running `:highlight CursorColumn guibg=#FFFFFF`.

Macros And Registers

Macro Basics

Problem:
You need to record a command sequence to a register.

Solution:
You can record a sequence of actions inside registers a-z and replay them later. This procedure is called a macro.

To record a macro, press `q`, followed by the register character (example: `qq` to record a macro in register q). To stop recording, press `q` again.

For example:

```
qagUUjq
```

- `qa` to record a macro in the register a.
- `gUU` to uppercase the current line.
- `j` to go down one line.

To replay a recorded macro on register a, run `@a`. To play the last executed macro, run `@@`.

Register Basics

Problem:
You heard that Vim can save texts in different registers.

Solution:
Vim registers work like your computer clipboard. Vim comes shipped with 10 registers. I won't go over them in detail here. They are:

- The unnamed register (`""`).
- The numbered registers (`"0-9`).
- The small delete register (`"-`).
- The named registers (`"a-z`).
- The read-only registers (`":`, `".`, and `"%`).
- The alternate file register (`"#`).

- The expression register (`"=`).
- The selection registers (`"*` and `"+`).
- The black hole register (`"_`).
- The last search pattern register (`"/`).

They can be used in a composition with other operators. Run `"` followed by the register symbol, then the operator. Registers are commonly used with the yank (`y`) operator and the delete (`d`) operator, but it can be used with other operators too. You can also enter the content of a register directly. `:put a` outputs the content from register a. While in insert mode, you can press `Ctrl-R a` to get the content from register a.

If you need to yank the current word into register a, run `"ayiw`. To paste that content, run `"ap`.

Registers and macros are closely related. If you think about it, macros are executed registers. When you record a macro in register a (`qagUUjq`), you are actually recording a string of instructions (if you don't believe me, after running `qagUUjq`, run `"ap` and you should see the key sequence of your actions printed out).

Closer Look At Vim Registers

Problem:
You saw that Vim has 10 registers. How do you use them?

Solution:
As mentioned above, Vim has 10 different registers. Some are automatically filled by Vim while some have to be explicitly inserted.

The unnamed register stores the last text you yanked, changed, or deleted. The unnamed register is like a computer's standard copy / paste operation. To get the text from the unnamed register, do ""p. The paste commands, `p` and `P`, automatically pastes from the unnamed registers.

There are two types of numbered registers: the 0 register and the 1-9 registers.

The 0 register is reserved for the yanked register (also known as the yank register). If you yank a line, like `yy`, it will be stored in register 0. `"0p` to paste it.

The 1-9 registers are reserved for deletions or changes. If you delete a line, `dd`, the deleted text will be stored in the numbered register starting from 1 going all the way up to 9, chronologically. If you do `dd` 3 times, the most recent deletion will be stored in register 1, the second most recent deletion in 2, and the third recent deletion in 3. To paste them, use `"1p`, `"2p`, `"3p`, etc.

The small delete register stores deletions or changes that are less than one line. If you delete a word (like `diw`), you can access it with `"-p` (the registers 1-9 only store deletes or changes that are at least one line size).

The named register is probably one of the most used registers. It uses the characters a-z to store yanked, changed, or deleted texts. To delete a word into the register a, run `"adiw`. To paste it, run `"ap`. To yank the rest of the line into the register b, run `"by$`. To paste it, run `"bp`.

The read-only registers are automatically handled by Vim. `.` stores the last inserted text, `:` stores the last executed command-line command, and `%` stores the name of the current file.

The alternate file register stores the name of the alternate file (the other file you just edited).

The expression register lets you evaluate Vimscript expressions. To get the result of calculating the expression `1 + 1`, run `"=1+1<Enter>`, then press `p`.

The selection registers, `*` and `+` are used to connect Vim to your computer clipboard. By default, if you copy something (for example, a text that you copy from a website), you can't paste it immediately to Vim and vice versa (if you yank something in Vim, you can't paste it outside of Vim). To paste a text from the system clipboard into Vim, run either `"+p` or `"*p`. The difference between `*` and `+` registers are historical. If Vim has `+xterm_clipboard` and your machine uses X11's primary selection, Vim uses `*` register. If your machine uses X11's clipboard selection, Vim uses `+`. To be frank, don't worry about it too much. Most of the time, they are interchangeable.

However, I find that doing `"*p` to paste and `"*yy` to yank is a lot of work. Add this in vimrc:

```
set clipboard=unnamed
```

This connects the unnamed (`p`) register to the clipboard system. Now you can yank and paste naturally with `p`.

The black hole register lets you delete, yank, and change text without storing it into the register. If you delete a line using `"_dd`, Vim won't save it in the numbered registers.

The last search pattern register `/` and `?` stores the last search term.

Clear Up a Register

Problem:
You need to empty a register.

Solution:
To quickly clear up the register a, run `qaq`. This command records an empty macro, effectively emptying that register.

Output Content From Any Register

Problem:
You need to output the content from a register directly.

Solution:
The `:put` command can output the content from any register

- `:put a` prints the register a.
- `:pu _` prints from the blackhole register (in this case, it outputs a blank line).

Edit an Existing Macro

Problem:
You need to edit an existing macro.

Solution:
To edit an existing macro in register a:

- Run `:put a` to output the macro steps.
- Edit or add steps to it.
- Run `"ay$` to yank it back to register a.

You can now execute the updated macro with `@a`.

For example, if you have a macro to uppercase the current line then go one line below in register a (`qagUUjq`), if you want to add to that macro a movement to the next word (`w`), do this:

1. Run `:put a` to paste the instruction string “gUUj”.
2. Add the needed action, “w” to the instruction string, making it: “gUUjw”.
3. RUN `"AY$` FROM THE START OF THAT LINE TO YANK IT BACK TO REGISTER A.
4. Now running `@a` will uppercase the current line, go down one line, and go to the next word.

An alternative is to edit it on the command-line mode.

- Type `:let @a =` (don’t press `<Enter>` yet!)
- Press `Ctrl-R a`. Vim will output the content of register a.
- You can now edit it.

Use the Blackhole Register to Prevent Register Pollution

Problem:
Some commands, like `:d`, save the affected texts into the register but you don’t want to store the affected texts into the register.

Solution:
Use the blackhole register _ to make changes without storing it to any register.

This is particularly useful if you are running a common on multiple rows and that command’s side effect saves to a register.

Instead of running `:g/donut/d` to delete all lines that contain “donut” and pollute the numbered registers, running `:g/donut/d _` deletes all lines that contain “donut” without polluting the numbered register.

See the Content From All Registers

Problem:
You want to see what your current registers are.

Solution:
To see the text from all registers, run `:registers`. You can also check for a specific register by passing the register name as an argument. Run `:register a` to see the content of the register a.

Execute a Macro Programmatically

Problem:
You need to execute a macro on multiple lines at the same time.

Solution:
You can run a macro programmatically using the `:normal` command.

To execute macro a on lines 5-10, run `:5,10 normal @a`.

You can also combine this with other commands, like the global command. To execute macro a on lines containing “const”, run `:g/const/normal @a`.

If you need to execute a macro on multiple files on the lines that contain the word “const”, you can run `:argdo g/const/normal @a`.

Quickly Append to an Existing Macro

Problem:
You want to add a few more instructions at the end of an existing macro.

Solution:
Use the uppercase letter of that register to append to an existing macro.

If you have the following macro to uppercase the current line then go down one line:

```
qagUUjq
```

Later you want to add `dd` after `j`. Instead of recording the macro from scratch, simply run `qAddq`. `qA` adds the action `dd` to the existing macro in register a. To put it in another word, Vim automatically adds the `dd` operation on top of the `gUUj` operations.

Quickly Paste From the Numbered Register

Problem:
You need to paste sequentially from the numbered registers (`"1P`, `"2P`, `"3P`).

Solution:
When you paste from the numbered register (`"1P`) followed by a dot command `.`, Vim automatically increments the numbered register.

Try this: paste from the numbered register 1 with `"1P`, then run `.` (this executes `"2P`), then run `.` again (this executes `"3P`).

Macro Factory

Problem:
You want to duplicate and combine macros.

Solution:
Recall that macro registers are really variables where you can store strings of instructions to be executed or texts to be outputted. Since macros are just texts, you can perform string operations on them.

To combine macros b and c and save in the register a, run `:let @a = @b . @c` (`.` in a Vimscript expression is a string concatenation; don't confuse it with the dot command). In fact, if you want to add an additional operation to a macro, like `dd`, you can do it with `:let @a = @b . @c . 'dd'`.

Recursive Macro

Problem:
You need to run a recursive macro.

Solution:
If you have the text:

```
a. chocolate donut
b. mochi donut
c. powdered sugar donut
d. plain donut
```

If you want to toggle the case of the first word using a recursive macro, run:

```
qaqqa0W~j@aq
```

Steps breakdown:

- `qaq` records an empty macro. It is imperative in a recursive macro to start with an empty register. When you recursively call that macro, it will run whatever is in that register.
- `qa` records in register a.
- `0W~j` goes to the first character in the current line, then goes to the next WORD, then toggles the case of the character under the cursor, then finally goes down one line.
- `@a` executes the macro a (this is the recursive part - you're still recording macro a and here you are also executing macro a).
- `q` stops recording.

