# Learning to play Vim


Master the best text editor,
from Beginner to Composer

Matthieu Cneude

# Learning to Play Vim

Matthieu Cneude

*Learning to Play Vim*

# Rank I - Vim for Rookies

“Where am I? What am I doing here?”

This is your first thought as you wake up on a small beach surrounded by rocks. You were on a boat last night, with your billionaire friends drinking some champagne and eating some caviar, as usual.

You don’t remember what happened, and how you ended up here. You have a slight headache, amplified by the harsh sun enveloping your whole body. Thanksfully you’re still perfectly dressed, and perfectly dry too. It doesn’t make much sense.

While looking around you, you catch the sight of a laptop bag abandoned on a rock. You reach it and open it quickly, wondering if there is an actual computer inside. There is! You might be able to contact somebody with this machine.

After the computer starts, a terminal run and opens a file, using a program called “Vim”. Here’s the content of it:

“Greating! Welcome to my Island! I hope you feel well.

I’ve followed you for quite a while now. Your lifestyle is so unhealthy! I don’t like your clothes, either. But nobody cares about that; what annoys me is your poor knowledge of the Glorious Vim.

That’s right! You’re here to become a Vim Composer, a rank which is only given to our best Vim players out there.

Why you? Why have you’ve been chosen for this incredible honor? Because of your extreme intelligence and the beauty of your skin, mainly.

Now, go! Explore this island, it will allow you to become better at Vim. If I’m satisfied with your success, you might be able to escape this Island.”

The file goes on, explaining the basics of Vim. There is a signature at the end: “The Island Master and The Carrier of The Word of Vim”.

Hoping to find a clue what happened to you, you begin to read the file. It begins by a little table of content for the first chapter:

- The different modes of Vim: this is what make Vim so different from other editor, and so powerful too.
- How to move around with your precious keyboard.
- How to speak with Vim, again with your keyboard.
- How to undo and redo your changes.
- How to search in the current files.
- How to create Vim’s configuration.

Are you ready for the beginning of this adventure?

## The Modes of Vim

Vim is a *modal editor*. It means that the keys of your keyboard will perform different actions depending on the mode you're in.

This is the first big difference with the text editors you're likely to be used to, and arguably one of the biggest reason why Vim is so beloved and powerful.

The modes you'll use the most are the NORMAL mode and the INSERT mode. This is what we'll discuss next; it's primordial to understand both of them to get what Vim is about.

### The NORMAL Mode

Many text editors out there allow us to directly type some text with our beloved keyboards, as soon as they're running. What you type appear immediately on your screen. It's so usual that anybody starting Vim for the first time feels very confused.

That's right: Vim doesn't work like that.

To understand what I mean, try to open Vim. You'll see a welcome message. Now, if you try to type "x" in the editor, you might be surprised: no "x" appears on the damn screen! How come?

It's because Vim always start by default in NORMAL mode. This mode is not meant to *insert new content*, but to *edit already existing content*.

In NORMAL mode, we can use many different commands to move the cursor and target the exact content we want to edit. For example, instead of using the mouse to select the word you want to replace, you'll use your keyboard to do the same thing.

You can think of NORMAL mode commands as keystrokes, or keyboard shortcuts. Usually, in other text editors, you would need to hit a shortcut to perform an action, like selecting all your text with `CTRL-a`. Since you don't write anything in NORMAL mode, you have way more keystrokes to manipulate your text. You have access to all the keys on your keyboard, in fact.

But if there are so many NORMAL mode commands, how is it possible to remember them all? It's where it gets very interesting. These commands make sense in Vim, most of the time, compared to the usual and meaningless shortcuts you'll find in other editors.

Vim's NORMAL mode commands use mnemonics for you to remember them easily. Even better: they are *composable*: you can combine some of them in a logical, easy-to-remember way.

Let's take for example the shortcut `CTRL+shift+n` from a random IDE. By only looking at it, you've no idea what it does. In contrast, you'll see soon that it's possible to guess NORMAL mode commands in Vim.

