How to Make an Emacs Minor Mode

February 06, 2013
nullprogram.com/blog/2013/02/06/

An Emacs buffer always has one major mode and zero or more minor modes. Major modes tend to be significant efforts, especially when it comes to automatic indentation. In contrast, minor modes are often simple, perhaps only overlaying a small keymap for additional functionality. Creating a new minor mode is really easy, it's just a matter of understanding Emacs' conventions.

Mode names should end in `-mode` and the command for toggling the mode should be the same name. They keymap for the mode should be called *mode*`-map` and the mode's toggle hook should be called *mode*`-hook`. Keep all of this in mind when picking a name for your minor mode.

There are a number of other tedious issues that need to be taken into account when manually building a minor mode. The good news is that no one needs to worry about most of it! Lisp has macros for cutting down on boilerplate code and so there's a macro for this very purpose: `define-minor-mode`. Here's all it takes to make a new minor mode, `foo-mode`.

```
(define-minor-mode foo-mode
  "Get your foos in the right places.")
```

This creates a command `foo-mode` for toggling the minor mode and a hook called `foo-mode-hook`. There's a strange caveat about the hook: it's not immediately declared as a variable. My guess is that this is some archaic optimization which now exists as bad design. The hook function `add-hook` will create this variable lazily when needed and the function `run-hooks` will ignore hook variables that don't yet exist, so it doesn't get tripped up by this situation. So despite its strange initial absence, the new minor mode *will* use this hook as soon as functions are added to it.

Minor Mode Options

This mode doesn't *do* anything yet. It doesn't have its own keymap and it doesn't even show up in the modeline. It's just a toggle and a hook that's run when the toggle is used. To add more to the mode, `define-minor-mode` accepts a number of keywords. Here are the important ones.

- `:lighter`: the name, a string, to show in the modeline
- `:keymap`: the mode's keymap
- `:global`: specifies if the minor mode is *global*

The `:lighter` option has one caveat: it's concatenated to the rest of the modeline without any delimiter. This means it needs to be prefixed with a space. I think this is mistake, but we're stuck with it probably forever. Otherwise this string should be kept short: there's generally not much room on the modeline.

```
(define-minor-mode foo-mode
  "Get your foos in the right places."
  :lighter " foo")
```

New, empty keymaps are created with `(make-keymap)` or `(make-sparse-keymap)`. The latter is more efficient when the map will contain a small number of keybindings, as is the case with most minor modes. The fact that these separate functions exist is probably another outdated, premature

optimization. To avoid confusing others, I recommend you use the one that matches your intended usage.

The keymap can be provided directly to `:keymap` and it will be bound to `foo-mode-map` automatically. I could just put an empty keymap here and define keys separately outside the `define-minor-mode` declaration, but I like the idea of creating the whole map in one expression.

```
(defun insert-foo ()
  (interactive)
  (insert "foo"))

(define-minor-mode foo-mode
  "Get your foos in the right places."
  :lighter " foo"
  :keymap (let ((map (make-sparse-keymap)))
            (define-key map (kbd "C-c f") 'insert-foo)
            map))
```

The `:global` option means the minor mode is not local to a buffer, it's present everywhere. As far as I know, the only global minor mode I've ever used is YASnippet[1].

Minor Mode Body

The rest of `define-minor-mode` is a body for arbitrary Lisp, like a `defun`. It's run every time the mode is toggled off or on, so it's like a built-in hook function. Use it to do any sort of special setup or teardown, such hooking or unhooking Emacs' hooks. A likely thing to be done in here is specifying *buffer-local variables*.

Any time the Emacs interpreter is evaluating an expression there's always a *current buffer* acting as context. Many functions that operate on buffers don't actually accept a buffer as an argument. Instead they operate on the current buffer. Furthermore, some variables are buffer-local: the binding is dynamic over the current buffer. This is useful for maintaining state relevant only to a particular buffer.

Side note: the `with-current-buffer` macro is used to specify a different current buffer for a body of code. It can be used to access other buffer's local variables. Similarly, `with-temp-buffer` creates a brand new buffer, uses it as the current buffer for its body, and then destroys the buffer.

For example, let's say I want to keep track of how many times `foo-mode` inserted "foo" into the current buffer.

```
(defvar foo-count 0
  "Number of foos inserted into the current buffer.")

(defun insert-foo ()
  (interactive)
  (setq foo-count (1+ foo-count))
  (insert "foo"))

(define-minor-mode foo-mode
  "Get your foos in the right places."
  :lighter " foo"
  :keymap (let ((map (make-sparse-keymap)))
            (define-key map (kbd "C-c f") 'insert-foo)
            map)
  (make-local-variable 'foo-count))
```

The built-in function `make-local-variable` creates a new buffer-local version of a global variable in the current buffer. Here, the buffer-local `foo-count` will be initialized with the value 0 from the global variable but all reassignments will only be visible in the current buffer.

However, in this case it may be better to use `make-variable-buffer-local` on the global variable and skip the `make-local-variable`. The main reason is that I don't want `insert-foo` to clobber the global variable if it happens to be used in a buffer that doesn't have the minor mode enabled.

```
(make-variable-buffer-local
 (defvar foo-count 0
   "Number of foos inserted into the current buffer."))
```

A big advantage is that this buffer-local intention for the variable is documented globally. This message will appear in the variable's documentation.

> Automatically becomes buffer-local when set in any fashion.

Which method you use is up to your personal preference. The Emacs documentation encourages the former but I think the latter is nicer in many situations.

Automatically Enabling the Minor Mode

Some minor modes don't have any particular major mode association and the user will toggle it at will. Some minor modes only make sense when used with particular major mode and it might make sense to automatically enable along with that mode. This is done by hooking that major mode's hook. So long as the mode follows Emacs' conventions as mentioned at the top, this hook should be easy to find.

```
(add-hook 'text-mode-hook 'foo-mode)
```

Here, `foo-mode` will automatically be activated in all `text-mode` buffers.

Full Code

Here's the final code for our minor mode, saved to `foo-mode.el`. It has one keybinding and it's easily open for users to define more keys in `foo-mode-map`. It also automatically activates when the user is editing a plain text file.

```
(make-variable-buffer-local
 (defvar foo-count 0
   "Number of foos inserted into the current buffer."))

(defun insert-foo ()
  (interactive)
  (setq foo-count (1+ foo-count))
  (insert "foo"))

;;;###autoload
(define-minor-mode foo-mode
  "Get your foos in the right places."
  :lighter " foo"
  :keymap (let ((map (make-sparse-keymap)))
            (define-key map (kbd "C-c f") 'insert-foo)
            map))

;;;###autoload
(add-hook 'text-mode-hook 'foo-mode)

(provide 'foo-mode)
```

I added some autoload declarations and a `provide` in case this mode is ever distributed or used as a package. If an autoloads script is generated for this minor mode, a temporary function called `foo-mode` will be defined whose sole purpose is to load the real `foo-mode.el` and then call `foo-mode` again with its new definition, which was loaded overtop the temporary definition.

The autoloads script also adds this temporary `foo-mode` function to the `text-mode-hook`. If a `text-mode` buffer is created, the hook will call `foo-mode` which will load `foo-mode.el`, redefining `foo-mode` to its real definition, then activate `foo-mode`.

The point of autoloads is to defer loading code until it's needed. You may notice this as a short delay the first time you activate a mode after starting Emacs. This is what keeps Emacs' start time reasonable despite having millions of lines of Elisp virtually loaded at startup.

tags [emacs]

1. https://github.com/capitaomorte/yasnippet