Being Productive With Emacs

Part 3

Phil Sung
sipb-iap-emacs@mit.edu
http://stuff.mit.edu/iap/emacs
Special thanks to Piaw Na and Arthur Gleckler
Previously...

- Customizing emacs
  - Setting variables
  - Key bindings
  - Hooks

- Extending emacs with new elisp procedures
  - Simple text manipulation
  - Interactive specifications
This time...

- Extending emacs
  - Advising functions
  - Foundations of elisp
  - More about interactive specifications
  - Manipulating text in emacs
  - Creating a major mode
Advice

• Used to add to any existing function

• Pieces of advice are modular

• Advice vs. hooks

• Advice can be dangerous!
Advice example: previous line

• When `next-line-at-end` is set to `t`, `next-line` on last line of buffer creates a new line

• Create analogous behavior for `previous-line` at beginning of buffer
  – When on first line of buffer, insert a newline before moving backwards
Advice example: previous-line

(defun advice previous-line
  (before next-line-at-end
    (&optional arg try-vscroll))
"Insert new line when running previous-line
at first line of file"
(if (and next-line-add-newlines
  (save-excursion (beginning-of-line)
    (bobp))
  (progn (beginning-of-line)
    (newline))
))
Advice syntax

(defadvice function-to-be-modified
  (where
    name-of-advice
    (arguments-to-original-function))
"Description of advice"
  (do-this)
  (do-that))

where can be before, after, or around
Enabling advice

- (ad-enable-advice 'previous-line 'before 'next-line-at-end)
- (ad-disable-advice 'previous-line 'before 'next-line-at-end)
Activating advice

- (ad-activate 'previous-line)
  - Do this every time advice is defined, enabled, or disabled

- (ad-deactivate 'previous-line)
Ways to use advice

- **before**: Add code before a command
- **after**: Add code after a command
- **around**: Make a wrapper around invocation of command
  - Useful for executing the command more than once or not at all
  - You can also modify the environment
Example: around-advice

- (defadvice previous-line
   (around my-advice)
   "Conditionally allow previous-line."
   (if condition1
      ad-do-it))
Foundations of elisp

- Data types in elisp
- Control flow
Data types

- Lisp data types
  - integer, cons, symbol, string, ...
  - Cursor position represented as integer
- Emacs-specific data types
  - buffer, marker, window, frame, overlay, ...
Control flow

- `(progn (do-this) (do-something-else))`

- All forms are evaluated, and the result of the last one is returned
  
  - Useful in e.g. `(if var (do-this) (do-that))` where a single form is required
  
  - Some control structures like `let` have an *implicit* `progn`
Control flow

- (if condition
do-this-if-true
do-this-is-false)
- (cond (condition1 result1)
 (condition2 result2)
 ... 
 (t default-result))
Control flow

• `or` returns the first non-nil argument, or nil

  - Short-circuit evaluation

    - `(defun frob-buffer (buffer)
        "Frob BUFFER (or current buffer if it's nil)"
        (let ((buf (or buffer
                     (current-buffer)))
             ...
    )

    - `(defun frob-buffer (buffer)
        "Frob BUFFER or prompt the user if it's nil"
        (let ((buf (or buffer
                     (read-buffer "Prompt: "))
             ...
    )
Control flow

- `and` returns the last argument if all arguments are non-nil

  - Short-circuit evaluation

  - `(and condition1 condition2 (do-this))`

    • equivalent to:

      `(if (and condition1 condition2) (do-this))`
Control flow

- (while condition
  (do-this)
  (do-that)
  ...)

-
Dynamic scoping

• (defun first (x)
   (second))
   (defun second ()
   (message "%d" x))

• What does (first 5) do?
  - Dynamic scoping: 5
  - Lexical scoping: a global value of x is found
Using dynamic scoping

- Setting variables can alter function behavior
  - No need to pass extra arguments through the chain of function calls

- \texttt{text search is case-sensitive; when case-fold-search is nil (let \((\text{case-fold-search nil})) \text{ (a-complex-command))}}
  - Any searches done inside \text{a-complex-command} are altered to be case sensitive
Interactive forms

- Recall: *interactive* tells elisp that your function may be invoked with \( \text{M-} \text{x} \), and specifies what arguments to provide.