Vim's NORMAL mode is a keyboard-centered way to control your editor, *telling* Vim what you want to do, for it to obey your mighty will.

To really grasp this concept, we need some text to edit. As we said, we write new text in NORMAL mode; so let's switch to the second most important mode, the INSERT mode.

## The INSERT Mode

Let's now execute our first NORMAL mode command, which will switch Vim from NORMAL to INSERT mode. Simply hit `i` on your keyboard.

Depending on the editor you use (Vim or Neovim), and how your terminal is configured, the shape of your cursor might change. More importantly, you'll see the string `--INSERT--` in the bottom left corner of Vim.

Welcome to INSERT mode.

In this mode, you're finally able to type the content you always wanted to bring to the world. Go ahead, don't be afraid: type anything you want, like you would do in more mainstream editors.

Now, let's try to hit the `ESC` key. The indicator `--INSERT--` disappears.

Welcome back to NORMAL mode.

That's what do a Vim user, most of the time: juggling between NORMAL mode to edit existing content, and INSERT mode to insert new content.

The command `a` in NORMAL mode can also let you switch to INSERT mode, but it will do it `a`fter the character you're on. You can try it to see the difference.

That's what I'm talking about when I say that Vim uses mnemonic for the NORMAL mode commands: `i` for insert, `a` for insert `a`fter.

Let's summarize this knowledge:

| | |
|---|---|
| `i` | Switch from NORMAL mode to INSERT mode (`i`nsert before current character). |
| `a` | Switch from NORMAL mode to INSERT mode (insert `a`fter current character). |
| `<Esc>` | Switch from INSERT mode to NORMAL mode. |

There are more NORMAL mode commands allowing us to switch to INSERT mode. We'll see them in the next chapter; for now, the ones above should be enough for your editing needs.

## The COMMAND-LINE Mode

The NORMAL mode and the INSERT mode are the ones you'll use most often. Following the order of importance, we'll find a third mode, the COMMAND-LINE mode.

Now, you might have noticed that we've spoken about NORMAL mode *commands*. What's this COMMAND-LINE mode? A new way to enter commands? Well, kind of.

In NORMAL mode, you run NORMAL mode commands; in COMMAND-LINE mode, you run *Ex-commands*. This is quite confusing; that's why I'll speak about NORMAL mode keystrokes in this book, instead of NORMAL mode commands. That said, be aware that many other resources about Vim (including Vim's help) often speak about NORMAL mode commands.

So, what's the point of this COMMAND-LINE mode? You can run these special Ex-commands allowing you to do many interesting things.

First, to switch to COMMAND-LINE mode, you need to use the NORMAL mode keystrokes `:`. Like in INSERT mode, to come back to NORMAL mode (our default, and soon favorite, mode), you need to hit the `ESC` key.

When you switch to COMMAND-LINE mode, your cursor moves automatically at the very bottom of Vim. Your cursor will be placed after a semi-colon `:`, indicating that you can run Ex-commands.

You can think of all these Ex-commands as the menu you would normally use in a text editor with a graphical user interface (GUI). In this mode, you can run Ex-commands to save the file you're editing, substitute some text, and much, much more.

Here are some useful basic Ex-commands you'll need to save your work, or simply quit Vim:

| Command | Short Name | Description |
|---|---|---|
| `:write` | `:w` | Write (save) the current file open. |
| `:write!` | `:w!` | Write (save) the current file open (even if it's read-only). |
| `:edit {filepath}` | `:e {filepath}` | Edit the file with filepath `{filepath}`. |
| `:quit` | `:q` | Quit the current window. |
| `:quit!` | `:q!` | Quit the current window without saving. |
| `:wq` | `:wq` | Write (save) the current file and quit Vim. |

A last important Ex-command, maybe the most important of all: `:help {subject}`, to open the help about whatever `{subject}` you want.

This help is insanely complete. If you don't remember how to quit Vim for example, you can type `:help quit`.

