daviwil / emacs-from-scratch

Code Issues 44 Pull requests 8 Actions Wiki Security Ins

emacs-from-scratch / show-notes / Emacs-Lisp-04.org

bkaestner Enhance defcustom definitions 7cc86d3 · 4 years ago

485 lines (295 loc) · 14.3 KB

Preview Code Blame Raw

Defining Variables and Scopes

What will we cover?

- The many ways to set the value of a variable
- Defining variables
- Creating buffer-local variables
- Understanding variable scopes
- Creating variable scopes with `let`
- Defining and setting customization variables

We'll briefly apply what we've learned to our project!

Review: what is a variable?

A variable is an association (binding) between a name (more specifically a symbol) and a value.

- `'tab-width -> 4`

In Emacs there are many ways to define these bindings and change the values that they are associated with.

There are also a few ways to change how the value for a particular variable is resolved!

Setting variables

You're probably familiar with this, a large part Emacs configuration is setting variables!

We usually do this with `setq`:

```
(setq tab-width 4)
```

But what is this really doing?

```
(set 'tab-width 4)

(set 'tab-width (- 4 2))
```

The variable does not have to exist or be pre-defined!

```
(set 'i-dont-exist 5)

i-dont-exist
```

`setq` is just a convenience function for setting variable bindings. It removes the need for quoting the symbol name!

```
(setq mouse-wheel-scroll-amount '(1 ((shift) . 1)))
(setq mouse-wheel-progressive-speed nil)
(setq mouse-wheel-follow-mouse 't)
(setq scroll-step 1)
(setq use-dialog-box nil))
```

It also allows you to set multiple variables in one expression:

```
(setq mouse-wheel-scroll-amount '(1 ((shift) . 1))
      mouse-wheel-progressive-speed nil
      mouse-wheel-follow-mouse 't
      scroll-step 1
      use-dialog-box nil)
```

Defining variables

Setting global variables is easy, but what if you want to document the purpose of a variable?

This is what `defvar` is for. It basically allows you to create a variable binding and assign a documentation string to it:

```
(setq am-i-documented "no")

(defvar am-i-documented "yes"
  "I will relieve my own concern by documenting myself")
```

Why didn't `am-i-documented` show up as "yes"? `defvar` only applies the default value if the binding doesn't already exist!

This is actually useful: packages can define their variables with `defvar` and you can set values for them **before** the package gets loaded! Your settings will not be overridden by the default value.

If you want the default value to be immediately applied while writing your code, use `eval-defun` (`C-M-x`)

In the end, you would use `defvar` when you want to define and document a variable in your configuration or in a package. In most other cases, plain `setq` is sufficient.

Emacs Lisp Manual: Defining Variables

Buffer local variables

You can set the value of a variable for the current buffer only using `setq-local`. Any code that runs in that buffer will receive the buffer-local value instead of the global value!

This is the first example where we see how the value of a variable can be different depending on where it is accessed.

```
(setq-local tab-width 4)
```

Why do this? There are many settings that should only be set per buffer, like editor settings for different programming languages and customization variables for major modes.

If the variable isn't already buffer-local, `setq-local` will make it so, but only for the current buffer!

```
;; some-value doesn't exist yet!
(setq some-value 2)

;; Make it buffer-local
(setq-local some-value 4)

;; Using setq now will only set the buffer-local binding!
(setq some-value 5)

;; A variable may only exist in a particular buffer!
(setq-local only-buffer-local "maybe?")
```

Emacs Lisp Manual: Buffer Local Variables

Making a variable local for all buffers

You can make any variable local for all future buffers with the `make-variable-buffer-local` function:

```
(setq not-local-yet t)
(make-variable-buffer-local 'not-local-yet)
```

If you are writing an Emacs Lisp package and want to provide a buffer-local variable, this is the way to do it!

```
;; Defining a variable with defvar and then making it buffer local
(defvar new-buffer-local-var 311)
(make-variable-buffer-local 'new-buffer-local-var)
```

Setting default values

You might also want to set the default value for a buffer-local variable with `setq-default`:

```
(setq-default not-local-yet nil)

(setq-default tab-width 2
              evil-shift-width 2)
```

```
;; BEWARE!  Unexpected results using buffer-local variables:
(setq-default evil-shift-width tab-width)

;; This will create a variable that doesn't exist
(setq-default will-i-be-created t)
```

Keep in mind that `setq-default` **does not** set the value in the current buffer, only future buffers!

