Hacking your way around in Emacs

Marcin Borkowski

Hacking your way around in Emacs

Marcin Borkowski

This book is for sale at http://leanpub.com/hacking-your-way-emacs

This version was published on 2021-11-03

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Marcin Borkowski by spreading the word about this book on Twitter!

The suggested hashtag for this book is #hackingyourwayemacs.

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

#hackingyourwayemacs

Contents

Introduction

The goal of this book is really very simple: to teach you, dear reader, how to program Emacs, or rather, to help you teach it to yourself. Emacs – and Emacs Lisp – have grown to be quite complex, but that doesn't mean you need a Ph.D. to code in Elisp. In this book, we will start small, though not *very* small – I assume that you've read the late Robert J. Chassell's excellent An introduction to programming in Emacs Lisp[1]. It's well-written, informative, pretty light and free, so why not? I will repeat a thing from that book here and there, but mostly I will build upon what you already know from it.

Of course, the first question we might want to ask is: *why* would you want to program Emacs? If you are an Emacs user, you most probably know the answer – to make your editing experience better. Maybe there is some behavior of Emacs which annoys you and you'd like to change it. Maybe you have some repetitive action you want to automate. Maybe you use some application which can be used via a CLI or a Web API and you want to drive it from Emacs. There can be probably a ton of other reasons.

If you are not (yet) an Emacs user, this book is probably not for you. If you heard that Emacs is a fantastic, programmable text editor and you could benefit from using it and coding in Emacs Lisp, you should probably start with learning how to *use* Emacs. The first resource you might want to check out is the built-in Emacs tutorial, and you could proceed to the (very fine!) Emacs manual or the Mastering Emacs[2] blog and book by Mickey Petersen. Then, at some point in time you can read *An introduction to programming in Emacs Lisp*, and come back to this book afterwards.

Ok, so you're still here. Great! Let me explain a bit about what I wanted to achieve with this book.

I spent considerable time thinking what to include here and what to leave out, and in what order to present things. And really, there is no universal way of doing it. It was clear to me from the beginning that I should start easy and then move on to more sophisticated topics, but that opened more questions than it answered. After a lot of internal discussions with myself and a few drafts, I decided on a few guiding principles for myself to follow.

First of all, I try not to pull rabbits out of my hat – instead of telling the reader "well, this and that is a function which solves the issue at hand, what, you didn't know about it? how stupid!", I tried to assume that you haven't read the whole source code of Emacs, or even the whole 1300+ pages of the GNU Emacs Lisp Reference Manual[3] (called the "Elisp reference" from now on). (If you have, why would you need this book anyway?) So, at least in the beginning, whenever I introduce some useful function, I try to show how you, yourself, could find out about it. (Later on you'll already know most of the tricks to find functions doing what you want.) It is not always easy, but Emacs

[1] https://www.gnu.org/software/emacs/manual/html_node/eintr/index.html
[2] https://www.masteringemacs.org/
[3] https://www.gnu.org/software/emacs/manual/elisp.html

really *is* self-documenting and many things are more or less easily discoverable. (Although why some functions were *named* the way they were is still a mystery to me.)

The next thing is that I tried to show real-life examples (as much as possible). Of course, many, many possible enhancements to basic text editing are either present in core Emacs or as packages, so sometimes the code here may replicate what is already available – but nevertheless, I tried to choose ideas that are really useful, and for as wide as audience as possible. This means that the examples are not tied to any specific programming language. Also, I did not touch Org-mode, which is great – and programmable – but not every Emacs user is also an Org-mode user.

Last but not least, I tried to cover as many useful things as possible (even if sometimes superficially), so that you will learn about some common techniques useful in many different situations, like leveraging built-in editing features, using region in your custom commands, writing a minor mode and many others. Sometimes I only mention some feature, variable or function, and if you find it interesting or useful, you can always go to the Elisp reference, the relevant docstrings or even the source code.

The book is intended to be read in order. This means that even if you are interested in e.g. writing a minor mode, you should probably at least skim through the first chapter before diving in to the second one. After all, this is a *textbook*, not a *reference*, and so in later chapters I sometimes use ideas explained earlier without much commentary.

Here is a short breakdown of the chapters. In the first one, we build a function to move the current line up or down, which is quite useful when programming – but also when e.g. typing short itemized lists. The second chapter is devoted to a utility helping with editing texts – a tool to rearrange words in a sentence. It is the longest one in the book, since the tool is going to be quite complicated, and we will use many Emacs features. In the third one, we count lines of code (in a way that can be used for Elisp or other programming languages).