I'll reference Vim's help very often in this book, in special blocks, at the end of some sections. They will allow you to dig deeper into the functionalities we'll cover.

For example:

```
:help vim-modes
:help write-quit
```

Don't worry if you don't understand what's written in Vim's help, or if there is too much information. When you'll get more comfortable with Vim, it will make more sense. Simply bear with me for a little while.

A last important tip concerning the COMMAND-LINE mode: you can use the `TAB` key to complete Ex-commands. Useful when you don't remember exactly the commands, or to discover new ones!

Like in a shell, you can also use the arrow keys `<up>` and `<down>` to go through your Ex-command history.

## Moving Around with Motions (NORMAL mode)

Let's now see how we can move our cursor horizontally or vertically, thanks to NORMAL mode keystrokes called *motions*.

Don't worry if you don't remember every single NORMAL mode keystroke we'll see here. You can always come back to this chapter and experiment with the one you forgot.

## Ditching the Arrow Keys

We're now at the most difficult part in our journey to learn Vim. At least it was the most difficult part for me: ditching the arrow keys to move the cursor.

As I said in the previous chapter, our fingers should stay on the row keys of the keyboard. First, for our typing speed and accuracy to improve, and second because the Vim's keystrokes you can use in NORMAL mode are more easily accessible from the row keys. Your hands shouldn't move too much; only your fingers should.

If you look at your arrow keys, you'll see that they're too further away from the row keys, forcing you to move your hands each time you want to use them. That's why, instead of using the arrow keys, many Vim users use the `h`, `j`, `k` and `l` keys instead, to move respectively left, down, up and right.

I would highly encourage you to use these keys, too. They'll improve your Vim experience significantly.

Why `hjkl`, and not some other keys close to the home row? For historic reasons. Vim is the ancestor of Vi, which was used on physical terminals. When you look at the keyboard of some of them, you'll see that the arrow keys are also the `hjkl` keys.

Like many other habits which seem ingrained in our brain, it will be difficult not to use the arrow keys at first. Your hand will come back to them over and over, even if you try not to. You need to accept this fact and be patient; you'll get there, and faster than you think.

For an easier transition, we can try to answer an important question: how to remember what `h`, `j`, `k` and `l` do in NORMAL mode? Here are some useful mnemonics:

- The `h` key is on the left of the sequence `hjkl`, and `l` is on the right. As a result, hitting `h` will move your cursor to the left, and `l` to the right.
- `j` moves your cursor down. Here are 3 mnemonics for this key:
  - With a bit of imagination, you can see `j` as an arrow going down.
  - The `j` has a little bump at the bottom of the key, meaning that it goes down.
  - Let's speak typography: `j` has a descender, meaning that part of the letter descends from its baseline. As a result, `j` goes down.
- `k` is the only letter left, so it has to go up. To come back to the secret art of typography, `k` is a letter with an ascender, meaning that part of the letter ascend from its baseline. As a result, `k` goes up.

Practice will get you there. I've got you covered for this one, with a revolutionary AAA game everybody will speak about in twenty years. To play it, you *must* use `hjkl`. If you prefer puzzle games, try this wonderful sokoban. You can use `hjkl` or the arrow keys this time, but try to only use `hjkl`.

## Horizontal Motions

The keys `h` and `l` are not the only ones you can use to move horizontally, on the current line. Actually, long time Vim users rarely use them. Instead, we can use other motions in NORMAL mode to move faster, like these:

| Keystroke | Description |
|---|---|
| `w` | Move forward to the beginning of the next `w`ord. |

| Keystroke | Description |
|---|---|
| `W` | Move forward to the beginning of the next `W`ORD. |
| `e` | Move forward to the `e`nd of the next word. |
| `E` | Move forward to the `e`nd of the next WORD. |
| `b` | Move `b`ackward to the beginning of the word. |
| `B` | Move `b`ackward to the beginning of the WORD. |

A question arise: what's the difference between a "word" and a "WORD"? They represent two different motions. A "WORD" follows the usual concept of a word; a string of characters delimited by spaces.