- The provided arguments may be:
  - The result of prompting the user (e.g. for a buffer)
  - Something in the current state (e.g. the region)
Interactive forms

- Example: find-file (C-x C-f)

  - (find-file FILENAME) opens FILENAME in a new buffer

  - M-x find-file or C-x C-f prompts user for a filename, then calls (find-file ...) with it

- Interactive forms make functions more flexible, allowing code reuse
Interactive forms

• Place any of the following at the top of your function
• Pass no arguments
  - (interactive)
• Prompt user for a buffer to provide
  - (interactive "bSelect a buffer: ")
  - Like how kill-buffer works
Interactive forms

• Prompt user for a file to provide
  - (interactive "fFile to read: ")
  - Like how find-file works

• Provide nil
  - (interactive "i")
Interactive forms

• Provide position of point
  - (interactive "d")

• Provide positions of point and mark, first one first
  - (interactive "r")
  - Example: indent-region
Interactive forms

- Provide prefix argument
  - `(interactive "p")`
  - Example: previous-line
Example: interactive forms

- (defun count-words-region (beginning end)
  "Print number of words in the region."
  (interactive "r")
  (save-excursion
    (let ((count 0))
      (goto-char beginning)
      (while
        (and
          (< (point) end)
          (re-search-forward \w+\W* end t))
      (setq count (1+ count)))
    (message "Region contains %d word%s"
      count
      (if (= 1 count) "" "s"))))
Interactive forms

- interactive can provide multiple arguments to your function
  - Separate different specifiers with a newline "\n"
  - Example:
    
    `(interactive
      " bSelect buffer: \n fSelect file: ")`
Reading text

- char-after, char-before
- (buffer-substring start end)
- (thing-at-point 'word)
  'line, 'whitespace, etc.
Locating the cursor

• point
• point-min, point-max
• bobp, eobp, bolp, eolp
• current-column
Moving around in text

• **goto-char**
  - Example: `(goto-char (point-min))`

• All your favorite keyboard-accessible commands (`C-f`, `C-b`, etc.)

• **save-excursion**
  - Saves current buffer, point and mark and restores them after executing arbitrary code
Modifying text

- (insert "string")
- (insert-buffer buffer)
- (newline)
- (delete-region start end)
Searching text

- (search-forward "text" LIMIT NOERROR)
  - LIMIT means only search to specified position
  - When no match is found, nil is returned if NOERROR is t

- (re-search-forward "regexp"
  LIMIT
  NOERROR)
Manipulating buffers

- `get-buffer-create`
  - Retrieves a buffer by name, creating it if necessary
- `current-buffer`
- `set-buffer`
- `kill-buffer`
Manipulating buffers

• Many functions can either take a buffer object or a string with the buffer name

• For internal-use buffers, use a name which starts with a space
Getting user input

- read-buffer
- read-file
- read-string
- etc.
Finding the right functions

• Many functions are only intended to be called interactively
  - `M-<` or `beginning-of-buffer` sets the mark and prints a message
  - To move to the beginning of the buffer, use `(goto-char (point-min))` instead

• Function documentation contains warnings about lisp use
Local variables

• Variables can be either global or local to a buffer
  - Example: `fill-column`
  - `make-local-variable`

• Default values
  - Example: `default-fill-column`
Defining a new major mode

• A major mode is defined by a procedure which:
  – Sets 'major-mode
  – Sets a keymap
  – Runs associated hooks
  – Sets local variables

• Lots of code reuse between modes
  – Usually, invoke another mode command first, then tweak keybindings, etc. (e.g. C mode)
Defining a new major mode

- The define-derived-mode macro does most of these things for you
  - Inherits settings from another major mode:
    - `(define-derived-mode new-mode parent-mode name-of-mode ...)`
Example: major mode

- (define-derived-mode
   sample-mode
   python-mode
   "Sample"
   "Major mode for illustrative purposes."
   (set (make-local-variable
         'require-final-newline)
        mode-require-final-newline))

- The macro defines M-x sample-mode
  - It also registers sample-mode-map, sample-mode-syntax-table, etc.
Example: major mode

• Now we define sample-mode-map:

  - (defvar sample-mode-map
      (let ((map (make-sparse-keymap)))
        (define-key map "\C-c\C-c"
                     'some-new-command)
        (define-key map "\C-c\C-v"
                     'some-other-command)
      map)
    "Keymap for `special-mode'.")

• Keys defined here take precedence over globally defined keys
Next steps

• Making a new major mode
  - `??-mode-syntactable`
  - *font lock* and *font-lock-defaults* to control syntax highlighting
Next steps

• Many emacs applications use buffers to interact with the user
  - Use *overlays* or *text properties* to make 'clickable' regions
Learning more about elisp

• Elisp tutorial
  - M-x info, select "Emacs Lisp Intro"

• Elisp manual
  - M-x info, select "elisp"

• Emacs source code
  - C-h f or C-h k to view function documentation;
   includes link to source code