This is a good place to thank a few people who made this book possible. It is obvious that it could never exist without Richard Stallman who wrote the first version of Emacs and numerous people who contributed to it over the decades. It is written in Org-mode, first made by Carsten Dominik and then developed by many other people. Diego Zamboni wrote the Org exporter[4] to convert Org-mode syntax to Markua, expected by Leanpub. Christian Tietze provided a nice idea now incorporated in the sentence-reordering code. Last but not least, my wife and children put up with me writing instead of spending more time with them.

If you like my writing and want to learn more about Emacs and Emacs Lisp, you might consider reading some Emacs blogs. A good portion of my personal blog[5] is also devoted to Emacs.

Just in case, please remember that when I link to some website, mention some person or quote some piece of art, it does not mean that I fully endorse any of those things or share the values of the person mentioned – it means that I consider that information relevant, interesting or maybe just funny in the context.

[4] https://github.com/zzamboni/ox-leanpub
[5] http://mbork.pl

Important note: this book is still subject to change. What you are reading is version 1, but I hope to add at least two more chapters some day in the future, creating version 2. (When? I don't know yet. A Wednesday, perhaps. Maybe next Wednesday, or just one of the Wednesdays. Seriously though, some time in mid-2022 is the most probable time.) Nevertheless, I consider it "complete" in the sense that it is a certain whole, and even if I don't manage to add anything, it can stand on its own as it is now, and the chapters that do exist should not change too much.

Also, you may want to know that I consider releasing this book on some pretty permissive license in the future (most likely one of the Creative Commons ones), but not earlier than perhaps 2023. If you prefer not to pay for it, it might be enough to wait a few years. On the other hand, I owe it to my family that the time I spend *without* them is compensated, e.g. in the form of money needed to sustain our well-being – so I am only willing to release the book for free if and when I attain a reasonable hourly rate (that is, if/when the income it generates divided by the amount of time I spent on it reaches some minimum). This means that paying me *now* is a step towards making it more probable that the book will be available for free *in the future*.

So, let's get into business of programming Emacs!

Moving lines around

Introduction

Let us start simple. We will write a function to move the current line up or down. Of course, "moving up" means "swapping this line with the previous one", and "moving down" means "swapping this line with the next one". This is actually already a solved problem – Emacs has the `transpose-lines` function, but the way it works is conceptually different than "moving the current line up or down" – you need to put the point *between* the lines you want to transpose (or more precisely, on the second one), and after the transposing the point lands *after* the line that was moved down. Therefore, the code we are going to write actually extends Emacs – if only a bit. (Well, there *does* exist a package which allows to move a line or the region up or down, called move-text[6], but our solution is in fact a bit different – definitely simpler, but in some respects better.)

In this chapter, we will first encounter the most basic technique of Elisp programming, which is mimicking the actions of a human editor. In other words, our first version of the `move-line-down` command will be just invoking commands which perform actions the user might do when asked to move the line the point is on below the next one. Then, we are going to make things more complicated for the programmer, but simpler for the user – we are going to use variables to make the user experience less surprising, then we will learn to use prefix arguments, make our command play nice with the undo system, manipulate text in the buffer (deleting and inserting portions of it), and finally use conditionals to support negative prefix arguments.

All the techniques we use in this chapter are pretty basic (with the possible exception of the short section about undo), and are hardly new if you did your homework and read Chassell's *An introduction to programming in Emacs Lisp*. But the main point of this chapter is something else. While any seasoned Emacs programmer knows most of the things used here by heart, a beginner does not, and the recurring theme in every section of this chapter is how one can learn about functions and variables we need without reading over thousand pages of the Elisp reference. We will learn how to use that manual to quickly find what we need, and also to use the various `apropos-...` and `describe-...` commands Emacs has to offer to find out how to do stuff in Elisp.

The first attempt will just use `transpose-lines`. Saying `M-x transpose-lines` shows that this is not what we want – that function swaps the current line with the one above. So, we need to move *the point* one line down before calling it. How to do that? Well, in normal, interactive use of Emacs, we would just press the down arrow or `C-n`. Let's check how Emacs does that by pressing `C-h k C-n`. It tells us that `C-n` (or `<down>`) runs the command `next-line`, but before presenting us with the documentation string (or *docstring* for short) of the command, it tells us that this function should

[6] https://github.com/emacsfodder/move-text

only be used interactively, and that if we want to use it in Elisp code, we should probably use `forward-line` instead.