You can think of a "word" as a keyword, containing only a specific set of characters. Mainly, it doesn't include some special characters.

For example, try to type in Vim the following:

```
-- A word and a WORD: see the difference?
local function restorePosition() {
```

Now, try to use the motions we've seen above to see the difference. Using "WORD" motions will skip the parenthesis, but the "word" motion won't.

Here are some more horizontal motions I find particularly useful:

| Keystroke | Description |
|---|---|
| `f{character}` | To `f`ind a `{character}` after your cursor. |
| `F{character}` | To `f`ind a `{character}` before your cursor. |
| `t{character}` | Move `t`ill a `{character}` after your cursor. |
| `T{character}` | Move `t`ill a `{character}` before your cursor |

After using one of these four keystrokes above, you can continue to move from character to character with:

| Keystroke | Description |
|---|---|
| `;` | Move forward |
| `,` | Move backward |

`:help cursor-motions`
`:help left-right-motions`

## Beginning and End of Line

Moving word by word can be slow and boring if you want to go to the beginning or the end of the line. Here are some more NORMAL mode keystrokes:

| Keystroke | Description |
|---|---|
| `0` | Move to the first character of the current line. |
| `$` | Move to the last character of the current line. |
| `^` | Move to the first non-whitespace character on the current line. |

A whitespace can be a space or a tab.

### Vertical Motions

Here are more NORMAL mode keystrokes, but to move your cursor vertically this time:

| Motion | Description |
|---|---|
| `{line_number}G` | Move to the be `g` inning of the line numbered `{line_number}`. For example, `10G` will move the cursor on line 10. |
| `G` | Move to the last line of your file. |
| `1G` or `gg` | Move to the first line of your file. |
| `CTRL-u` | move `u` pward for half a screen. |
| `CTRL-d` | move `d` ownward for half a screen. |
| `CTRL-b` | move `b` ackward (upward) an entire screen. |
| `CTRL-f` | move `f` orward (downward) an entire screen. |

You can also use COMMAND-LINE mode to move to a specific line number, with `:{line-number}`. For example, `:10` will move your cursor to line 10.

Finally, here are three last keystrokes allowing us to move to the top, middle, or last line of the actual display:

| Motion | Description |
|---|---|
| `H` | Move to the top (`H` ome) of the window. |
| `L` | Move to the `l` ast line of the window. |
| `M` | Move to the `m` iddle line of the window. |

> `:help up-down-motions`

## The Language of Vim (NORMAL Mode)

If you searched online some tutorials or other resources about Vim, I'm pretty sure you've seen this one: Vim has a language! You can speak with your editor!

In Vim, NORMAL mode keystrokes can be seen as "sentences", describing an action you want to perform. That's what I meant when I was saying that the keystrokes are *composable*. It's nothing less than brilliant.

These "sentences" are quite easy to understand. It allows us to link what we know already (the sentence) by what we need to learn (the NORMAL mode keystrokes).

Even better: knowing that Vim has a "keystroke language" will push you to combine them instinctively to do what you need to do, and, in many cases, it will work!

## The Operators

We've learned how to walk in Vim with motions. It's time to perform some actions. To operate on our content.

The *operators* are the verbs of the Vim language. Here are three common operators:

| Operator | Description |
|---|---|
| `d` | `d`elete |
| `c` | `c`hange |
| `y` | `y`ank (copy) |

These operators won't do anything if you hit them in NORMAL mode. You need to combine them with motions. For example:

| | |
|---|---|
| `d$` | To `d`elete from your cursor to the end of line. You can also use the alias `D`. |
| `dgg` | To `d`elete everything from the cursor to the beginning of the file. |
| `ggdG` | Move your cursor to the beginning of the file, and delete everything till the end. |
| `dl` | Delete a character. You can also use the alias `x`. |

Let's explain a bit further what these three operator are doing:

- The delete operator is self-explanatory.
- The change operator will delete and immediately switch to INSERT mode, effectively allowing us to... change our text.
- The yank operator allows us to copy some of our text. It can be then paste somewhere else in Vim, using the NORMAL mode keystroke `p`, for `p`ut.