Multi-File Operations

Different Ways to Execute a Command In Multiple Files

Problem:
You need to execute a Vim command across multiple files.

Solution:
Vim has 8 different means to execute commands across multiple files.

```
:argdo    arg list files
:bufdo    buffers
:windo    windows in the current tab
:tabdo    tabs
:cdo      each item in the quickfix list
:cfdo     each file in the quickfix list
:ldo      each item in the location list
:lfdo     each file in the location list
```

These commands require you to make a list in their respective categories (argument list, buffer list, window list, tab list, quickfix list, and location list) before you can run the command.

- To create an argument list, run `:args file1.txt file2.txt file3.txt`
- To create a buffer list, just open more files (like `:e file1.txt`)
- To create a window list, create more split windows (like `:vsplit file1.txt`)
- To create a tab list, create more tabs (like `:tabnew tab1`)
- To create a quickfix list, use a command that uses quickfix (like `:vimgrep /searchkeyword/ *.txt`)
- To create a location list, use a command that uses location list (like `:lvimgrep /searchkeyword/ *.txt`)

In this section, I'll use `:argdo`, but overall the flow is the same for all of them.

Substitute In Multiple Files

Problem:
You need to substitute across multiple files.

Solution:
To substitute and save across multiple files, run `:argdo %s/donut/pancake/g | update`

The `update` command is to save each file.

Execute a Macro in Multiple Files

Problem:
You need to run a macro across multiple files.

Solution:
To execute a macro in multiple files, you can use the `:normal` command. To execute macro in register a, run `:argdo normal @a | update`.

Limiting the Files to Operate On

Problem:
You only need to run the command on only the first 5 files.

Solution:
You can limit the number of files you want to operate on with a range when running the `-do` command.

To substitute only the first 5 items in the argument list: `:1,5argdo s/pancake/waffle/g | update`.

Creating an Argument List

Problem:
You need to quickly create an argument list.

Solution:
The argument list is needed to run the `:argdo` command.

- To get all the markdown files in the current directory, run `:args *.md`.
- To get all the markdown files in the current directory recursively, run `:args **/*.md`.

Once you get all the files, you can view them with `:args`.

Adding to an Argument List

Problem:
You already created an argument list and you need to add more items.

Solution:
Use `:arga` to add to the argument list.

Supposing you created an argument list of all markdown files that start with "a" by running `:args a*.md`, then later you also want to add a list of of all markdown files that start with "b". If you run `:args b*.md`, it will replace the first markdown list. To achieve this, run `:arga b*.md`. Now you have a list of markdown files that start with "a" and "b".

If you call `:arga` without argument, Vim will add the current buffer to the argument list.

Numbers

Quickly Increment or Decrement a Number on a Line

Problem:
You need to quickly increment or decrement a number on the current line.

Solution:
Vim has `Ctrl-A` and `Ctrl-X` operators to increment and decrement the next closest number (from the cursor) in the current line.

If you have the text:

```
dozens of donut: 2
calories per donut: 200
```

- If your cursor is on the first line and you press `Ctrl-A`, the cursor will automatically go to "2" and increment it to "3".
- If your cursor is on the second line and you press `Ctrl-X`, the cursor will automatically go to "200" and decrement it to "199".

Incrementing Number By More Than One

Problem:
The increment / decrement operators (`Ctrl-A` and `Ctrl-X`) only adds or subtracts one each time.

Solution:
You can pass a count before the increment / decrement operators.

If you run `10 Ctrl-A` at the start of the line with "I ate `2` donuts", it will increment it into "I ate `12` donuts". Now that is sweet!

Likewise if you run `5 Ctrl-X` on the same line, it will decrement it into "I ate `7` donuts".

Sequentially Increment Multiple Numbers

Problem:
You need to increment numbers on multiple rows, sequentially.

Solution:
If you use visual selection to highlight multiple rows and run `Ctrl-A`, Vim will increment into the same number across the rows.

However, if you run `g Ctrl-A` or `g Ctrl-X` in visual selection, it will sequentially increment the numbers on each line.

If you have the text:

```
breakfast0
breakfast0
breakfast0
```

If you visually highlight all 3 lines then press `g Ctrl-A`, Vim will increment each line:

```
breakfast1
breakfast2
breakfast3
```

Substitute and Increment Number

Problem:
You need to increment numbers using substitution.

Solution:
You can do this by utilizing the substitute command with the global.

Suppose that you have the following:

```
## I'm a header

## I'm another header

## I'm yet another header
```

And you want to substitute `##` into incrementing numbers, like:

```
1.)  I'm a header

2.)  I'm another header

3.)  I'm yet another header
```

Run:

```
let i = 1 | g/^##/s//\=i . '.)'/g | let i += 1
```

- `let i = 1` sets the variable `i` to equal `1`.
- `g/^##/` runs the global command that matches the `^##` pattern (lines starting with "`##`").
- `s//` to use the substitute command. If you leave the search pattern blank, Vim will use the same search pattern as the global command search pattern (which is `^##`).
- `\=i . '.)'/g` uses the expression `\=i . '.)'` as the substitute pattern (the variable `i` concatenated with the string ".)". The `g` is the global flag.
- `let i += 1` increments the variable `i` by `1` with each iteration.

Quickly Generate Incrementing Numbers Anywhere

Problem:
You need to generate a list with incrementing numbers.

Solution
If you need to create a list quickly, you can generate incrementing number prefixes on the fly. Here is how you can do this using visual mode and the number increment operator.

Suppose that you have this list:

```
first item
second item
third item
```

1. Put your cursor on the first character on the first line, the character "f" in "first",
2. Activate the block-wise visual mode (`Ctrl-V`),
3. Go down two lines.
4. Press `I` to activate insert mode before the cursor and type "`0.` ". Press `<Esc>`.

You should have:

```
0. first item
0. second item
0. third item
```

5. Press `gv` to quickly highlight the previously selected visual mode. Vim will automatically highlight all "0" texts.
6. Press `g Ctrl-A` to sequentially increment numbers and you're done!

```
1. first item
2. second item
3. third item
```

Get ASCII Value

Problem:
You need to get an ASCII value of a character.

Solution:
If you run `ga`, Vim will print the ASCII value of the character under the cursor (in decimal, hexadecimal, and octal).

Repeat

Repeat the Last Command-Line Command

Problem:
You need to repeat the last command-line command.

Solution:
Repeat the last command-line command with `@:`.

If you just ran `:s/donut/pancake`, you can repeat it with `@:`.

Repeat the Last Change

Problem:
You need to repeat the last change you made.

Solution:
You can repeat the last change with `.` (the dot command).

If you type "I love breakfast", you can insert that text again by pressing the dot command (`.`).

Repeat the Last Substitute

Problem:
You need to repeat the last substitute.

Solution:
Repeat the last substitute with `:&` or `&`. If you do `:s/donut/waffle/g`, then you run `&`, Vim will run `:s/donut/waffle` on that line. Vim doesn't remember the `g` flag though.

To repeat the last substitute and remember the flags, you can use `:&&`. If you do `:s/donut/waffle/g` and if you run `:&&`, Vim will run `:s/donut/waffle/g` on that line.

To repeat the last substitute on all lines with the same flags, use `g&`.

If you do `:s/donut/pancake/g`, then run `g&`, Vim runs `:%s/donut/pancake/g`. Technically Vim runs `:%s//~/&`, which consists of:

- `%s` to sub on all lines
- `//` to repeat the last pattern
- `~` to repeat the sub string
- `&` to use the last flag

Repeat the Last Executed Macro

Problem:
You need to repeat the last executed macro

Solution:
You can repeat the last executed macro with `@@`.

If you just executed macro a with `@a`, running `@@` will execute that macro again.

Repeat the Last External Command

Problem:
You need to repeat the last external command.

Solution:
You can repeat the last external-command execution (`:!{cmd}`) with `:!!`.

If you just ran `:!ls -l`, running `:!!` will re-run `:!ls -l`.

Sort

Sorting Lines

Problem:
You need to sort multiple lines alphabetically.

Solution:
Vim has the `:sort` command to sort multiple rows. A few things you can do:

- To sort the entire file, run `:sort`.
- To sort between lines 5-10, run `:5,10 sort`.

This command also accepts flags to sort rows in different manners.

If you have duplicate lines:

```
pancake
donut
waffle
donut
```

Run `:sort u` to sort them and to remove the duplicates.

Vim `:sort` does not sort numbers correctly. If you have:

```
2 donuts
1 donut
11 donut
10 donuts
```

Running `:sort` will give you:

```
1 donut
10 donuts
11 donut
2 donuts
```

To fix this, run `:sort n`.

To sort based on pattern with `:sort /pattern/`.

```
pancake, donut, waffle
sushi, pasta, taco
apple, banana, carrot
water, apple juice, milk
```

- If you want to sort based on the second item ("donut" vs "pasta", "banana", "apple juice"), you can use `:sort /,/`. Vim will use the characters after the match to sort.
- If you want to sort based on the third item ("waffle" vs "taco" vs "carrot" vs "milk"), you can use `:sort /,.\+,/`.

External Sort

Problem:
You need to use a different sorting program.