Note that when Emacs tells us that you shouldn't use `next-line` in Elisp code, you should probably heed that advice – unless you really know what you're doing. Also, notice that this is a very useful piece of information. One thing you should always do when coding Elisp is asking Emacs about various functions, variables etc. – it really *is* very helpful with that. Also, it is a *very* good idea to install Emacs *with sources* on your machine. This way, when you look at the description of a function provided by one of the `describe-...` commands, you'll also see the name of the file it is defined in, and clicking on that name – or pressing `RET` with point on it – will take you directly to the place in Emacs sources where you can see and study the source code of any function. While we are at that, remember that you can move between "buttons" in the `*Help*` buffer with `<tab>` and `S-<tab>` (that is, "shift" and `<tab>`) way faster than with the mouse. In fact, the `*Help*` buffer is always set up in `help-mode`, which contains more useful commands and keybindings – to learn them, you can e.g. press `C-h m`, switch to the `*Help*` buffer that pops up and press `C-h m` again. That way, you will see the docstring of `help-mode`, together with lots of other useful information, including the key bindings defined by that mode. The most useful (besides the ones I already mentioned) are probably `SPC`, `DEL` (i.e., the "backspace" key), `l` and `r`. Also, you might want to check out the Helpful[7] Emacs package (available on MELPA[8]), which contains much more powerful alternatives to many built-in `describe-...` commands.

The first, naive approach to moving a line down

So, we are going to use `forward-line` as we are told.

```
1  (defun move-line-down ()
2    "Move the current line down."
3    (interactive)
4    (forward-line 1)
5    (transpose-lines 1))
```

Note that I could just say `(forward-line)`, since – while it takes one argument (the number of lines to go forward) – it is optional and defaults to one.

Unfortunately, this command does not work very well – it would be nicer if it left the point in the line being moved. That way, calling it more times would move the same line further down. That is easy, it is enough to say `(forward-line -1)` at the end.

[7] https://github.com/Wilfred/helpful
[8] https://melpa.org/

```
(defun move-line-down ()
  "Move the current line down."
  (interactive)
  (forward-line 1)
  (transpose-lines 1)
  (forward-line -1))
```

It is much better now – you can call it a few times and move a line further down. (This is especially nice if you decided to bind it to same key, e.g. `M-<down>`. You can use the `global-set-key` command for that.)

Preserving the column

Of course, there is always (or almost always) room for improvement. Sometimes it is important to know where to stop – coding little editing utilities like this is about making your life easier and saving time, and it won't save you much time if you spend several hours on one little function. Here, however, we are learning, so let's explore some of the possibilities.

One thing is that it would really be nice if the point did not move to the beginning of the line moved. (This is because how `forward-line` works.) It turns out that it is a bit tricky. Using `next-line` (the interactive one we were explicitly told not to use) does not help at all, since `transpose-lines` *also* moves point to the beginning (in fact, it calls `forward-line` at the end). One thing we could do would be to store the position of the point in the line in a temporary variable and move that many characters to the right after performing the moving. The question is, how do we know the position of the point in the line? One way to learn that would be to recall that there is a command which displays that – `C-x =`, or `what-cursor-position`. Let's look at its code – press `C-h k C-x =`, go to the `*Help*` buffer and press the button saying `simple.el` (either with the mouse or with `RET`). The source code for `what-cursor-position` is pretty long and complicated, but near the end there is a call to `message`, and it turns out that the "column" information is taken from the `col` variable. Now, it is enough to go there and type `C-r col` to start looking for its definition, and we can quickly see that it is assigned the value returned by the function `current-column` in a `let`.

The only thing we don't know yet is how to move to the column whose number we store. A naive version would be to call `forward-char` with the number, and it would probably work – but there is a better way. You can say `M-x apropos-function column RET` and Emacs will show you all functions (and macros, for that matter) whose names match the regex `column` – since this regex only contains non-special characters, this means just "contain the word «column»". (One tip is that if you get too many results this way – for instance, in my Emacs it gives more than 140 hits – you may want to repeat it in a fresh Emacs session, started with `emacs -Q`. This way you will only get the results from stock Emacs – no your customizations, no packages etc. In my Emacs it gave 31 hits, which is much more manageable to look through manually.) And indeed, while there is no `goto-column` function, there is `move-to-column` which does exactly what we want (go figure – there is `goto-line`, but `move-to-column` – Emacs function names are sometimes a mess...)

Another way to find the `move-to-column` function would be to assume that most probably, Emacs has a *command* to move to a given column, and look for it in the Emacs manual. Indeed, it is mentioned in the "Changing the Location of Point" section.