By default, the keystroke `p` will paste the content after the character under the cursor. To put it after, use the uppercase variant of the keystroke `P`.

I encourage you to try out all these operators. You can combine them with the motions we've seen in this chapter. Again, more practice you'll have, better you'll get! The exercises at the end of the chapter will help you to get there, too.

```
:help operator
:help objet-motions
```

## The Text-Objects

Instead of motions, we can also use another construct with our operators: the famous Vim text-objects. If the operators are the verbs of the Vim language, the text-objects are the nouns.

Simply put, a text-object is a set of character with a specific start and end. In Vim, "a word" is a text object, as well as "a sentence", or "a paragraph".

For example, you can use operators and text-objects in NORMAL mode as follows:

| Keystroke | Description |
|---|---|
| `diw` | To `d`elete `i`nside the `w`ord.<br>It deletes the current word under the cursor. |
| `daw` | To `d`elete `a`round the `w`ord.<br>It deletes the current word under the cursor and its leading and trailing whitespaces. |
| `ciw` | To `c`hange `i`nside the `w`ord.<br>It deletes the current word under the cursor and switch to INSERT mode.<br>In short, you... change the word! |
| `dip` | To `d`elete `i`nside the `p`aragraph. |

Note that each of these examples are composed of an operator and a text-object. For example, for the first example, `d` is the operator, `iw` is a text-object.

In Vim's help, `daw` is described as `d`elete `a` `w`ord. I find this "translation" quite confusing, that's why I use `a`round instead of `a`. Indeed, text-objects beginning with a `a` (like `aw`) often delete something more than the text-object beginning with `i` (like `iw`).

There are more text-objects you can use for your editing needs. You can discover new ones in the bonus exercises at the end of this chapter, or by looking at Vim's help.

```
:help text-objects
```

## Undo and Redo (NORMAL mode)

I would be totally lost if I didn't have any way to undo or redo my work. Here's how to do so in NORMAL mode:

| | |
|---|---|
| `u` | To `u`ndo your last edit. |
| `CTRL-r` | To `r`edo. |

You can think of `CTRL-r` as you being in control (`CTRL`) of your content.

You'll notice that whatever you're doing in INSERT mode is equivalent to one undo. We'll learn how to change this behavior later in the book.

```
:help undo-redo
```

## Bending Vim to Our Will (Customization)

In Vim, almost everything is configurable. It's insane, I tell you. You can shape your editor according to your megalomaniac desires. Let's see the very basics here.

## Creating Your Configuration

Your main configuration file should be in the following path by default, depending on what you use:

| | |
|---|---|
| For Vim | `~/.vimrc`. |
| For Neovim | `$XDG_CONFIG_HOME/nvim/init.vim` |

This file is sourced when Vim starts. Except that, they have nothing special; you can create another file with some Vimscript inside (including Ex-commands) and source it manually if you want. We'll explore this functionality later in this book.

Whatever you're using, I'll call this configuration file *the vimrc* throughout the book.

As you can see, the path of Neovim's vimrc depends on the environment variable `$XDG_CONFIG_HOME`. It's most likely `~/.config`. If you don't know what are the XDG user directories, here's a good resource to learn more about them.

Let's now write our first lines of configuration. I would encourage you to write them using Vim. What you've learned in this first chapter should be enough for you to put your toes into Vim's relaxing waters.

To open this file, you can run `vim {path}` in your terminal. For example: `vim ~/.vimrc`.

First, let's add the following to our config:

```
noremap <Up> <Nop>
noremap <Down> <Nop>
noremap <Left> <Nop>
noremap <Right> <Nop>
```

The Ex-command `noremap` allows us to bind a key to something else. We'll see this command more in details in the third chapter of the book.