Solution:
Recall that you can use external commands as filters. Your terminal probably comes with a `sort` command. To sort using the terminal `sort` command, run `:%!sort`.

Note: when using an external command to filter texts in Vim, pass it a range.

- To sort the entire file, run `:%!sort`.
- To unique sort lines 5 to 10, run `:5,10!sort -u`.

Reverse Sort

Problem:
You need to do a reverse sort.

Solution:
You can do a reverse sort with `:sort!`.

You can also use the terminal `sort` command, `:%!sort -r`.

Substitute

Basic Substitution

Problem:
You need to substitute a pattern.

Solution:
The substitute command is a versatile Vim command and probably the one you'll use often. Its basic syntax is:

```
:s/pattern1/pattern2
```

To substitute "donut" with "pancake" in the current line, run `:s/donut/pancake`. Note that if you have multiple "donut" on the same line, Vim will only substitute the first one. To substitute multiple instances, pass to it the global (`g`) flag: `:s/donut/pancake/g`.

You can perform substitution in a range.

- To substitute "donut" with "pancake" between lines `2` and `5`: `:2,5s/donut/pancake/g`.
- To substitute "donut" with "pancake" on the entire buffer, run `:%s/donut/pancake/g`.

Case Insensitive Match

Problem:
You need to do a case insensitive substitution match.

Solution:
Use the `i` flag for a case insensitive match.

If you run `:%s/dOnuT/salad/gi`, it will substitute "donut", "dOnuT", and "DONUT" to "salad". This is useful if you're matching a phrase that has varying cases.

Asking For Confirmation Before Substituting

Problem:
You want Vim to ask for confirmation before substituting.

Solution:
Use the `c` flag for a confirmation before substituting.

If you are working in a large buffer and you need to substitute *some*, but not all of the matches, this is a perfect solution.

Suppose that you need to substitute "donuts" with "waffle", you can run `:%s/donut/waffle/gc`.

```
Chocolate donut
Strawberry donut
Vanilla donut
```

If you only need to substitute the first and third match, after running the substitute command, Vim will ask you for a confirmation on each match. Just say yes on the first one, no on the second one, and yes on the third one.

Group Match In Substitution

Problem:
You need to reverse the order of a pattern.

Solution:
You can group the match pattern and refer them as `\n` (where `n` is the integer).

If you want to substitute "chocolate donut" into "donut chocolate", you can just do `:s/chocolate donut/donut chocolate`.

But what if you need to make the second word the first and the first word second? You need a group match.

To turn "chocolate donut" into "donut chocolate", run `:s/\(.\+\) \(.\+\)/\2 \1/g`.

- The first group `\(.\+\)` matches "chocolate" and is represented by `\1`.
- The second group `\(.\+\)` matches "donut" and is represented by `\2`.

Another example: if you need to swap the 1st letter to 2nd, the 2nd letter to the 3rd, the 3rd to 1st, you can run `:s/^\(\w\)\(\w\)\(\w\)/\2\3\1/g`. This will swap "123" to "231", "abc" to "bca".

Use Very Magic to Avoid Escaping Special Characters

Problem:
You are sick and tired of escaping special characters.

Solution:
Some characters, when doing search / substitution, have special meanings (like `.`, `( )`, or `+`). There are times when you have to escape a plethora of characters. Use the very magic flag `\v` to avoid escaping special characters.

- To turn "chocolate donut" into "donut chocolate" with very magic, run `:s/\v(.+) (.+)/\2 \1/g` (contrast that with `:s/\(.\+\) \(.\+\)/\2 \1/g`).
- To swap the 1st letter to 2nd, 2nd to 3rd, and 3rd to 1st with very magic, run `:s/\v^(\w)(\w)(\w)/\2\3\1/g` (contrast that with `:s/^\(\w\)\(\w\)\(\w\)/\2\3\1/g`).

Representing the Entire Match

Problem:
You need a way to represent "the entire match" pattern.

Solution:
In a group match, `\0` is a special keyword that represents the entire match.

If you have the string "pancake" and you want to update "pancake" into "fluffy pancake", instead of running `:s/pancake/fluffy pancake`, you can run `:s/pancake/fluffy \0/`. `\0` represents the entire match, which is the word "pancake".

If you need to add parentheses around "pancake", you can run `:s/pancake/(\0)/g` to give you "(pancake)".

Alternatively, `&` is also another keyword for the entire match: `:s/pancake/fluffy &/` and `:s/pancake/(&)/g`.

Using the Group Match to Remove Words

Problem:
You need to reuse parts of the search pattern.

Solution:
You can use group matches, `\1 \2 \3 etc`, to remove words.

You can delete subsequent duplicate lines with `:%s/(.+)\n\1/\1/g`. Here you are using `\1` as a pattern to substitute and pattern to substitute into. If you have:

```
chocolate
donut
donut
```

Vim deletes the second "donut".

Another example, if you have a list of phone numbers:

```
123-123-1234
111-111-1111
```

If you want to get rid of the area code (the first 3 digits), you can run `:%s/\v(\d{3}-)(\d{3}-)(\d{4})/\2\3`. You'll get:

```
123-1234
111-1111
```

Reusing the Previous Search Pattern

Problem:
You just searched for a word (`/some keyword`) and you don't want to retype that word again when doing substitution.

Solution:
If you use an empty substitute search pattern, Vim will use the previous search pattern!

If you recently searched for `/donut` and then you run `:%s//pancake/g`, this is equal to `:%s/donut/pancake/g`. Why type more if you can type less?

Deleting In Substitution Quicker

Problem:
When you're using a substitution to delete, like in `:s/donut//`, is there a faster way to do this?

Solution:
If you leave the substitute pattern blank, Vim assumes that you want to delete them:

- `:%s/waffles` removes all "waffles" in a buffer.
- `:%s/^\n` removes all empty lines in a buffer.

Using an Expression in a Substitution

Problem:
You need to do a complicated substitution using an expression

Solution:
You can use the expression register in substitution with `\={expr}`.

To substitute all `__DATE__` string with today's date, you can run `:%s/__DATE__/\=strftime("%c")/g`. `strftime()` is a Vimscript function (`:h strftime()`).

Removing Trailing Whitespaces

Problem:
You have trailing whitespaces all over the buffer.

Solution:
To remove trailing whitespaces, run `:%s/\s\+$//g`.

- `\s` represents whitespace characters (literal spaces and tabs).
- `\+` means one or more preceding character.
- `$` here means the end of the line.

Together, `\s\+$` means one or more whitespace characters at the end of the line. Recall that the `:s` command automatically deletes if you leave the second half blank, so `:%s/\s\+$` works too.

Add a New Line In Substitution

Problem:
You need to add a new line to a phrase.

Solution:
Substitution can also be used to add new lines. Use the `\r` pattern.

If you want to add a new line after each line that starts with "pancakes", run `:%s/^pancakes.*/\0\r/g`:

Repeat the Last Substitute Command

Problem:
You need to repeat the last substitute command.

Solution:
`:s` repeats the last substitute command without the flag.

If you recently did `:s/pancake/donut` and you repeat it `:s` will quickly repeat that. Alternatively, you can also press `&`. `&` is the normal mode version of `:s`.

If you recently did `:s/pancake/donut/g` and you want to keep the `g` flag, run `:&&`.

Capitalize the First Letter Of Each Word

Problem:
You need to capitalize the first letter of each word in a line.

Solution:
Run `:s/\<./\u\0/g` to capitalize the first letter of each word in a line.

- `\<.` is the pattern for the beginning of a word.
- `\u` is a special keyword modifier to uppercase the subsequent pattern.
- `\0` represents the entire match.

Perform a Chain of Substitutes

Problem:
You need to substitute a few phrases simultaneously.

Solution:
You can chain multiple substitutions with `|`.

To substitute "good" to "awesome", "bad" to "terrible", and "ugly" to "hideous", run:

```
:s/good/awesome/ | s/bad/terrible/ | s/ugly/hideous/
```

Repeat the Last Substitute String

Problem:
You need to do a different substitution but with the same substitute string as your previous substitution.

Solution:
Use the ~ pattern in substitution to repeat the last substitute string.

If you recently ran `:s/pancake/donut/g`, then `:s/waffle/~/g`, Vim will also substitute "waffle" with "donut", the last substitute string.

Repeat the Last Substitution With the Last Search Pattern and the Last Substitute String

Problem:
You just searched with `/`. You just performed a substitution. Now you need to substitute using the last search pattern but with the same previous substitute string.

Solution:
Use `:~` to repeat the last substitution using the last search pattern and the last substitute string

If you recently ran `:s/pancake/donut/` and you recently searched `/mochi`, running `:~` will do `:s/mochi/donut/`.

Changing the Delimiter

Problem:
You got lost in the jungle of front-slashes.

Solution:
You need to substitute a web URL with a long path like `https://mysite.com/a/b/c/d/e` and you end up with something like `:s/https:\/\/mysite.com\/a\/b\/c\/d\/e/hello/`.