Yet another way would be to recall that `M-g M-g` moves to a *line* with the given number, and hope that the `M-g` prefix has something to move to a column – then, pressing `M-g C-h` shows all bindings starting with `M-g`, and indeed `M-g TAB` is what we are looking for.

Knowing that, we can code yet another version of `move-line-down`.

```
(defun move-line-down ()
  "Move the current line down."
  (interactive)
  (let ((position-in-line (current-column)))
    (forward-line 1)
    (transpose-lines 1)
    (forward-line -1)
    (move-to-column position-in-line)))
```

Moving by more lines

Let us now extend this function and make it accept a prefix argument to move the current line *that many* lines down. For that to work, we are going to add an argument to our command, and modify the `interactive` clause accordingly. Thing is, providing arguments to commands is tricky business – they may be numbers (given via `C-u`, like in the case of numerous commands accepting a *count*), filenames (like in `C-x C-f`), buffer names and many, many more. And, there is more than one way to teach a command to get the values of its arguments from the user. The simplest one, however, is to provide `interactive` with a special string, describing (using special codes) where to get those values from. Basically, for each argument you should provide a one-letter code describing its "nature", optionally followed by a prompt (for arguments whose values are read using the minibuffer). In addition, that string may have one or more special characters at the beginning, giving Emacs additional information about the command. See the manual node Defining Commands[9] for the whole (pretty long) story – for now, we need only two pieces of information. To use the *numeric prefix argument*, we need the `p` code. Note that this is lower case "p" – upper case variant means the *raw prefix argument*, which we will discuss later. When using the "numeric" variant, bare `C-u` is translated to 4 to the power of n (where n is the number of times `C-u` was pressed), no prefix argument is translated to 1, and an isolated minus sign is translated to -1 – and this is precisely what we need to interpret the prefix argument as a "count"-type parameter. Also, we are going to begin the string with an asterisk (`*`), which makes Emacs signal an error if the buffer the command was called in is read-only (which makes sense for commands that change the buffer contents).

Of course, all this is only half of the story – apart from properly *getting* the argument, we still need to *use* it. Checking the docstring for `transpose-lines` again we learn that it also accepts a numeric

[9] https://www.gnu.org/software/emacs/manual/html_node/elisp/Defining-Commands.html

argument (and in fact, its `interactive` form is exactly the same as we will use for our command), so we are lucky and can do this.

```
(defun move-line-down (count)
  "Move the current line down."
  (interactive "*p")
  (let ((position-in-line (current-column)))
    (forward-line 1)
    (transpose-lines count)
    (forward-line -1)
    (move-to-column position-in-line)))
```

Making `move-line-down` work with undo

And that is almost the end of the story, at least with this version of `move-line-down`. Unfortunately, it is still not perfect. One thing that bothers me a bit is that our function behaves very strangely when provided a numeric argument of zero. (Try it out. Can you see what happens?) Another is that undoing works, but leaves the point in a strange place. This is easy to fix. Going to the "Undo" section in the Elisp reference tells us about the `buffer-undo-list` variable, which is a list containining information needed to perform, well, undo. One possible value that can be put (or `push`ed, really) onto it is an integer, which means that undo should move point to that position.

```
(defun move-line-down (count)
  "Move the current line down."
  (interactive "*p")
  (let ((position-in-line (current-column)))
    (push (point) buffer-undo-list)
    (forward-line 1)
    (transpose-lines count)
    (forward-line -1)
    (move-to-column position-in-line)))
```

To be fair, `buffer-undo-list` is a pretty advanced concept, and putting it into the first chapter is perhaps a bit controversial – but having an actually useful function is what we really want here. There is a certain trade-off when extending Emacs – oftentimes, either the functions we write are simple, or they are designed to work well with various Emacs "subsystems". Emacs has accumulated *a lot* of stuff over the years, and trying to make your functions play well with *all* of them is pretty daunting. Don't even try to do that – code something that works for you, and if you encounter an unexpected interference between your code and some more obscure feature, decide if you want to (and can) support it. The undo feature, however, while a bit complicated from the coding perspective, is such a basic part of the Emacs experience that making your code play well with it is probably *always* worth the effort.

And now for a completely different approach

Now, instead of fixing the zero-argument behavior, let us step back a bit and reconsider our approach to this function. About half of the issues we had with this approach was that we were so intent on using `transpose-lines` – but in order to do it, we had to move point to a suitable line, then move it to a suitable column, mess around with the undo list, and even then the zero-argument behavior is not what we want. Why not implement everything from scratch instead of relying on `transpose-lines`, which neither does *exactly* what we need nor is a simple function to work with?