Here, we assign the arrow keys to... nothing. This will force you to use `hjkl` instead. It might be a bit painful at first, but, trust me, it's for your own good. You can also conclude that I'm just a little pig, and then decide to use the arrow keys all your life; I'll be sad, but it's your right.

Here's another line we can add to our vimrc:

```
set clipboard+=unnamedplus
```

It will make the copy-paste mechanism less confusing, till you learn more about it in chapter IV.

We now have Ex-commands on every line of our file. That's right: you could also run them in COMMAND-LINE mode, but they would be reset the next time you're closing Vim. In fact, you can add any Ex-command in your vimrc, and Vim will automatically execute them during its startup, in order.

We've already seen that these commands have a long and short form. You can use either of them in COMMAND-LINE mode; but, when you write these Ex-commands in a vimrc, I would encourage you to use the long form for a better readability.

If you use Vim instead of Neovim, we need to add these lines too:

```
" No compatibility with Vi
set nocompatible

" Display line numbers
set number

" Better completion in COMMAND-LINE mode
set wildmenu

" Syntax highlighting
syntax on

" Enable filetype, indentation, plugin
filetype plugin indent on
```

As you can see, every line beginning with a double quote `"` is a comment.

Some explanations:

- `set nocompatible` - Vim looks less like Vi, its ancestor. That's great, because we don't want to use Vi, but Vim.
- `set number` - Display the line numbers.
- `syntax on` - Enable the syntax highlighting.
- `filetype plugin indent on` - Load a bunch of files to set automatically the indentation, and other options depending on the type of files open in Vim.

To see the effect of your new configuration, you can relaunch Vim and try to use your arrow keys. They don't work anymore. Great! Out of some constraints can come great creativity.

With all this knowledge in mind, I advise you to add some comments for everything you add to your vimrc, at least at the beginning. For example:

```
" Easier copy-paste from other applications to Vim
set clipboard+=unnamedplus
```

`:help vimrc`

## The Configuration Addiction

At that point, I'd like to warn you: configuring Vim can become addictive. Not I-lost-my-house-and-my-partner-left-me kind of addictive, but you can easily spend (too) many hours trying to come up with the best configuration in the universe.

My advice: just try to add what's useful for you, step by step. Don't try to recreate all the functionalities you had in your text editor or, even worst, your IDE. You'll get eventually there when we'll speak about plugin a bit later in this book but, before that, you should consider trying to understand and use the functionalities directly available in Vim.

## Exercises

To solve these exercises, open with Vim the file `rank_01/functions.lua` from the book companion. You can do so by going into the root of the repository and run `vim rank_01/functions.lua`.

Don't forget that you can quit Vim with the Ex-command `:q`. Add a *bang* to the command to quit without saving: `:q!`.

Each exercise has a series of question. You should solve them in order. The changes done for each question is the starting point for the next one.

### Exercise 1 - Horizontal Motions

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of `restorePosition`:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

How would you:

1. Move to this position: `function`?
2. Move back to this position: `restorePosition()`?
3. Move to the end of the line?

### Exercise 2 - Horizontal Motions, the Return

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of the following line:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

**Using only one key per question**, how would you:

1. Move to this position: `if vim.fn.line("'\"") > 1`?
2. Move to this position: `if vim.fn.line("'\"") > 1`?
3. Move to this position: `if vim.fn.line("'\"") > 1`?
4. Move to this position: `if vim.fn.line("'\"")`?

### Exercise 3 - Vertical Motions

In NORMAL mode, how would you:

1. Move to the 5th line of the file?
2. Move back to the very beginning of the file?
3. Move to the very end of the file?
4. Move multiple lines upward with one keystroke?