Frankly it is hard to tell which forward slashes are part of the substitution patterns and which ones are the delimiters. Vim lets you change the substitution delimiter with any single-byte characters (except for alphabets, numbers, `"`, `|`, and `\`). What that means is that you can do something like:

```
:s+https:\/\/mysite.com\/a\/b\/c\/d\/e+hello+
```

or

```
:s@https:\/\/mysite.com\/a\/b\/c\/d\/e@hello@
```

With this, the distinction between `/` as the literal character and the delimiters are clearer.

Global Command

Global Command Basics

Problem:
You don't know how to use the global command.

Solution:
Global command is a useful command to execute a command on multiple lines. The general pattern is `:g/pattern/command`.

For example, to delete all lines containing "let", you can use `:g/let/d`.

Inverse Match

Problem:
You want to run the global command on non-matching lines instead of matching lines.

Solution:
The command `:g/let/d` deletes all lines containing the word "let". What if you want to delete all lines not containing "let"?

You can use the inverse global command with `g!` or `v`. Run either `:g!/let/d` or `:v/let/d`.

Prepend or Append On Multiple Lines

Problem:
You need to append a semicolon on all lines containing "const".

Solution:
From normal mode, you can enter insert mode at the end of the line with `A`. You can take replicate normal mode commands with the `normal` command.

To add a semicolon at the end of all lines containing "const", run `:g/const/normal A;`.

By the way, recall that you can enter insert mode at the start of the line with `I`. To add "const" at the start of all lines containing ";", run `:g/;/normal Iconst `.

Global Command Within a Range

Problem:
You need to execute the global command within a range, not on the entire file.

Solution:
You can run `:5,10g/donut/s/pancakes/waffles/g` to execute the global command between lines 5 and 10.
You can also run the global command between a marked range. The command `:'a,'bg/donut/s/pancakes/waffles/g` executes the global command between the marks `'a` and `'b`.

Delete Blank and Empty Lines

Problem;
You need to quickly get rid of empty lines.

Solution:
The global command is great to for deleting multiple empty and blank lines lines.

- Run `:g/^$/d` to delete all empty lines.
- Run `:g/^\s*$/d` to delete all blank lines.

Running Global Command Between Matching Patterns

Problem:
You need to execute the global command only between `pattern1` and `pattern2`.

Solution:
The global command accepts the form of `:g/pattern1/,/pattern2/ {cmd}` to execute `cmd` between `pattern1` and `pattern2`.

If you want to delete the text between "apples" and "oranges", excluding "apples" and "oranges", run `:g/apples/+, /oranges/- d`. If you want to delete "apples" and "oranges" too, run `:g/apples/, /oranges/ d`.

```
apples
bananas
oranges

apples
blueberries
pineapple
mangos
oranges

papayas
```

Explanation:

- `pattern1` is `apples`.
- `pattern2` is `oranges`.
- The `+` means one line below and `-` means one line above.
- `d` is the delete command.

If you want to sort only the items inside the parentheses, run `:g/(/+1,/)/-1 sort`.

```
(
  chokeberry
  blueberry
  acaiberry
)
  apples
  carrots
  bananas
)
```

Explanation:

- `pattern1` is `(` (left parentheses).
- `pattern2` is `)` (right parentheses).
- The `+1` in `/(/+1` and the `-1` in `/)/-1` means that the global command will affect texts after `(` and before `)`.
- `sort` is the command to execute.

Condense Multiple Empty Lines

Problem:
You have multiple empty lines in a file and you need to wrap them into single lines.

Solution:
You can condense multiple empty lines with `:g/^$/,/./-1 j`.

If you have:

```
Hello

There are

many empty
lines


in here
```

Running `:g/^$/,/./-1 j` will condense the empty lines into single lines. This global command uses two patterns: `/^$/` (empty line) and `/./-1` (one line before a non-empty line). The command to be executed is the join command (`:j`).

Alternatively, if you have the `cat` program in the terminal, you can also run `:%!cat -s` (the `-s` stands for "squeeze") to squeeze multiple empty lines into single lines.

Reuse the Global Command Search Pattern in the Substitute Command

Problem:
You want to reuse the global command pattern in the substitute command.

Solution:
Vim will actually re-use the global command pattern if it is followed-up with the substitute command with a blank search pattern.

`:g/apples/s//oranges/g` is the same as `:g/apples/s/apples/oranges/g`.

The Default Command

Problem:
You want to print the global command results.

Solution:
If you don't pass any `{cmd}` to the global command, Vim will automatically run the print command (`:p`).

`:g/bagel` is the same as `:g/bagel/p`.

Note that this command has a pattern of `g/re/p`. This is the same grep command found in the terminal.

This pattern, `g/re/p`, originally comes from the Ed editor (remember that?).

Use Execute to Run Complex Expressions

Problem:
You want to run the global command using a complex expression.

Solution:
You can use Vim's `:execute` command with the global command.

Given a list of filenames (without any extension):

```
file1
file2
file3
```

To generate new markdown files for each line. Run `:g/^/execute "w " . getline(".") . ".md"`.

The command executes `:w file1.md`, `:w file2.md`, and `:w file3.md`. Vim will generate `file1.md`, `file2.md`, and `file3.md`. Neat!

Reverse All Lines

Problem:
You need to reverse the order of each line.

Solution:
To reverse all lines, just run `:g/^/m 0`.

- `^` will target all lines in a file.
- `m 0` is a move command. It will move each line, into the 1st line, effectively reversing the entire order.

Surround All Digits With Double Quotes

Problem:
You need to surround any digits with double quotes.

Solution:
Given the following text:

```
const one = 1
const two = 2
const three = 3
const nope = "don't mind me"
const four = 4
const five = 55
```

Run `:g/\d\+/s//"&"/g` to get:

```
const one = "1"
const two = "2"
const three = "3"
const nope = "don't mind me"
const four = "4"
const five = "5"
```

Explanation:

- The pattern `\d\+` matches one or more digits.
- `s//` is a substitute command with a blank substitute pattern. Recall that if you leave the search pattern blank, Vim will use the pattern used by the global command.
- `"&"` is the substitute pattern. Recall that `&` represents the entire match. `"&"` surrounds the entire match with double quotes.

Changing the Delimiter

Problem:
You got lost in the sea of `/`.

Solution:
Just like you can change the delimiter of the substitute command, you can also change the delimiter of the global command with any single-byte characters.

This is a valid global command:

```
:g@console@d
```

If you are using a global command with the substitute command, you can do something like this:

```
g@one@s+const+let+g
```

It is easier to see which part belongs to the global command and which belongs to the substitute command.

Programmatic Global Command

Problem:
You need to programmatically run the global command.

Solution:
The global command is very versatile. You can enhance the global command with Vimscript.

A few things of you can do:

- Delete even lines with `:g/^/if line(".") % 2 != 0 | norm! d$`
- Delete lines that are less than 3 char long with `:g/^/if strlen(getline(".")) < 3 | d`

Tags

What Are Tags?

Problem:
You want to understand about Vim's tag system.

Solution:
In many modern editors, you can click on a function or class definition and it will take you to where that function or class is defined. You can do that in Vim with tags. Although you can just search for any definition with `grep`, `vimgrep`, or external libraries like (`fzf.vim`), tags are the best tool for this kind of job.

Think of a Vim tag like an address book:

```
Name    Address
Iggy1   123 Awesome St, 12345
Iggy2   987 Incredible Ave, 98765
Iggy3   555 Handsome Rd, 55555
```

To look for `Iggy1`'s address, first you open the address book, then search for `Iggy1` on the `Name` column, then search for the corresponding address. Tags work the same way. Instead of having name-address table, Vim tags store word definitions and their locations in the project.

So how do you get started with tags?

You need a tool called tag generator to generate this address file. Modern Vim doesn't come with a tag generator so you have to download it separately. There are several options you can choose, but the one I recommend is either Universal Ctags[3] or Exuberant Ctags[4]. Follow the direction on their websites how to install it.

Here I'll use the Universal Ctags. After you install it, you should get the `ctags` command. If you use a different tags generator, you can still follow along.

Assume that you have two ruby files:

[3]https://github.com/universal-ctags/ctags
[4]http://ctags.sourceforge.net

```
## one.rb
class One
  def initialize
    puts "Initialized"
  end

  def donut
    puts "Bar"
  end
end


## two.rb
require './one'

one = One.new
one.donut
```

Running `ctags -R .` from the terminal will generate a basic tag file (named `tags`). The `-R` means a recursive call.

Inside that file:

```
One	one.rb	/^class One$/;"	c
donut	one.rb	/^  def donut$/;"	f	class:One
initialize	one.rb	/^  def initialize$/;"	f	class:One
```

Yours may have additional rows, but it should at least contain a word definition and its address.

If you look at one of them:

```
donut	one.rb	/^  def donut$/;"	f	class:One
```

`donut` is the tag name. When your cursor is on “donut”, Vim searches the tag file for a row that contains “donut”.

`one.rb /^  def donut$/;"` is the address. It consists of two parts:

- `one.rb` is the file name. Vim looks for a file `one.rb`.
- `/^  def donut$/` is the address. `/.../` is a pattern indicator. `^` is a pattern for the first element on a line. It is followed by two spaces, then the string `def donut`. Finally, `$` is a pattern for the last element on a line. This is the same as if you’re searching with `/^  def donut$`.
- `f class:One` is the tag option that tells Vim that “donut” is a function (f) and is part of the `One` class.