So, back to square one. How would we move a line down as human editors, working in Emacs? (This is actually a very important question. Many Emacs extensions can start as some simple code replicating the actual actions of a human user.) One way to do it would be to kill the current line, go down and yank it. We could use the approach of "just coding in Elisp the action a human would perform" and do exactly this, but in fact this is not a good approach – that way, our command would "pollute" the kill ring with the line we're at. Remember, when a user issues a command like `move-line-down`, they will most probably *not* think about it as "killing a line and yanking it below". If they try to yank something later, having the line moved earlier would be pretty surprising.

However, this is not a problem. If we go to the chapter "Text" of the Elisp reference – specifically, the "Deletion" section, we are going to learn about a function which is (in a sense) a programmatic equivalent of killing. `delete-and-extract-region` takes the region between two given positions, deletes it, and returns the string it deleted – all that without touching the kill ring.

Note: if you look at the "Buffer Contents" section, you'll learn about other functions which could be useful here, the `buffer-substring-*` ones. And this is the place where you could start thinking. Should we use `buffer-substring` here? Or maybe `buffer-substring-no-properties`, and what are *text properties* anyway? Or maybe we should heed the warning in the description of `filter-buffer-substring` ("Lisp code should use this function instead of `buffer-substring`, `buffer-substring-no-properties`, or `delete-and-extract-region` (...)")? Here is my (unofficial, since I'm not part of the Emacs developer team) advice. Do. Not. Care. If the simplest function (like `delete-and-extract-region` in this case) works for you, go for it. If it's the right choice, then great. If it's the wrong choice, one day it won't work for you, and then you will have a minor moment of frustration, a need for undo (which works great in Emacs) and possibly a dive into the docs or a question on the help-gnu-emacs[10] mailing list. And if you share your code with other people, maybe it won't work for someone else, so you'll get a bug report or feature request – the rest of the cycle is the same. In either case, your code should converge over time to a "good enough" state.

Don't get me wrong – I'm not in favor of sloppy coding. But Emacs is a complicated beast, and it may be the case that there are two functions which seem to do the same thing, and you read about the difference in the docs and you can't make that out because it talks about some gibberish like "text properties". But if you don't even know what "text properties" are, chance is that you don't care for them, so either function would be good for you. And if it turns out that you *do* care for them after all, only you didn't know the name, but your code doesn't work as you expected – now

[10] https://mail.gnu.org/mailman/listinfo/help-gnu-emacs

that's even better, since you'll learn something new along the way. (The alternative would be to stop, study what text properties are, think carefully if they are what you need and code accordingly. This is a great idea if you have time for it, which is not always the case…)

So, coming back. As we said, instead of using `buffer-substring` to get some portion of the buffer and store it *somewhere* temporarily (only not in the kill ring, which is a feature more for human users and not Elisp code – we'll get to the "somewhere" shortly) and then deleting that exact same portion with `delete-region` (which, by the way, you can also find in the "Deletion" section of the same chapter), we can use `delete-and-extract-region` which combines both. So, what we want to do now is basically this: find out where the beginning and end of the current line are, `delete-and-extract-region` the text between them, store it somewhere, move to the next line and insert it back there.

We can almost write the code now, except for one thing. How do we get the position of the beginning of the current line? The natural thing would be to look for functions containing the strings "begin" and "line". This is easy: just say `M-x apropos-function RET begin.*line RET`. (If you don't know regular expressions – and `begin.*line` is a regular expression, or regex – go learn some basics about them now, seriously. They are really cool and *extremely* useful.) If you use some clever completion engine instead of Emacs bare-bones completion mechanism (I use Ivy, there exist others, like – in alphabetical order – Helm, Icicles and Ido), you'll probably have a similar thing baked into `C-h f`, too. Anyway, you can see all sorts of functions which *go* (as in: move the point) to the beginning of the current line (for various ways of understanding the words "beginning" and "current line"), but no functions for *getting the position of the beginning of the current line*. (Well, there is also `bolp`, which is not easy to find in this way because of its cryptic name, but it still isn't it.) So, what do we do? Well, there are two more ways of finding such a function, both being rather a long shot, but they shouldn't take us more than a minute, so why not? One is looking at help for some *other* function, doing a "similar" thing – e.g., `point` (which gives us the *current* position, not the position of the beginning of the *current line* – but it seems somehow related). In fact, `C-h f point RET` pops up a window which says – among others – something like "Other relevant functions are documented in the buffer group", and the word "buffer" is blue and underlined. Going to it with `TAB` and pressing `RET` shows yet another buffer, in `shortdoc-mode` (seemingly a bit underused in Emacs), which contains short information about various related functions, together with examples and links to documentation. There are a few position-related functions there apart from `point`, and one of them is… `line-beginning-position`, which wxactly what we need!