## Exercise 4 - Operators and Text Objects

Using the `hjkl` keys in NORMAL mode, move your cursor to the following position:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

1. How can you delete inside a word, keeping the space(s) surrounding it?
2. How can you undo what you just did?
3. How can you delete the word `local` , this time with the space following it?
4. Undo your change. How would you delete the word `local` , and directly switch to INSERT mode, only using 3 keys?
5. Again, undo your change. Move to the `a` of the word `local` , and hit `dw` . Can you explain what happened? What's the difference between `dw` and `diw` ?

## Exercise 5 - Operators, Motions, and Text-Objects

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of the word "vim" as follows:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

1. How would you move to the first parenthesis of the same line, only using two keys?
2. How would you delete everything inside these parentheses?
3. Undo your change.
4. How would you move your cursor at the "i" of "line", using only two keys (including the key `i` ), as follows: `if vim.fn.line("'\"") > 1` ?
5. How would you move to the "i" of "vim", using only one key, as follows: `if vim.fn.line("'\"") > 1` ?
6. While staying on the same line, try to use `fn` , `Fn` , `tn` , `Tn` .Then, try to use `;` or `,` , to get used to these movements. Try to replace "n" with other characters present on the line, too.

## Exercises BONUS

These exercises dive deeper in some concept seen in the chapter.

### Exercise 6 - More Text Objects!

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of the function "restorePosition" as follows:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

A paragraph is also a text object. It starts on an empty line, and end on the next empty line.

1. How would you delete the whole function `restorePosition` only using three keys?
2. Undo your change. How would you delete the next block of parenthesis, only using three letter keys?
3. Undo your change. How would you delete a sentence?
4. To find the start and end of the text-object "sentence", what Ex-command would you use?
5. What text-object could be useful to edit some HTML? Do you think this text-object exists in vanilla Vim?

### Exercise 7 - More Motions!

Using the `hjkl` keys in NORMAL mode, move your cursor at the beginning of the function name "restorePosition" as follows:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

1. We've seen the text-object "paragraph" in the previous exercise. Using Vim's help, try to find what motion allowing you to move from one paragraph to another.
2. How would you delete the letter `r` in `local function restorePosition()` with two letter keys? One?
3. Undo your change. How would you delete the space before your cursor: `local function restorePosition(` with two letter keys?
4. The answers of the two previous questions use motions or text-objects?

### Exercise 8 - Swapping

Using the `hjkl` keys in NORMAL mode, move your cursor on the character `l` at the beginning of the word `local` :

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

Undo all your changes after each question to come back to the starting position above.

How would you:

1. Swap the character `l` you're on, with the following character `o`, only using two keys?
2. Swap the current line you're on, with the line just below, only using three keys?
3. Swap the word `local` with the word `function` using five keys?

### Exercise 9 - Up and Down Following Indentations

Using the `hjkl` keys in NORMAL mode, move your cursor on the character `l`, at the beginning of the word `local`:

```
local function restorePosition()
    if vim.fn.line("'\"") > 1 and vim.fn.line("'\"") <= vim.fn.line("$") then
        vim.cmd([[normal! g`"]])
    end
end
```

How would you:

1. Move to the next line, on the `i` of `if`, only using one key?
2. Move back to the starting position, only using one key?

## Solutions

### Exercise 1

| Question | Keystroke | Result |
|---|---|---|
| start | | `local function restorePosition()` |
| 1. | `b` | `local function restorePosition()` |
| 2. | `w` | `local function restorePosition()` |
| 3. | `$` | `local function restorePosition()` |

### Exercise 2

| Question | Keystroke | Result |
|---|---|---|
| start | | `    if vim.fn.line("'\"") > 1` |

| Question | Keystroke | Result |
|---|---|---|
| 1. | `^` or `w` | `if vim.fn.line("'\"") > 1` |
| 2. | `w` | `if vim.fn.line("'\"") > 1` |
| 2. | `W` | `if vim.fn.line("'\"") > 1` |
| 4. | `0` | `if vim.fn.line("'\"")` |

## Exercise 3

| Question | Keystroke | Result |
|---|---|---|
| start | | `#!/usr/bin/env lua` |
| 1. | `5G` | `vim.cmd([[normal! g`"]])` |
| 2. | `1G` or `GG` | `#!/usr/bin/env lua` |
| 3. | `G` | `}` |
| 4. | `CTRL-u` | `vim.cmd([[normal! g`"]])` |

Note that you can also move at the fifth line of the file in COMMAND-LINE mode, with `:5`.

