```
;; This gives an introduction to Emacs Lisp in 15 minutes (v0.2d)
;;
;; Author: Bastien / @bzg2 / http://bzg.fr
;;
;; First make sure you read this text by Peter Norvig:
;; http://norvig.com/21-days.html
;;
;; Then install GNU Emacs 24.3:
;;
;; Debian: apt-get install emacs (or see your distro instructions)
;; OSX: http://emacsformacosx.com/emacs-builds/Emacs-24.3-universal-10.6.8.dmg
;; Windows: http://ftp.gnu.org/gnu/windows/emacs/emacs-24.3-bin-i386.zip
;;
;; More general information can be found at:
;; http://www.gnu.org/software/emacs/#Obtaining

;; Important warning:
;;
;; Going through this tutorial won't damage your computer unless
;; you get so angry that you throw it on the floor.  In that case,
;; I hereby decline any responsibility.  Have fun!

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Fire up Emacs.
;;
;; Hit the `q' key to dismiss the welcome message.
;;
;; Now look at the gray line at the bottom of the window:
;;
;; "*scratch*" is the name of the editing space you are now in.
;; This editing space is called a "buffer".
;;
;; The scratch buffer is the default buffer when opening Emacs.
;; You are never editing files: you are editing buffers that you
;; can save to a file.
;;
;; "Lisp interaction" refers to a set of commands available here.
;;
;; Emacs has a built-in set of commands available in every buffer,
;; and several subsets of commands available when you activate a
;; specific mode.  Here we use the `lisp-interaction-mode', which
;; comes with commands to evaluate and navigate within Elisp code.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Semi-colons start comments anywhere on a line.
;;
;; Elisp programs are made of symbolic expressions ("sexps"):
(+ 2 2)

;; This symbolic expression reads as "Add 2 to 2".

;; Sexps are enclosed into parentheses, possibly nested:
```

```
(+ 2 (+ 1 1))

;; A symbolic expression contains atoms or other symbolic
;; expressions.  In the above examples, 1 and 2 are atoms,
;; (+ 2 (+ 1 1)) and (+ 1 1) are symbolic expressions.

;; From `lisp-interaction-mode' you can evaluate sexps.
;; Put the cursor right after the closing parenthesis then
;; hold down the control and hit the j keys ("C-j" for short).

(+ 3 (+ 1 2))
;;           ^ cursor here
;; `C-j' => 6

;; `C-j' inserts the result of the evaluation in the buffer.

;; `C-xC-e' displays the same result in Emacs bottom line,
;; called the "minibuffer".  We will generally use `C-xC-e',
;; as we don't want to clutter the buffer with useless text.

;; `setq' stores a value into a variable:
(setq my-name "Bastien")
;; `C-xC-e' => "Bastien" (displayed in the mini-buffer)

;; `insert' will insert "Hello!" where the cursor is:
(insert "Hello!")
;; `C-xC-e' => "Hello!"

;; We used `insert' with only one argument "Hello!", but
;; we can pass more arguments -- here we use two:

(insert "Hello" " world!")
;; `C-xC-e' => "Hello world!"

;; You can use variables instead of strings:
(insert "Hello, I am " my-name)
;; `C-xC-e' => "Hello, I am Bastien"

;; You can combine sexps into functions:
(defun hello () (insert "Hello, I am " my-name))
;; `C-xC-e' => hello

;; You can evaluate functions:
(hello)
;; `C-xC-e' => Hello, I am Bastien

;; The empty parentheses in the function's definition means that
;; it does not accept arguments.  But always using `my-name' is
;; boring, let's tell the function to accept one argument (here
;; the argument is called "name"):

(defun hello (name) (insert "Hello " name))
;; `C-xC-e' => hello
```

```
;; Now let's call the function with the string "you" as the value
;; for its unique argument:
(hello "you")
;; `C-xC-e' => "Hello you"

;; Yeah!

;; Take a breath.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Now switch to a new buffer named "*test*" in another window:

(switch-to-buffer-other-window "*test*")
;; `C-xC-e'
;; => [screen has two windows and cursor is in the *test* buffer]

;; Mouse over the top window and left-click to go back.  Or you can
;; use `C-xo' (i.e. hold down control-x and hit o) to go to the other
;; window interactively.

;; You can combine several sexps with `progn':
(progn
  (switch-to-buffer-other-window "*test*")
  (hello "you"))
;; `C-xC-e'
;; => [The screen has two windows and cursor is in the *test* buffer]

;; Now if you don't mind, I'll stop asking you to hit `C-xC-e': do it
;; for every sexp that follows.

;; Always go back to the *scratch* buffer with the mouse or `C-xo'.

;; It's often useful to erase the buffer:
(progn
  (switch-to-buffer-other-window "*test*")
  (erase-buffer)
  (hello "there"))

;; Or to go back to the other window:
(progn
  (switch-to-buffer-other-window "*test*")
  (erase-buffer)
  (hello "you")
  (other-window 1))

;; You can bind a value to a local variable with `let':
(let ((local-name "you"))
  (switch-to-buffer-other-window "*test*")
  (erase-buffer)
  (hello local-name)
  (other-window 1))

;; No need to use `progn' in that case, since `let' also combines
```

```
;; several sexps.

;; Let's format a string:
(format "Hello %s!\n" "visitor")

;; %s is a place-holder for a string, replaced by "visitor".
;; \n is the newline character.