The second possible way is (now obviously) to not only look for functions matching `begin.*line`, but also `line.*begin`. Changing the order of "keywords" when looking for built-in functions is another useful tip.

So, let's write the next version of `move-line-down`. While storing something temporarily when *using* Emacs usually means the kill ring (or perhaps registers), in Elisp *code* it usually means a local variable – in other words, `let` (or `let*`).

```
(defun move-line-down ()
  "Move the current line down."
  (interactive)
  (let ((current-line (delete-and-extract-region
                       (line-beginning-position)
                       (line-end-position))))
    (forward-line 1)
    (insert current-line)))
```

Well, if you try it out, you'll see that it doesn't work. Of course it can't – we moved the line without the end-of-line character, i.e., we extracted one character too few. There are many ways to remedy that, but the simplest one is probably just to add one to `(line-end-position)`. There is a built-in function to add one to something, `1+`, or we could use the normal addition function, `+`. And while we are at that, let's reinstate the code moving the point to the right column.

```
(defun move-line-down ()
  "Move the current line down."
  (interactive)
  (let ((position-in-line (current-column))
        (current-line (delete-and-extract-region
                       (line-beginning-position)
                       (1+ (line-end-position)))))
    (forward-line 1)
    (insert current-line)
    (move-to-column position-in-line)))
```

Well, still not good – the point jumped to a seemingly random location, one line too far. When you think about it, it becomes obvious, too – after `inserting current-line`, which now contains the end-of-line character, the point is effectively in the *next* line and not in the one inserted. We could say `(forward-line -1)` to remedy that, but we can also tell Emacs to get back where we were after inserting `current-line`. In fact, "do something and get back where you were" is such a common thing to do when coding in Elisp that there is a special idiom for that: `(save-excursion ...)`. We will encounter it many times later, so let's just recall that it remembers the current point position and the current buffer, evaluates the code denoted by `...` above, and restores the buffer and point. This means that wrapping `(insert current-line)` in `(save-excursion ...)` is enough. (Note: sometimes you may want to restore more things, like e.g. the value of *mark*, from before your code was evaluated – there are other `save-...` constructs, like `save-mark-and-excursion`, `save-match-data`, `save-restriction` and a few more. They seem to be less frequently used, but they are useful, too. A nice exercise now would be to find out about all of them. The fact that they are named similarly to commands for actually *saving buffer to the disk* is not helpful, but there aren't *that* many of them anyway.)

And yet another approach

However, there is another way. Instead of remembering in what column the point was (and the point's position is obviously lost when the current line is deleted), we can flip the whole idea and delete the *next* line and reinsert it above. That way, we won't even have to move the point at all.

```
(defun move-line-down ()
  "Move the current line down."
  (interactive)
  (save-excursion
    (forward-line 1)
    (let ((region-to-move-up
           (delete-and-extract-region
            (point)
            (save-excursion
              (forward-line 1)
              (point)))))
      (forward-line -1)
      (insert region-to-move-up))))
```

And this seems the best idea we had so far – alas, it contains a serious bug. Can you spot it? I couldn't, I only discovered it by experimenting, and even then it was not obvious for me what was happening… This is a common theme in programming – it is often just complicated enough we humans have problems wrapping our heads around it. Here is the problem: it doesn't work when the point is *at the beginning of the line* when our command is invoked. And after a while it becomes fairly obvious why is that so – in such a case, we insert the *next* line basically *at point* (by which I mean at the position of the point before the command was invoked), so `save-excursion` is fooled and (more or less) thinks that instead of inserting *before* the current line, we are *prepending* to it, and lefts the point (as instructed) at the beginning of the current line – but now it thinks that the "current line" is two lines, the one that used to be the next one and the proper "current line".

There are a few ways to mitigate this, but the best one I found is this. Instead of `insert`, we can use `insert-before-markers`. This is a function which is probably much less known and much less often used than `insert`. It does exactly what it says on the tin – it inserts the text(s) given at point (like `insert`), but if some *marker* (not only point, but also e.g. mark and other remembered positions, including everything on the mark ring – you can read about markers in the Elisp Reference, we will not deal with them in this book) points at the very same position, it is moved along with point (unlike what `insert` does).