## Exercise 4

| Question | Keystroke | Result |
|---|---|---|
| start | | `local function restorePosition()` |
| 1. | `diw` | `function restorePosition()` |
| 2. | `u` | `local function restorePosition()` |
| 3. | `daw` | `function restorePosition()` |
| 4. | `ciw` | `\| function restorePosition()` |
| 5. | `dw` | `locfunction restorePosition()` |

4. All the NORMAL mode keystrokes `dw`, `daw`, and `diw` use the operator `d` (for `d`elete).

The difference between these keystrokes: `w` is a motion, `aw` and `iw` are text-objects.

If you:

- Use a motion without operator, you'll move your cursor.
- Use a motion with an operator, you'll operate from the cursor position to the destination of the motion.

That's why `dw`, in this example, delete from the cursor position to the beginning of the next word `function`.

Said differently, the start and the end of a text-object are always the same: for example, the start of `aw` will be the first letter of the word the cursor is on, and the end will be the potential space after the word.

Another difference: a text-object always need to be prefixed by an operator. A motion can be used on its own.

### Exercise 5

| Question | Keystroke | Result |
|---|---|---|
| start | | `if vim.fn.line("'\"") > 1` |
| 1. | `f(` or `t"` | `if vim.fn.line("'\"") > 1` |
| 2. | `da(` or `dab` | `if vim.fn.line > 1` |
| 3. | `u` | `if vim.fn.line("'\"") > 1` |
| 4. | `Fi` | `if vim.fn.line("'\"") > 1` |
| 5. | `;` | `if vim.fn.line("'\"") > 1` |

## Exercises BONUS

### Exercise 6

| Question | Keystroke | Result |
|---|---|---|
| start | | `local function restorePosition()` |
| 1. | `dap` or `dip` | `local function deleteTrailingWS()` |
| 2. | `dab` | `local function restorePosition` |
| 3. | `das` or `dis` | `g`"]])` |

4. You first need to go to COMMAND-LINE mode by typing `:`. Then, you need to use the help Ex-command `help`, followed by what you want to find, i.e `sentence`, or `text-object`. In short: `:help sentence` or `:help text-boject`.
5. The text object `at` or `it` stand for "around an HTML tag" and "inside an HTML tag", respectively. Add whatever operator as a prefix to edit your HTML easily.

### Exercise 7

1. You'll find this info by running the Ex-command `:help object-motions`. You can use the motion `{` to move up a paragraph, `}` to move down.

Other than that, you could try `:help cursor-motions` and search for "paragraph", by running `/paragraph`.

| Question | Keystroke | Result |
|---|---|---|
| start | | `local function restorePosition()` |
| 2. | `dl` or `x` | `local function estorePosition()` |
| 3. | `dh` | `local functionrestorePosition()` |

4. We're using motions here: `l` is the motion to move one character to the right, `h` to move one character to the left.

The letter `l` is also a useful mnemonic for `l`etter. Even if it's easier to use `x` to delete a character, you can use other operators with the motion `l`, like `yl` if you want to copy it.

## Exercise 8

| Question | Keystroke | Result |
|---|---|---|
| start | | `local function restorePosition()` |
| 1. | `xp` | `olcal function restorePosition()` |
| 2. | `ddp` | `local function restorePosition()` |
| 3. | `dwelp` | `function local restorePosition()` |

## Exercise 9

| Question | Keystroke | Result |
|---|---|---|
| start | | `local function restorePosition()` |
| 1. | `+` | `if vim.fn.line("'\"") > 1` |
| 2. | `-` | `local function restorePosition()` |