In short, a tag file is an address book that tells Vim where to find a particular definition.

Setting Up the Tag File

Problem:
You successfully created a tag file but you don't know how to set it up.

Solution:
Run `:set tags?`, Vim will display `tags=./tags,tags` (depending on your Vim settings, it might be a little different). Vim looks at the `tags` option for locations where it can find the tag file.

You can add or remove path locations. If you prefer to have a centralized tags for your projects, you can do something like `set tags+=/path/to/your/tags`. Vim will search for tags definition in order. If Vim finds a tag file in `./tags` (the current directory) it will use that and not the other tag files. If there is no tag file in the current directory, Vim will check the next path, `tags` (the project root file). Finally, it will look for tags in `/path/to/your/tags`.

Tags Navigation

Problem:
You want to quickly jump to a method definition.

Solution:
If your cursor is on "donut", you can jump to "donut" origin by pressing `Ctrl-]` or by running `:tag donut` (assuming that you already have a tag file properly set up in the correct path)

What if you have multiple methods with the same name? Vim will jump to the definition with a higher tag priority.

Wait, what is tag priority? When a tag file contains duplicate item names, Vim decides which address to jump to based on the tag priority.

The priorities are, in order:

1. A fully matched static tag in the current file.
2. A fully matched global tag in the current file.
3. A fully matched global tag in a different file.
4. A fully matched static tag in another file.
5. A case-insensitively matched static tag in the current file.
6. A case-insensitively matched global tag in the current file.
7. A case-insensitively matched global tag in a different file.
8. A case-insensitively matched static tag in the current file.

Selective Jump

Problem:
You want to choose which keyword to jump to, regardless of tag priority.

Solution:
Suppose that you have two "pancake" keyword definitions. If you press `Ctrl-]` while your cursor is on the word "pancake", Vim decides it for you based on its tag priority. But what if you want to jump to the one with less priority? How can you fight back Vim's dictatorship?

With `g Ctrl-]`, if Vim finds multiple matches, you can choose which definition to jump. The command-line alternative is the command `:tjump pancake`.

Autocompletion

Problem:
You want to use tags for autocompletion.

Solution:
You can use autocompletion in insert mode using tags. Just press `Ctrl-X Ctrl-]` and Vim will autocomplete according to the tag file.

Automatic Tag Generation

Problem:
You don't want to keep running the tags generator manually.

Solution:
Your code changes over time. You add, update, and remove methods and classes. Meanwhile, the tag file does not know about these changes so it keeps the old definitions.

Ideally you need to re-run `ctags -R .` each time you make a significant change, but that takes a ton of work! The good news is, you can use Vim autocommand (`:autocmd`) to generate tags automatically.

For example, if you want to automatically generate tags each time you save a ruby file, run:

```
:autocmd BufWritePost *.rb silent !ctags -R .
```

Another alternative is to use external plugins to manage the tags. Some popular tag plugins are (the list is not comprehensive, so check out if there are better ones to suit your coding style):

- vim-gutentags[5]

[5]https://github.com/ludovicchabant/vim-gutentags

- vim-tags[6]
- vim-easytags[7]
- vim-autotag[8]

[6]https://github.com/szw/vim-tags
[7]https://github.com/xolox/vim-easytags
[8]https://github.com/craigemery/vim-autotag

Text Generation

Get a File Content

Problem:
You need to get the content of another file into the current file.

Solution:
You can read the content of a file with `:r myfile.txt`. By default it will put the content below the current line. If you want to have it on the first line, run `:0r myfile.txt`.

Get the Current Date

Problem:
You need to get the current date quickly.

Solution:
Your terminal probably has a `date` command. Recall that you can insert from a file with `:r somefile.txt`. You can get the current date with `:r !date`.

This command also works with other external commands:

- If you need to get all the filenames in the current directory, run `:r !ls`.
- If you need to get the response of a curl command, run `:r !curl -s https://jsonplaceholder.typicode.com/posts` (by the way, the `-s` flag in this command stands for silent).

Generate Numbers

Problem:
You need to quickly generate numbers from 1 to 10.

Solution:
Run `:put =range(1,10)` to generate the numbers 1 to 10.

The `put` method outputs the text from the given register. In this case, = is the expression register. The expression is `range(1,10)`.

By the way, the `range()` command also accepts an optional third parameter, stride. Run `:put=range(1,10,2)` to generate numbers 1 to 10 incremented by 2. Run `:put=range(10,1,-1)` to generate decrementing numbers.

Generate IP Addresses

Problem:
You need to generate different IP addresses.

Solution:
To generate a sequential IP addresses, use `:for i in range(1,10) | put ='192.168.0.'.i | endfor`.

This is an expansion of the `put` + `range` method above using the `for` loop and string concatenation.

Generate Random Dice Throw

Problem:
You want to generate a random number between 1 to 6.

Solution:
To generate 10 different dice rolls, you can use `rand()`:

```
:for i in range(1,10) | put='Roll the dice: ' . (rand() % 6 + 1) | endfor
```

Generate Numbers Horizontally

Problem:
You need to generate a number series on the same line.

Solution:
To generate numbers horizontally, use `put` and `join`.

- `:put=join([1,2,3,4], \" - \")` to generate 1 - 2 - 3 - 4.
- `:put=join(range(1,10), \"', '\")` to generate 1', '2', '3', '4', … '10 (it won't generate the first and last single-quote, you can just add them manually).

Transform a Text Into a Numbered List Based On Line Numbers

Problem:
You want to turn a text into a numbered list based on line numbers.

Solution:
To create a numbered list on multiple lines, run `:%s/^/\=printf('%-2s', line('.'))`.

If you have following text starting on line 1:

```
Breakfast is good
But lunch is better
Dinner is still the best
```

Run `:%s/^/\=printf("%-2s", line("."))` to get:

```
1 Breakfast is good
2 But lunch is better
3 Dinner is still the best
```

Command breakdown:

- `^` represents the start of the line.
- `\=` allows you to use Vimscript expression in a substitution.
- `printf('%-2s')` prints the string with space (the `-2` is the field with, padded left).
- `line('.')` is the current line number.

Text Manipulation

How to Speak Vim

Problem:
You are having trouble remembering myriads of Vim commands.

Solution:
Don't try to remember every command. Instead, break it down into smaller chunks. Vim commands contain a grammar-like pattern (let's call it the "Vim Grammar") to create compositions.

There is one rule in Vim grammar:

```
verb + noun
```

Operators are verbs and motions are nouns. Most actions are done by following that rule. You only need to master a handful of operator and navigation keys to become productive.

For example:

- To delete the next word, run `dw`; verb (`d`) + noun (`w`).
- To yank two characters to the left, `y2h`; verb (`y`) + noun (`2h`). Note that you can pass a count before the motion (`2h`).
- To change the next line, `cj`; verb (`c`) + noun (`j`).

Texts often come structured. There is a concept of a "word", a "paragraph", a "parentheses", etc in a file. Vim has an abstraction known as text objects to address these structures. There are two different text objects: inner (`i` + object) and outer (`a` + object) text objects. This looks complicated, but once you get it, it's fairly simple and intuitive.

A text object is the noun in the Vim grammar rule of verb + noun. You need to pass an operator before a text object.

```
const hello = function() {
  console.log("Hello Vim");
  return true;
}
```

If your cursor is on the "H" in "Hello":

- To delete the entire "Hello Vim": `di(`.
- To delete the function block (surrounded by {}): `di{`.
- To delete the "Hello" string: `diw`.

If you are new to Vim, you don't have to remember dozens of operators. You can start by just committing these 3 to memory:

- `d` (delete).
- `c` (change).
- `y` (yank).

Changing Cases

Problem:
You need to change the case of your text.

Solution:
There are different ways to change the case of your text.

To change the case of a single character:

- ~ toggles the case of the character the cursor is on. If your cursor is on "d" in "donut", pressing ~ toggles it to uppercase.
- `vu` lowers the case of the character the cursor is on. If your cursor is on "D" in "Donut", pressing `vu` lowers the case to "d".
- `vU` uppercases the case of the character the cursor is on. If your cursor is on "d" in "donut", pressing `vU` ups the case to "D".

To change the case using a motion:

- `g~ {motion}` toggles the case of the texts indicated by motion. When combined with `g`, `g~` acts as an operator. To toggle the case of the next two words, run `g~2w`. To toggle the case of the current inner word, run `g~iw`.
- `gu {motion}` lowers the case of the texts indicated by motion. To lowercase the text inside the `()`, run `gui(`.
- `gU {motion}` ups the case of the texts indicated by motion. To uppercase to the end of the file, run `gUG`.

To change the case of the current line:

- To togglecase the current line, run `g~~`.
- To lowercase the current line, run `guu`.
- To uppercase the current line, run `gUU`.

Swap Two Characters

Problem:
You need to quickly swap two adjacent characters.

Solution:
I do these kinds of typos all the time when editing: typing "teh" instead of "the", "cosnt" instead of "const", etc. The easiest way to fix a two-character typo is to use `xp`.

`x` deletes the character under the cursor and `p` pastes that recently deleted character.