Defining variable scopes

What is a "scope"?

It's a region of your code where a variable is bound to a particular value (or not).

More specifically, the value of `x` can be different depending on where in your code you try to access it!

There are two different models for variable scope in Emacs Lisp, we will discuss this later.

Global scope

So far, we've been using variables that are defined in the "global" scope, meaning that they are visible to any other code loaded in Emacs. A buffer-local variable can be thought of as a global variable for a particular buffer.

Global variables are great for two things:

- Storing configuration values that are used by modes and commands
- Storing information that needs to be accessed by future invocations of a piece of code

Defining a local scope with `let`

Sometimes you just need to define a variable temporarily without "polluting" the global scope. For example:

```
(setq x 0)

(defun do-the-loop ()
  (interactive)
  (message "Starting the loop from %d" x)
  (while (< x 5)
    (message "Loop index: %d" x)
    (incf x))
  (message "Done!"))

(do-the-loop)
```

But what if we run the function again?

We can use `let` to define `x` inside of `do-the-loop`:

```
(defun do-the-loop ()
  (interactive)
  (let ((x 0))
    (message "Starting the loop from %d" x)
    (while (< x 5)
      (message "Loop index: %d" x)
      (incf x))
    (message "Done!")))

(do-the-loop)
```

`x` is bound inside of the scope contained within the `let` expression!

However, what happened to the `x` that we defined globally?

```
(defun do-the-loop ()
  (interactive)
  (message "The global value of x is %d" x)
  (let ((x 0))
    (message "Starting the loop from %d" x)
    (while (< x 5)
      (message "Loop index: %d" x)
      (incf x))
    (message "Done!")))
```

The `x` defined in the `let` overrides the global `x`! Now when you set the value of `x`, you are only setting the value of the local `x` binding.

NOTE: In the examples above, I am using `let` inside of a function definition, but it can be used anywhere! We'll see this in the next section.

Emacs Lisp Manual: Variable Scoping

Defining multiple bindings with `let` and `let*`

Once you start writing code that isn't so trivial, you'll find that you need to initialize a few temporary variables in a function to precalculate some results before running the real function body.

The `let` expression enables you to bind multiple variables in the local scope:

```
(let ((y 5)
      (z 10))
  (* y z))
```

However, what if you want to refer to `y` in the expression that gets assigned to `z`?

```
(let ((y 5)
      (z (+ y 5)))
  (* y z))
```

`let*` allows you to use previous variables you've bound in subsequent binding expressions:

```
(let* ((y 5)
       (z (+ y 5)))
  (* y z))
```

The difference between `let` and `let*` is that `let*` actually expands to something more like this:

```
(let ((y 5))
  (let ((z (+ y 5)))
    (* y z)))
```

Side note: there are a couple of useful macros called `if-let` and `when-let`, we will cover them in another video about helpful Emacs Lisp functions!

Understanding “dynamic” scope

Emacs Lisp uses something called “dynamic scope” by default. This means that the value that is associated with a variable may change depending on where an expression gets evaluated.

It’s easier to understand this by looking at an example:

```
(setq x 5)

;; x is considered a "free" variable
(defun do-some-math (y)
  (+ x y))

(do-some-math 10)     ;; 15

(let ((x 15))
  (do-some-math 10))  ;; 25

(do-some-math 10)
```

The value of `x` is resolved from a different scope based on where `do-some-math` gets executed!

