Defining Functions and Commands

What will we cover?

- Basic function definitions
- Function parameter types
- Documenting functions
- Invoking functions with `funcall` and `apply`
- Interactive functions
- Example code!

What is a function?

- A reusable piece of code
- Can accept inputs via parameters
- Usually returns a result
- Often has a name, but can be anonymous!
- Can be called by any other code or function

Defining a function

You've probably seen this before, but let's break it down:

```
(defun do-some-math (x y)
  (* (+ x 20)
     (- y 10)))

(do-some-math 100 50)
```

[Emacs Lisp Manual: Defining Functions]

Function arguments

Functions can have different types of arguments:

- "Optional" arguments: Arguments that are not required to be provided
- "Rest" arguments: One variable to contain an arbitrary amount of remaining parameters

They can both be used in the same function definition!

```
;; If Y or Z are not provided, use the value 1 in their place
(defun multiply-maybe (x &optional y z)
  (* x
     (or y 1)
     (or z 1)))

(multiply-maybe 5)              ;; 5
(multiply-maybe 5 2)            ;; 10
(multiply-maybe 5 2 10)         ;; 100
(multiply-maybe 5 nil 10)       ;; 50
(multiply-maybe 5 2 10 7)       ;; error

;; Multiply any non-nil operands
(defun multiply-many (x &rest operands)
  (dolist (operand operands)
    (when operand
      (setq x (* x operand))))
  x)

(multiply-many 5)               ;; 5
(multiply-many 5 2)             ;; 10
(multiply-many 5 2 10)          ;; 100
(multiply-many 5 nil 10)        ;; 50
(multiply-many 5 2 10 7)        ;; 700
```

```
;; Multiply any non-nil operands with an optional operand
(defun multiply-two-or-many (x &optional y &rest operands)
  (setq x (* x (or y 1)))
  (dolist (operand operands)
    (when operand
      (setq x (* x operand))))
  x)

(multiply-two-or-many 5)        ;; 5
(multiply-two-or-many 5 2)      ;; 10
(multiply-two-or-many 5 2 10)   ;; 100
(multiply-two-or-many 5 nil 10) ;; 50
(multiply-two-or-many 5 2 10 7) ;; 700
```

Documenting functions

The first form in the function body can be a string which describes the function!

```
(defun do-some-math (x y)
  "Multiplies the result of math expressions on the arguments X and Y."
  (* (+ x 20)
     (- y 10)))
```

The convention is to capitalize the names of all parameters to the function.

Use `describe-function` to look at `alist-get` as an example!

When you need to write a longer documentation string, you can use newlines inside of the string to wrap it. Don't indent the following lines, though!

Let's wrap this documentation string:

```
  (defun do-some-math (x y)
    "Multiplies the result of math expressions on the arguments X
and Y. This documentation string is long so it wraps onto the
next line. Keep in mind that you should not indent the following
lines or the documentation string will look bad!"
    (* (+ x 20)
       (- y 10)))
```

You can use `M-q` inside of the documentation to wrap multi-line documentation strings!

Functions without names

Sometimes you need to pass a function to another function (or to a hook) but you don't want to define a named function for it.

Use a lambda function!

```
(lambda (x y)
  (+ 100 x y))

;; You can call a lambda function directly
((lambda (x y)
   (+ 100 x y))
 10 20)
```

Why "lambda"? It comes from a mathematical system called lambda calculus where the Greek lambda (λ) was used to denote a function definition.

Invoking functions

You can store a lambda function or named function reference in a variable:

```
;; The usual way
(+ 2 2)

;; Calling it by symbol
```

emacs-from-scratch / show-notes / **Emacs-Lisp-03.org** ↑ Top

Preview Code Blame Raw

```
  (message "Function: %s -- Result: %d
           fun
           (funcall fun x)))

;; Store a lambda in a variable
(setq function-in-variable (lambda (arg) (+ arg 1)))

;; Define an equivalent function
(defun named-version (arg)
  (+ arg 1))

;; Invoke lambda from parameter
(gimmie-function (lambda (arg) (+ arg 1)) 5)
```

```
;; Invoke lambda stored in variable (same as above)
(gimmie-function function-in-variable 5)

;; Invoke function by passing symbol
(gimmie-function 'named-version 5)
```

Maybe you have a list of values that you want to pass to a function? Use `apply` instead!

```
(apply '+ '(2 2))
(funcall '+ 2 2)

;; Even works with &optional and &rest parameters
(apply 'multiply-many '(1 2 3 4 5))
(apply 'multiply-two-or-many '(1 2 3 4 5))
```

Defining commands

Interactive functions are meant to be used directly by a user in Emacs!