Replace Mode

Problem:
You need to replace characters with the ones you are typing.

Solution:
Vim has a replace mode that you can access with `R`. When in the Replace mode, Vim replaces the existing character with each character you type.

You can also use the virtual replace mode with `gR`. This command is similar to `R`, except when you press `<Tab>`, it will replace multiple characters (as opposed to only one character in replace mode).

Using a Command-Line Command As a Motion to an Operator

Problem:
There is not really a problem, but it's neat to know.

Solution:
In Vim, you can use `:` as a motion to an operator. In the context of Vim grammar verb + noun, the `:cmd` can act as a noun for a verb.

- To delete from the current position to the first match of "b": `d:call search("b")`.
- To delete from the current position to line 10: `d:10`.

The examples above look like a lot of work just to do a simple task and you're probably wondering when you will ever need this. The truth is, I haven't spent a lot of time on this revelation.

However, if you have an arsenal of Vimscript functions that can move your cursor based on a complex pattern, like `d:call FindWordsIn("Spanish")`, `d:call FindEnglishAdjectives()`, etc, this can be a useful feature. Also keep in mind that you can use it with any terminal command (`d:!{some_command}`).Maybe someone will figure out a way to use this effectively.

Force Motion

Problem:
You need to run an operator on columns.

Solution:
You can force a motion to be line-wise, character-wise, or block-wise, like the visual mode.

In this case, if you need to run the delete operator on the three columns containing `0`s:

```
br0eakfast
lu0nch
di0nner
```

With your cursor on the “`0`” in the first line, run `dCtrl-V2j` (note that this is similar to block-wise Visual mode).

Persistent Undo

Problem:
Each time you close and re-open Vim, you wish that you can undo the last action you did right before you closed Vim.

Solution:
An undo file allows you to preserve the undo history even after you close Vim; you can create an undo file with `:wundo` and load it with `:rundo`

Edit `somefile.txt`, then create an undo file with `:wundo! somefile.undo` (you can give it any name), then close Vim and reopen `somefile.txt`. Run `:rundo somefile.undo` to load the undo file. You can now undo from that file as if you never exited Vim.

Undoing In Chunks

Problem:
After you typed a long paragraph, when you undo, Vim removes the entire paragraph even though you only need to undo the last sentence.

Solution:
Vim undo undoes the last “change”. All the texts that you typed while you’re in the insert command count as a single change. You can exit the insert mode after each sentence to create smaller changes.

Alternatively, you can create an undo chunk with `Ctrl-G u` to break your change into smaller parts without leaving insert mode. Here’s how.

If you type `ipancake breakfast<Esc>` then you press `u`, Vim will undo the entire "pancake breakfast". To add a break-point, run `ipancake <Ctrl-G u>breakfast<Esc>`. Now when you press `u`, Vim will remove only "breakfast".

Join Lines

Problem:
You need to join the next line to the current line.

Solution:
If you press `J`, Vim will join the line below the current line to the current line. Vim automatically adds an extra space when joining two lines, so if you don't want the extra line, use `gJ` to join two lines without a space. If you are on visual mode, you can join all highlighted lines with `J`.

The `:j` (short for `:join`) is the command-line version of `J`. By running just `:j`, Vim will act like `J`. If you want to join all lines, run `:%j`.

Using Terminal Commands

Problem:
You need to filter your texts using the terminal commands

Solution:
Vim has a bang (`!`) operator to filter the text through the external command.

For example, if you have the following text that you need to tabularize and remove the rows without "Ok":

```
Id|Name|Cuteness
01|Puppy|Very
02|Kitten|Ok
03|Bunny|Ok
```

You can achieve this using the `column` terminal command and the `awk` command. Just run:

```
!}column -t -s "|" | awk 'NR > 1 && /Ok/ {print $0}'
```

You'll get a clean, filtered table:

```
02  Kitten  Ok
03  Bunny   Ok
```

This is one of Vim's shining features because it lets you extend Vim's capability to use external commands in addition to Vim's internal commands, allowing you to compose more complex actions.

Terminal

Vi Mode

Problem:
You want to use Vi keybindings in the terminal.

Solution:
To set the Vi mode in the terminal, run `set -o vi` (not supported by all terminals, but most terminals should have it).

Now when you press `<Esc>`, the you'll enter Vi Normal mode. Some available features are:

- Line navigations (`0`, `$`, `w`, etc).
- Go through history with `j` and `k`.
- Search the command history with `?some command` or `/other command` and follow-up with `n` and `N`.
- Etc.

Opening Files

Problem:
You want to quickly open multiple files.

Solution:
You can open multiple files at once with `vim file1.txt file1.txt file3.txt`.

You can use glob to open multiple files in Vim :

- Run `vim *.js` to open all Javascript files in the current directory.
- Run `vim **/*.js` to open all Javascript files recursively.

Diffing With Vim

Problem:
You need to perform a quick diff between two or more files.

Solution:
Run `vimdiff file1.txt file2.txt` to perform diff with Vim. Alternatively, you can also run `vim -d file1.txt file2.txt`.

Running Vim Without Plugins or Vimrc

Problem:
You need to run Vim without plugins.

Solution:

- To launch Vim without plugins but with vimrc, run `vim --noplugin`
- To launch Vim with plugins but without vimrc, run `vim -u NORC`
- To launch Vim without plugins and vimrc, run `vim -u NONE`

These are useful when you need to debug Vim.

Open Files in Vim as a Result of Another Command

Problem:
You need to pass the output of terminal commands to open files in Vim.

Solution:
You can run `vim` followed by `$(command)`. Some use cases:

- You can open all your `git status` files, run `vim $(git status --porcelain | awk '{print $2}')`.
- You can open the result of the `find` command with `vim $(find . -name "my_file*.js)`.

Open Files in Read-Only Mode

Problem:
You need to open files without modifying it.

Solution:
`vim -R filename.txt` opens the file in read-only mode. This is a good alternative for `less`. You get the benefit of Vim navigation without worrying that you may accidentally edit it.

Open Vim With Vertical and Horizontal Splits

Problem:
You want to open multiple files in Vim with horizontal or vertical splits.

Solution:

- You can open Vim with multiple vertical splits with `vim -O firstfile.txt secondfile.txt`
- You can open Vim with multiple horizontal splits with `vim -o firstfile.txt secondfile.txt`

You can also open Vim with multiple horizontal windows with `vim -on` and horizontal windows with `vim -On`, where n is the number of windows.

- To open Vim with 2 horizontal windows, run `vim -o2`.
- To open Vim with 3 vertical windows, run `vim -O5`.

Generate New Files Programmatically

Problem:
You want to generate new files programmatically in Vim.

Solution:
You can programmatically generate multiple new buffers with `vim hello{1..10}.txt`.

Passing Text as STDIN to a Command

Problem:
You are given a file from which you need to execute terminal commands from

Solution:
You can utilize Vim to execute terminal commands with `:%w !sh`. Vim will use the text in the buffer and pass it as a STDIN.

If you have the following text:

```
file1.txt
file2.txt
file3.txt
```

- Run `:%s/\v(.+)(\.txt)/mv & \1\.md/g` to change each line from "testN.txt" into "mv testN.txt testN.md".
- Run `:%w !sh` to execute the instruction.

This effectively executes `mv file1.txt file1.md`, `mv file2.txt file2.md`, and `mv file3.txt file3.md`.

Using the Terminal From Vim

Problem:
Most modern editors have a built-in terminal. You want that too.

Solution:
Vim (8.1 and up) has a terminal-mode that you can access with the `:terminal` command.

For more, check out `:h terminal.txt`.

Check Available Vim Features

Problem:
You want to see what features are supported in your current Vim build.

Solution:
You can run `vim --version` to see what versions are available. Likewise, you can also run `:version` from inside Vim.

Visual Mode

Visual Mode Basics

Problem:
You don't know how to use Vim's visual mode.

Solution:
Vim's visual mode is similar to highlights in most text editors (you highlight a body of text, then you apply changes to the highlighted texts).

There are three different visual modes:

- `v` - character-wise visual mode
- `V` - line-wise visual mode
- `Ctrl-V` block-wise visual mode.

Try them out! Use each of the different visual modes to get a feel. You can use Vim navigation to expand or contract the highlighted area. To exit the visual mode, press `<Esc>`.

You can also perform an operation to the highlighted body of text. For example, if you press `d` while having a body of text highlighted, that body of text will be deleted.

Insert Text on Multiple Lines

Problem:
You need to insert text on multiple lines.

Solution:
You can insert text on multiple lines with the block-wise visual mode (`Ctrl-V`).

If you have:

```
strawberry
chocolate
sugar
```

Once you've highlighted multiple columns with the block-wise visual mode, you can use either `I` to insert text before the cursor or `A` to insert text after the cursor.

- To insert the word “sweet” at the start of each line, use the visual block-wise selection (`<Ctrl-V>` and down) to highlight all 3 lines, then type `0Isweet <Esc>`.
- To insert the word “donut” at the end of each line, use the visual block-wise selection (`<Ctrl-V>` and down) to highlight all 3 lines, then type `$A donut<Esc>`.

Quickly Reselect the Previous Visual Highlight

Problem:
You need to re-select the last visual highlight.