A legitimate question you may ask now is *how am I supposed to know* about `insert-before-markers`? The short answer is, well, you aren't, at least not necessarily *that* easily. While Emacs *is* very well-documented, not all functions are readily discoverable. You *could* say `M-x apropos-function RET insert RET` (about a hundred hits!), or even `M-x apropos-function`

`RET ^insert RET` (about 35 hits), but for that you should at least *suspect* that such a function exists. Probably a better idea (which I admit I didn't try, and now that I know Elisp pretty well it's too late for me;-)) is this: if you have spare 30 minutes, go through the Elisp reference, visit every chapter and read one or two paragraphs at the beginning. This should give you a rough idea about what's in there in general. If you then need something related to e.g. markers, you can go to the "Markers" chapter and do a similar exercise with its sections. You will then learn about "marker insertion types", and `insert-before-markers` is also mentioned there. (Of course, who knew it is there you should be looking at? Well, I said Emacs *is* complicated…) Another way to find out about that function is to study the manual section "Excursions", which talks about `save-excursion` and some related constructs. At the end of that section there is a warning about precisely the situation we ran into, with a link to the section about marker insertion types – which then gives the hint about `insert-before-markers`.

I guess that the moral of the story is that if some Elisp construct does not work as you expect (or as you wanted it to work!), you should go to the Elisp reference and read about it – chances are that the relevant section somehow deals with the issue you're having. And you can ask on the mailing list, too.

Changing `insert` to `insert-before-markers` is easy, so let's up the ante a bit and add three more things. First of all, we will reinstate the `undo` fix as we did previously. Then, we will generalize our function to moving by more than one line down (again). Finally, let us make it work nicer with code. When moving a line of code down, it may happen that it will end up on a different level in the syntax tree – it might land in a then-branch of a conditional, or fall out of a loop etc. This means that we might want to reindent it to match its new surroundings. This sounds pretty complicated – if we were to do it by hand, it could depend on the programming language used, and even individual style of a person using it. However, Emacs (obviously) has good support for both these things (at least in theory – in practice the support quality varies from language to language…), namely major modes (to work with different languages) and individual customizations (to align with the user's personal style). The only thing we need to know is what function indents the current line according to those.

In my Emacs, `M-x apropos-function RET indent RET` gives a whopping 255 hits (no kidding!). If I start it with `emacs -Q`, the number is down to 74 – still quite a lot. Instead of going through all of these and try to find the suitable one, let's go to the manual. Happily, the table of contents for the Elisp reference has something called "the detailed node listing", which apparently contains links to *all* the individual nodes of the whole book. This means that we might find what we need with the good ol' `C-s` – though probably a better way would be to use `m` (for `Info-menu`). Either way, we should arrive to the node called "Mode-Specific Indent" (possibly via the node "Auto-Indentation"), where we can find the function (actually, command) `indent-according-to-mode`.

Another tip is to use the index – when in an Info buffer with the Elisp reference, press `i` and type e.g. `indentation`. The command `Info-index` is smart enough to tell you if the entry you selected points to more than one place in the index, and if so, it gives you a nice tip: pressing `,` (the comma) moves you to the next one. Even better, there is the `Info-virtual-index` command (bound to capital `I`), which constructs a "virtual node" with a menu of all the places the given index entry points to. Either way, the first index entry for "indentation" is the "Indentation" section, which contains "Mode-specific

indent" as its second subsection.

And here is the code.

```
(defun move-line-down (count)
  "Move the current line down."
  (interactive "*p")
  (save-excursion
    (push (point) buffer-undo-list)
    (forward-line 1)
    (let ((region-to-move-up
           (delete-and-extract-region
            (point)
            (save-excursion
              (forward-line count)
              (point)))))
      (forward-line -1)
      (insert-before-markers region-to-move-up)
      (indent-according-to-mode))))
```

Supporting different prefix arguments

Works great, right? Well, not necessarily. Giving a numerical argument works fine *if the argument is positive*. If the argument is negative, moving the line works ok, but the point ends up in the wrong place.

This is easy to fix – instead of `(forward-line -1)`, we may need `(forward-line 1)` for negative arguments – but before we implement that, let us stop for a moment and discuss prefix arguments. If we write a command and decide that it should support them (and we should *always* consider supporting them if that makes sense!), it is a very good idea to consider a few cases.

The simplest case is using the prefix argument as a flag – then, `M-x our-command` does one thing, but `C-u M-x our-command` does something a bit different. For instance, one of the most important commands every Emacs user needs to learn at some point (usually at the very beginning!), `save-buffers-kill-terminal` (bound to `C-x C-c` by default) asks if it should save unsaved files, but does it without asking if prefixed by `C-u` (or, actually, *any other prefix argument*). This is very easy to support – you can just define your command to have a parameter and declare that it is going to use the "raw prefix argument" for that: `(interactive "P")` (note: that is an upper-case `P`). This means that it is nil if there was no prefix argument and some other value otherwise. (You can check the "Prefix Command Arguments" node in the Elisp reference for details about the possible "other values".)