This can actually be useful for customizing the behavior for functions from other packages. We’ve seen this before!

```
  (defun dotfiles-tangle-org-file (&optional org-file)
    "Tangles a single .org file relative to the path in
dotfiles-folder.  If no file is specified, tangle the current
file if it is an org-mode buffer inside of dotfiles-folder."
    (interactive)
   ;; Suppress prompts and messages
    (let ((org-confirm-babel-evaluate nil)
          (message-log-max nil)
          (inhibit-message t))
      (org-babel-tangle-file (expand-file-name org-file dotfiles-folder
```

We didn’t actually change the global value of any of these variables!

The other scoping model in Emacs is called “lexical scoping”. We will cover this and contrast the differences with dynamic scoping in another video.

Emacs Lisp Manual: Variable Scoping

Defining customization variables

Customizable variables are used to define user-facing settings for customizing the behavior of Emacs and packages.

The primary difference between They show up in the customization UI (users can set them without code)

We'll only cover them briefly today because they are a core part of Emacs. I'll make another video to cover custom variables and the customization interface in depth.

Using `defcustom`

The `defcustom` function allows you to define a customizable variable:

```
(defcustom my-custom-variable 42
  "A variable that you can customize")
```

`defcustom` takes some additional parameters after the documentation string:

- `:type` - The expected value type
- `:group` - The symbol that identifies the "group" this variable belongs to (defined with `defgroup`)
- `:options` - The list of possible values this variable can hold
- `:set` - A function that will be invoked when this variable is customized
- `:get` - A function that will be invoked when this variable is resolved
- `:initialize` - A function to be used to initialize the variable when it gets defined
- `:local` - When `t` , automatically marks the variable as buffer-local

There are a few more properties that I didn't mention but you can find them in the manual:

Emacs Lisp Manual: Defining Customization Variables Emacs Lisp Manual: Defining Customization Groups

Setting customizable variables (correctly)

Some variables are defined to be customized and could have behavior that executes when they are changed.

The important thing to know is that `setq` does not trigger this behavior!

Use `customize-set-variable` to set these variables correctly in code:

```
(customize-set-variable 'tab-width 2)
(customize-set-variable 'org-directory "~/Notes)
```

If you're using `use-package` (which I recommend), you can use the `:custom` section:

```
(use-package emacs
  :custom
  (tab-width 2))

(use-package org
  :custom
  (org-directory "~/Notes"))
```

How do I know that a variable is customizable?

The easiest way is to use `describe-variable` (bound to `C-h v`) to check the documentation. If the variable is customizable it should say:

```
"You can customize this variable"
```

NOTE: The Helpful package gives a lot more useful information!

You can also use `custom-variable-p` on the variable's symbol (eval with `M-:`)

```
(custom-variable-p 'tab-width)
(custom-variable-p 'org-directory)
(custom-variable-p 'org--file-cache)
```

Continuing the project

We've covered a lot today so we'll keep the example short this time!

We're going convert a couple of the variables from last time into customizable variables using `defcustom`:

```
(defcustom dotfiles-folder "~/.dotfiles"
  "The folder where dotfiles and org-mode configuration files are store
  :type 'directory
  :group 'dotfiles)

(defcustom dotfiles-org-files '()
  "The list of org-mode files under the `dotfiles-folder' which
contain configuration files that should be tangled"
  :type '(repeat string)
  :group 'dotfiles)

(defun dotfiles-tangle-org-file (&optional org-file)
  "Tangles a single .org file relative to the path in
dotfiles-folder.  If no file is specified, tangle the current
file if it is an org-mode buffer inside of dotfiles-folder."
  (interactive)
 ;; Suppress prompts and messages
  (let ((org-confirm-babel-evaluate nil)
        (message-log-max nil)
        (inhibit-message t))
    (org-babel-tangle-file (expand-file-name org-file dotfiles-folder))

(defun dotfiles-tangle-org-files ()
  "Tangles all of the .org files in the paths specified by the variable
  (interactive)
  (dolist (org-file dotfiles-org-files)
    (dotfiles-tangle-org-file org-file))
  (message "Dotfiles are up to date!"))
```

What's next?

In the next episode we will start discussing the most important extensibility points in Emacs:

- Major and minor modes
- Hooks