Solution:
Run `gv` to visually highlight the previous visual selection.

If you highlight a body of text with either `v`, `V`, or `Ctrl-V`, then you leave the Visual mode, the next time you press `gv`, Vim will re-select the same block of text. Use this when you forget to apply an additional operation to the previously highlighted test.

Expanding Visual Highlight Bidirectionally

Problem:
You need to expand your visual highlight in any direction.

Solution:
Pressing `o` or `O` while in visual highlight moves the cursor to the other end of the visual highlight.

Suppose that you start highlighting the text below you, then you realize that you also need to highlight the text above you. If you move up, it will shrink the visual selection. To expand the highlight to the text above, first change the cursor location.

To change the cursor location, press either `o` or `O`. With your cursor now on the opposite location, you can now expand to highlight the text above you as well.

Switching to a Different Visual Mode While in a Visual Mode

Problem:
You’re currently in the character-wise visual mode but you need to switch to line-wise visual mode.

Solution:
While in a visual mode, you can switch to a different visual mode by pressing another visual mode command.

- If you're in the character-wise visual mode (`v`), pressing line-wise visual mode command (`V`) switches it to the line-wise visual mode.
- If you're in the line-wise visual mode (`V`), pressing the block-wise visual mode command (`Ctrl-V`) switches it to the block-wise visual mode.
- If you're in the block-wise visual mode (`Ctrl-V`), pressing the same visual mode command, in this case block-wise visual mode command (`Ctrl-V`), will exit the visual mode.

Visually Highlight the Last Search Term

Problem:
You recently searched with `/donut` and you realized that you needed to jump back to that word and modify it.

Solution:
Press `gn` to jump to the next last search term (similar to pressing `n` after `/hello`) and automatically highlights that search term.

Suppose that you have the following text:

```
one donut
two donut
three donut
four donut
```

If you had just searched for `/donut`, if you press `gn`, Vim will jump to the next "donut" phrase and do a visual mode highlight.

This is also useful when combined with the dot command (`.`). Press `gn`. Vim will highlight the next "donut". Then press `d` to delete it. To delete subsequent donuts, just run `n` (next match) and `.`. You can now repeat `n . n . n .` etc.

Replace Multiple Characters With Visual Selection

Problem:
The replace operator `r` only replaces one character at a time.

Solution:
When you use `r{something}` on a visual highlight, Vim will replace all of the highlighted texts at once.

In markdown, === represents a header. You can quickly generate a header on the following text with this method:

```
My awesome title
```

- Yank the line with `yy`.
- Paste it with `p`.
- Visually highlight the pasted text with `V`.
- Replace the selection with "=" by running `r=`.

`yypVr=`. That's all you need to create:

```
My awesome title
================
```

Vimrc

Quick Access to Vimrc

Problem:
You need to quickly access vimrc.

Solution:
If you edit vimrc often, it can be useful to create a shortcut to go there and another to source it.

I have the following mapping in my vimrc:

```
nnoremap <Leader>vs :source ~/.vimrc<CR>
nnoremap <Leader>ve :vsplit ~/.vimrc<CR>
```

When I need to open vimrc, I just run `<Leader>ve`. When I want to source my changes, after I save the vimrc, I just run `<Leader>vs`.

Line Numbers

Problem:
You need a clearer indicator to tell you which line you're on.

Solution:
Use `:set number` to display numbers on the left column.

Also, you might want to consider using `:set relativenumber` to display relative numbers on the left column. This setting is useful for line operations. You can easily see how far down or up the text is that you want to delete.

More Helpful Search

Problem:
You need to highlight the word as you're typing the search phrase.

Solution:
`set incsearch` sets up incremental search. With this, Vim will highlight the match as you are typing the search phrase.

This is best combined with `set hlsearch` to highlight all matches.

However, there are times when you don't want Vim to display highlight. To remove the highlight, run `:noh`.

I have this mapping to quickly run `noh` by pressing `<Esc>` twice because I use `noh` often.

```
nnoremap <Esc><Esc> :noh<return><Esc>.
```

Smarter Search Case

Problem:
You need Vim to search smarter and ignore case when needed.

Solution:
Use `set ignorecase smartcase` to do both case insensitive and smartcase search.

- If you search using all lowercase or all uppercase (homogeneous case) letters, Vim will do a case insensitive search.
- If you mix the case of the search case, Vim will do a case sensitive search.

Given:

```
hello
HELLO
Hello
```

- `/hello` matches "hello", "Hello" "HELLO".
- `/Hello` matches "Hello".
- `/HELLO` matches "HELLO".

Programmatic Options

Problem:
You need to set some options programmatically.

Solution:
You can use `&` to set an option in vimrc instead of `set`. With this, you can use Vimscript expressions on your options.

Instead of `set background=light`, you can run `let &background = "light"`.

To set different background values depending on the current time, use:

```
let &background = strftime("%H") < 10 ? "light" : "dark"
```

Create a Custom Command

Problem:
You need to create your own `:` command.

Solution:
You can create a custom command-line command with `:command`.

The following will create a custom command `:GimmeDate`.

```
function MyDate()
  echo call("strftime", ["%F"])
endfunction

command GimmeDate call MyDate()
```

Now when you run `:GimmeDate`, Vim will run `call MyDate()`.

Conditionally Run Vimrc Settings Based On Directory

Problem:
You want to have some options only when you're doing work and a different set of options on personal projects.

Solution:
To set your background to dark when you're on a "Work" directory and to set your background to light when you're not on a "Work" directory, add this to your vimrc:

```
let cwd = getcwd()
if cwd =~# "Work"
  echom "WORK STUFF"
  set background=dark
else
  echom "NON WORK STUFF"
  set background=light
endif
```

Organize Vimrc With the Fold Syntax

Problem:
Your vimrc is getting too long and too hard to read.

Solution:
The fold syntax can help to organize vimrc.

Vim fold syntax by default uses `{{{` and `}}}`. To activate it, run `:set foldmethod=marker`. You can toggle the fold with `za`.

To automatically setup marker folds for all vim related files including vimrc, add this at the start of your vimrc:

```
augroup filetype_vim
  autocmd!
  autocmd FileType vim setlocal foldmethod=marker
augroup END
```

You can now organize your vimrc based on categories, like:

```
" Plugins {{{
call plug#begin('~/.vim/plugged')
  your plugin here
call plug#end()
" }}}

" Setups {{{
set relativenumber number
set ignorecase smartcase
more setup here
" }}}

" Functions {{{
function! SomeUsefulFunction()
  dosomething
endfunction
" }}}
```

Each time you open vimrc, this is what you'll see:

```
+-- 5 lines: Plugins -------

+-- 5 lines: Setups --------

+-- 5 lines: Functions -----
```

Run a Specific Configuration Depending On File Type

Problem:
You want to use different options depending on the current file type.

Solution:
Vim allows you to run specific configurations when you're on a certain file type.

Vim can generally detect the type of the file you're on. To check the file type of the file you're currently on, run `:set ft?`.

For example, if you are on `test1.py`, running `:set ft?` will return `filetype=python`.

Vim has a filetype plugin runtime system that lets you run a particular Vim script depending on the filetype. You can find it inside the `.vim` directory. It is usually located in the root path.

Inside this `.vim` directory, create a directory named `ftplugin`. Inside it, create a file with the name of the file type you want to have the custom settings. In this case, let's create two filetype plugin files, one for python and one for ruby.

```
~/.vim/ftplugin/python.vim
# Inside it:
set background=light

~/.vim/ftplugin/ruby.vim
# Inside it:
set background=dark
```

When I open test.py, Vim has a light background. When I open test.rb, Vim will have a dark background.

Toggle a Boolean Option

Problem:
You need to quickly toggle any Vim boolean options.

Solution:
You can toggle any Vim boolean option values with `!` suffix.

Some values like `number`, `relativenumber`, `wrapscan`, etc contain boolean values (`set number`, `set relativenumber`, `set wrapscan`). Their opposite values are usually prefixed with `no`: `nonumber`, `norelativenumber`, `nowrapscan`, etc.

One way to turn off `set number` is to run `set nonumber`. A quicker way is to toggle it with `set number!`.

By the way, recall that `@:` can repeat the last command-line command. Since `:set number!` is just a command-line command, you can use `@:` to toggle between Vim boolean options quickly.

Split Vimrc Into Multiple Files

Problem:
Your vimrc got too big and you want to split it into multiple smaller files.

Solution:
You can keep your vimrc small by splitting it into multiple smaller files.

To use this, create a new directory for your vimrc files called `settings/` (you can name it anything you want): `~/.vim/settings/`.

Suppose that you want to split your vimrc into 4 sections, create these 4 files:

- Third-party plugins (`~/.vim/settings/plugins.vim`).
- General settings (`~/.vim/settings/configs.vim`).
- Custom functions (`~/.vim/settings/functions.vim`).
- Key mappings (`~/.vim/settings/mappings.vim`).

Inside your main vimrc, add:

```
source $HOME/.vim/settings/plugins.vim
source $HOME/.vim/settings/configs.vim
source $HOME/.vim/settings/functions.vim
source $HOME/.vim/settings/mappings.vim
```

The key here is to use the `:source` command. It is similar to `import` or `require` in many programming languages. Inside each of these setting files, add the usual vimrc settings and configs.