In Emacs terminology, an interactive function is considered to be a "command."

They provide a few benefits over normal functions

- They show up in `M-x` command list
- Can be used in key bindings
- Can have parameters sent via prefix arguments, `C-u`

When you write your own package, your user-facing functions should be defined as commands (unless you are writing a programming library!)

[Emacs Lisp Manual: Defining Commands]

Defining an interactive function

The form `(interactive)` needs to be the first one in the function definition!

```
(defun my-first-command ()
  (interactive)
  (message "Hey, it worked!"))
```

Invoke it using `M-x` !

The description will now be different in `describe-function` .

Interactive parameters

The `interactive` form accepts parameters that tells Emacs what to do when the command is executed interactively (either via `M-x` or when used via key binding). Some examples:

General arguments

- `N` - Prompt for numbers or use a numeric prefix argument
- `p` - Use numeric prefix without prompting (only prefix arguments)
- `M` - Prompt for a string
- `i` - Skip an "irrelevant" argument

Files, directories, and buffers

- `F` - Prompt for a file, providing completions
- `D` - Prompt for a directory, providing completions
- `b` - Prompt for a buffer, providing completions

Functions, commands, and variables

- `C` - Prompt for a command name
- `a` - Prompt for a function name
- `v` - Prompt for a custom variable name

We won't go through every possibility, check the documentation for more:

Emacs Manual: Interactive codes

Examples

Try to bind `C-c z` to `do-some-math` which we defined earlier:

```
(global-set-key (kbd "C-c z") 'do-some-math)
```

Let's run it!

It tells us that `commandp` returns false for this function, it's not a command!

```
(defun do-some-math (x y)
  "Multiplies the result of math expressions on the two arguments"
  (interactive)
  (* (+ x 20)
     (- y 10)))
```

Run it again!

Now it complains about not having arguments for `x` and `y`. Let's fix it!

```
(defun do-some-math (x y)
  "Multiplies the result of math expressions on the two arguments"
  (interactive "N")
  (* (+ x 20)
     (- y 10)))
```

It needs to prompt for both parameters!

```
(defun do-some-math (x y)
  "Multiplies the result of math expressions on the two arguments"
  (interactive "N\nN")
  (* (+ x 20)
     (- y 10)))
```

Improve the prompts by adding a descriptive string after each:

```
(defun do-some-math (x y)
  "Multiplies the result of math expressions on the two arguments"
  (interactive "NPlease enter a value for x: \nNy: ")
  (* (+ x 20)
     (- y 10)))
```

Need to write out the result!

```
(defun do-some-math (x y)
  "Multiplies the result of math expressions on the arguments X and Y."
  (interactive "Nx: \nNy: ")
  (message "The result is: %d"
           (* (+ x 20)
              (- y 10))))
```

Try it with numeric prefix argument:

Let's look at a couple other examples:

```
(defun ask-favorite-fruit (fruit-name)
  (interactive "MEnter your favorite fruit: ")
  (message "Wrong, %s is disgusting!" fruit-name))

(defun backup-directory (dir-path)
  (interactive "DSelect a path to back up: ")
  (message "Oops, I deleted %s" dir-path))

(defun run-a-command (command)
  (interactive "CPick a command: ")
  (message "Run %s yourself!" command))
```

A real example!

Let's define the project we'll be following for the rest of the series:

- A package for managing your dotfiles!
- Handles tangling org-mode files containing most of your configuration
- Can also initialize and manage your dotfiles repository

Today we'll define a command that automatically tangles the `.org` files in your dotfiles folder.

Finished code

```
  (setq dotfiles-folder "~/Projects/Code/emacs-from-scratch")
  (setq dotfiles-org-files '("Emacs.org" "Desktop.org"))

  (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 variab
```

```
        (interactive)
        (dolist (org-file dotfiles-org-files)
          (dotfiles-tangle-org-file org-file))
        (message "Dotfiles are up to date!"))
```