A slightly more complicated case is when the prefix argument should be treated as an integer. This is usually the case when we want to treat it as a "count", making our command perform some action

more than one time if prefixed with an integer. In that case, say `(interactive "p")` (note: this is a lower-case `p`). Then – as we already know – if no prefix argument was supplied, the argument will get the value of one, if the prefix argument was `C-u -` (i.e., just a minus sign), it will be minus one, if it was `C-u` repeated `N` times, it will be four to the power of `N`, and in all other cases it will be the provided numeric argument.

The most involved case is when we want to actually distinguish between e.g. `C-u 4` and plain `C-u`, or between `C-u 16` and `C-u C-u`, or between `C-u` and `C-u C-u` etc. In that case we can use `(interactive "P")` again, but look carefully at what the argument value is. It can be nil (no argument), an integer (a numeric prefix argument), the symbol `-` (remember to write `'-` when you mean the symbol itself and not the value of a *variable* named "-"!) for `C-u -`, or a list like `(4)`, `(16)`, `(64)` etc. for one, two, three etc. `C-u`'s.

In our case, we need just a count (as we did previously), so `(interactive "p")` is just fine – and we are going to use the asterisk again so that the command is not even run in read-only buffers.

```
(defun move-line-down (count)
  "Move the current line down."
  (interactive "*p")
  (unless (zerop count)
    (save-excursion
      (push (point) buffer-undo-list)
      (forward-line (if (> count 0) 1 0))
      (let ((region-to-move-up
             (delete-and-extract-region
              (point)
              (save-excursion
                (forward-line count)
                (point)))))
        (forward-line (if (> count 0) -1 1))
        (insert-before-markers region-to-move-up)
        (indent-according-to-mode)))))

(defun move-line-up (count)
  "Move the current line up."
  (interactive "*p")
  (move-line-down (- count)))
```

Note that we check if `count` is equal to zero, which is kind of strange – our function does not make too much sense then – but the user *can* say `C-u 0 M-x move-line-down RET`, and we *should* do something reasonable then – like not changing anything. The `zerop` function just tests if its argument is equal to zero, and `unless` is more or less a shorthand for `(if (not ...) (progn ...))` – in other words, it evaluates the forms inside if the given condition is nil. (There is also `when`, which works the same but checks for non-nil-ness of the condition. We will meet it later.) Also, since we decided to support

negative arguments, it is useful to add another function, `move-line-up`, so that both can be bound to some keys easily.

Summary

In this chapter, we learned a whole lot of things. Probably the most important lesson, however, is how to learn about Emacs functions to perform various practical tasks.

Elisp, at its heart, is a very simple language (it is a Lisp, after all!). It *does* have advanced features and dark corners, but those are outside the scope of this book (and you can go a *long* way without ever looking at them). The most intimidating part when learning to code actually useful tools is learning the "library" – in other words, knowing what functions are available. Luckily, Emacs is more than happy to tell us that – we just have to ask nicely;-). The places to look are:

- the docstrings of commands you use to perform a similar task by hand (displayed with `C-h k` or `C-h f`),
- links in those docstrings (sometimes leading to other, similar functions and sometimes to a `shortdoc` buffers),
- output of commands like `M-x apropos-function` with a regex constructed from some keywords (this sometimes requires a bit of luck, since the naming of Emacs library functions can be a bit wobbly at times, and it is worth trying them out in various order), and
- Elisp reference, especially its detailed node list and index (for quick access of those, it's worth remembering about the `m`, `i` and `I` keys).

Reordering parts of a sentence

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Modal versus non-modal design

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

The set-up command

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Constructing the reordered sentence

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Bringing the reordered sentence back

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Fixing a bug with region boundaries inside words

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Defaulting to the current sentence

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Introducing a mode

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

A shorter definition of the mode's keymap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Showing the key for a word

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Generating the list of keys to use

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Showing the keys for all words in the region

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Making the word selection easier – handling the keys

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Dealing with punctuation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Avoiding duplication in code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Dealing with capitalization

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Undoing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Marking words already copied and making copying faster

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Implement a more robust undo feature

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Final touches

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Package information

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Counting lines of code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

The skeleton of the counting command

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Counting non-blank lines

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Let's get more abstract...

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Skipping comments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.

Afterword

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/hacking-your-way-emacs.