Inside `~/.vim/settings/plugins.vim`:

```
call plug#begin('~/.vim/plugged')
  Plug 'mattn/emmet-vim'
  Plug 'preservim/nerdtree'
call plug#end()
```

Inside ~/`.vim/settings/configs.vim`:

```
set nocompatible
set relativenumber
set number
```

Inside ~/`.vim/settings/functions.vim`:

```
function! ToggleNumber()
  if(&relativenumber == 1)
    set norelativenumber
  else
    set relativenumber
  endif
endfunc
```

Inside ~/`.vim/settings/mappings.vim`:

```
inoremap jk <esc>
nnoremap <silent> <C-f> :GFiles<CR>
nnoremap <Leader>tn :call ToggleNumber()<CR>
```

Now your main vimrc is only 4 lines long!

Mapping a New Key

Problem:
You need to create custom mappings.

Solution:
If you find yourself repeatedly performing the same complex task, it is a good indicator that you should create a new key mapping. There are a number of things that you can map.

The first is the leader key. I like using `<Space>` as my leader key. You can change the leader key with:

```
let mapleader = "\<space>"
```

You can also re-map operators and commands. For example, I have these two mappings in my vimrc:

```
nnoremap <silent> <C-f> :GFiles<CR>
```

```
nnoremap <Leader>tn :call ToggleNumber()<CR>
```

The first mapping is a map to `fzf.vim` plugin's `:GFiles` command. The second mapping is a custom function call (I have a function called `ToggleNumber()`. By using `<C-f>`, I am overwriting Vim's native `Ctrl-F` native page scroll feature. If your mapping collides with Vim's native command, it will be overwritten.

The command I use to map, `nnoremap`, is actually composed of three components:

- `n` represents the normal mode.
- `nore` means non-recursive.
- `map` is the map command.

Vim has a mapping for all sorts of modes. The `nnoremap` means it will work in normal mode.

If I want to create mapping in insert mode, I'll use the `i` prefix:

```
inoremap jk <Esc>
```

This only works in insert mode and not in normal mode.

It is also important to use the non-recursive version (by using `nnoremap` and `inoremap` instead of `nmap` and `imap`) because you could accidentally create an infinite loop.

Let's do an example. The mapping below supposedly moves the cursor to the end of the line, adds a semi-colon, then go back one WORD.

```
nmap B A;<Esc>B
```

So what happens when you press `B`? Vim adds `;` uncontrollably (interrupt with `Ctrl-C`), because in the mapping, the `B` in `A;<Esc>B` doesn't refer to Vim's native back-one-WORD action, but itself. What you actually have is a recursive call of:

```
A;<esc>A;<esc>A;<esc>A;esc>...
```

What you want is for Vim to use Vim's native `B` back-one-WORD action, so you use the non-recursive map:

```
nnoremap B A;<Esc>B
```

In general when you're not sure whether to use `nore` or not, always use it.

The other map modes are: map (Normal, Visual, Select, and Operator-pending), vmap (Visual and Select), smap (Select), xmap (Visual), omap (Operator-pending), map! (Insert and Command-line), lmap (Insert, Command-line, Lang-arg), cmap (Command-line), and tmap (terminal-job). I won't cover them in detail. To learn more, check out :h map.txt.

You may find that it is easier to copy someone else's vimrc. Although that's fine, I would suggest building your own vimrc from scratch. Everyone codes differently. Create a set of mappings that work best for you. Make your vimrc yours.

Installing Plugins With Packages

Problem:
You want to install plugins with Vim.

Solution:
Ever since version 8, Vim comes with its own built-in plugin manager called packages.

Vim will check inside a directory named `pack` inside the `.vim` directory (~`/.vim/pack/`). The package feature supports two loading mechanisms:

- Automatic plugin loading.
- Manual plugin loading.

To load a plugin automatically, you need to put it inside the `start/` directory using the following path:

```
~/.vim/pack/*/start/
```

Note the asterisk * means an arbitrary name. You can name it anything you want. In this case, let's call it `packdemo`:

```
~/.vim/pack/packdemo/start/
```

You cannot skip the directory between `pack/` and `start/`. If you try to do this, it won't work. You must put a directory name between `pack/` and `start/`.

```
~/.vim/pack/start/
```

Suppose that you want to install the NERDTree plugin. In the terminal, go all the way to the `start/` directory and clone the NERDTree repository there.

```
cd ~/.vim/pack/packdemo/start/
git clone https://github.com/preservim/nerdtree.git
```

That's all. Once the library plugin is inside the `start/` directory, Vim automatically uses it when it starts. The next time you start Vim, you can now use the NERDTree commands. You can clone however many plugin repositories as you need inside this path. To remove a plugin, just remove that plugin directory from the `start/` directory.

To load plugins manually, you need to put it inside the `opt/` directory:

```
~/.vim/pack/*/opt/
```

Again, the `*` stands for "any directory name". You can use the same directory name or a different one. In this case, let's use `packdemo/` name again.

```
~/.vim/pack/packdemo/opt/
```

Now let's install the killersheep game (requires Vim 8.2).

```
cd ~/.vim/pack/packdemo/opt/
git clone https://github.com/vim/killersheep.git
```

Now when you start Vim, the game won't be available yet. To activate a package in the `opt/` directory manually, run `:packaddd {package-name}`.

```
:packadd killersheep
```

Now run the killership command, `:KillKillKill`. Enjoy!

Installing Plugins With Plugin Managers

Problem:
You need to install plugins without Vim's packages.

Solution:
There are many Vim plugin manager available out there:

- vim-plug[9]
- vundle.vim[10]

[9]https://github.com/junegunn/vim-plug
[10]https://github.com/VundleVim/Vundle.vim

- vim-pathogen[11]
- dein.vim[12]
- … and many more!

I personally use vim-plug, but most plugin managers are also equally good. Many of them are also easy to install. Check them out and find the one you like best!

Trigger Action On Certain Event

Problem:
You want to perform a specific action each time a file is saved.

Solution:
Vim has an autocommand that fires when a certain event is triggered, `:autocmd`.

For example, to set a `filetype` to `awesomefile` each time you open an `.awesome` file extension, add this in vimrc:

```
autocmd BufNewFile,BufRead *.awesome set filetype=awesomefile
```

To automatically generate tags when you're saving a Python file, run:

```
autocmd BufWritePost *.py silent !ctags -R .
```

Autocommand supports myriads of different events. To see what all the events are, check out `:h autocommand-events`.

[11]https://github.com/tpope/vim-pathogen
[12]https://github.com/Shougo/dein.vim

Window

Splitting Windows Horizontally and Vertically

Problem:
You need to split the current windows.

Solution:

- You can use `Ctrl-W s` or `:sp` to split the window horizontally. You can also do `:sp somefile.txt` to explicitly specify which file to open on the split window.
- You can use `Ctrl-W v` or `:vsp` to split the window vertically. You can also do `:vsp somefile.txt` to explicitly specify which file to open on the split window.

Resizing Windows

Problem:
You need to resize the current window.

Solution:
You can change the window length and width:

- To make the current window longer or shorter, run `Ctrl-W +` or `Ctrl-W -`.
- To increase or decrease the current window width, run `Ctrl-W >` or `Ctrl-W <`.

Alternatively, you can also resize with:

- To set the current window height N, run `zN<Enter>` (ex: `z10<Enter>` to set the window height to 10). You can also run `:resize N` to set the height to N (ex: `res 10` to set it to 10).
- To set the current window width, run `Ctrl-W |`. By default, it will set it to the max width. You can also run `:vertical res N` where `N` is the width (ex: `:vertical res 10` to set it to 10).

Finally, if you have multiple windows open with different heights and widths, run `Ctrl-W =` to make all windows to have equal sizes.

Close or Zoom the Current Window

Problem:
You need to quickly close the current window or zoom the current window.

Solution:

- Use `Ctrl-W c` to close the current window (you can't close it if it is the last window). You can also run `:close`.
- Use `Ctrl-W o` to "enlarge" the current window and close other open windows. You can also run `:on`.

Moving the Cursor to Another Windows

Problem:
You have multiple windows open and you need to move your cursor to a different window.

Solution:
Use `Ctrl-W h/j/k/l` to go to the left, down, up, or right window.

You can also Use `Ctrl-W w` to move to another window. You can pass a count to this command. If you run `3 Ctrl-W w`, Vim will go to the 3rd window.

Open a New Window

Problem:
You need to quickly open a new window.

Solution:

- Use `Ctrl-W n` or `:new` to create a new horizontally split buffer.
- Use `:vnew` to create a new vertically split buffer.

Use Two Windows to Diff Files

Problem:
You need to diff two files from Vim.

Solution:
To create a diff between two files, you can use `:diffthis`. Here is how to use it:

- First open `file.txt` in Vim.

- Then open another file, `file2.txt`, in split mode (`:vsp file2.txt`).
- In the first buffer, run `:diffthis`.
- In the second buffer, run `:diffthis`.
- To turn it off, run `:diffoff`.

Essentially, you need to run the `diffthis` command from each window.