;; Let's refine our function by using format:
(defun hello (name)
  (insert (format "Hello %s!\n" name)))

(hello "you")

;; Let's create another function which uses `let':
(defun greeting (name)
  (let ((your-name "Bastien"))
    (insert (format "Hello %s!\n\nI am %s."
                    name       ; the argument of the function
                    your-name  ; the let-bound variable "Bastien"
                    ))))

;; And evaluate it:
(greeting "you")

;; Some function are interactive:
(read-from-minibuffer "Enter your name: ")

;; Evaluating this function returns what you entered at the prompt.

;; Let's make our `greeting' function prompt for your name:
(defun greeting (from-name)
  (let ((your-name (read-from-minibuffer "Enter your name: ")))
    (insert (format "Hello!\n\nI am %s and you are %s."
                    from-name ; the argument of the function
                    your-name ; the let-bound var, entered at prompt
                    ))))

(greeting "Bastien")

;; Let's complete it by displaying the results in the other window:
(defun greeting (from-name)
  (let ((your-name (read-from-minibuffer "Enter your name: ")))
    (switch-to-buffer-other-window "*test*")
    (erase-buffer)
    (insert (format "Hello %s!\n\nI am %s." your-name from-name))
    (other-window 1)))

;; Now test it:
(greeting "Bastien")

;; Take a breath.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;
;; Let's store a list of names:
(setq list-of-names '("Sarah" "Chloe" "Mathilde"))

;; Get the first element of this list with `car':
(car list-of-names)

;; Get a list of all but the first element with `cdr':
(cdr list-of-names)

;; Add an element to the beginning of a list with `push':
(push "Stephanie" list-of-names)

;; NOTE: `car' and `cdr' don't modify the list, but `push' does.
;; This is an important difference: some functions don't have any
;; side-effects (like `car') while others have (like `push').

;; Let's call `hello' for each element in `list-of-names':
(mapcar 'hello list-of-names)

;; Refine `greeting' to say hello to everyone in `list-of-names':
(defun greeting ()
    (switch-to-buffer-other-window "*test*")
    (erase-buffer)
    (mapcar 'hello list-of-names)
    (other-window 1))

(greeting)

;; Remember the `hello' function we defined above?  It takes one
;; argument, a name.  `mapcar' calls `hello', successively using each
;; element of `list-of-names' as the argument for `hello'.

;; Now let's arrange a bit what we have in the displayed buffer:

(defun replace-hello-by-bonjour ()
    (switch-to-buffer-other-window "*test*")
    (goto-char (point-min))
    (while (search-forward "Hello")
      (replace-match "Bonjour"))
    (other-window 1))

;; (goto-char (point-min)) goes to the beginning of the buffer.
;; (search-forward "Hello") searches for the string "Hello".
;; (while x y) evaluates the y sexp(s) while x returns something.
;; If x returns `nil' (nothing), we exit the while loop.

(replace-hello-by-bonjour)

;; You should see all occurrences of "Hello" in the *test* buffer
;; replaced by "Bonjour".

;; You should also get an error: "Search failed: Hello".
;;
```

```
;; To avoid this error, you need to tell `search-forward' whether it
;; should stop searching at some point in the buffer, and whether it
;; should silently fail when nothing is found:

;; (search-forward "Hello" nil 't) does the trick:

;; The `nil' argument says: the search is not bound to a position.
;; The `'t' argument says: silently fail when nothing is found.

;; We use this sexp in the function below, which doesn't throw an error:

(defun hello-to-bonjour ()
    (switch-to-buffer-other-window "*test*")
    (erase-buffer)
    ;; Say hello to names in `list-of-names'
    (mapcar 'hello list-of-names)
    (goto-char (point-min))
    ;; Replace "Hello" by "Bonjour"
    (while (search-forward "Hello" nil 't)
      (replace-match "Bonjour"))
    (other-window 1))

(hello-to-bonjour)

;; Let's colorize the names:

(defun boldify-names ()
    (switch-to-buffer-other-window "*test*")
    (goto-char (point-min))
    (while (re-search-forward "Bonjour \\(.+\\)!" nil 't)
      (add-text-properties (match-beginning 1)
                           (match-end 1)
                           (list 'face 'bold)))
    (other-window 1))

;; This functions introduces `re-search-forward': instead of
;; searching for the string "Bonjour", you search for a pattern,
;; using a "regular expression" (abbreviated in the prefix "re-").

;; The regular expression is "Bonjour \\(.+\\)!" and it reads:
;; the string "Bonjour ", and
;; a group of            | this is the \\( ... \\) construct
;;   any character       | this is the .
;;   possibly repeated   | this is the +
;; and the "!" string.

;; Ready?  Test it!

(boldify-names)

;; `add-text-properties' adds... text properties, like a face.

;; OK, we are done.  Happy hacking!
```

```
;; If you want to know more about a variable or a function:
;;
;; C-h v a-variable RET
;; C-h f a-function RET
;;
;; To read the Emacs Lisp manual with Emacs:
;;
;; C-h i m elisp RET
;;
;; To read an online introduction to Emacs Lisp:
;; https://www.gnu.org/software/emacs/manual/html_node/eintr/index.html

;; Thanks to these people for their feedback and suggestions:
;; - Wes Hardaker
;; - notbob
;; - Kevin Montuori
;; - Arne Babenhauserheide
;; - Alan Schmitt
;; - LinXitoW
;; - Aaron Meurer
```