

Emacs, Lisp, and Python

Programming Information for UMaine COS 470

Roy M. Turner

Spring, 2021 (version 2021-01-23)

Contents

1	Emacs	3
1.1	What is Emacs, anyway?	3
1.2	Installing Emacs	4
1.3	Using Emacs	4
1.3.1	Starting and quitting Emacs	4
1.3.2	Interrupting Emacs	5
1.3.3	Learning Emacs basics	5
1.3.4	Customizing Emacs	5
1.3.5	Editing Lisp files	8
1.3.6	Finding and installing Emacs packages	10
1.3.7	Useful modes and commands	10
2	Lisp	13
2.1	What is Lisp, anyway?	13
2.2	Installing Lisp	14
2.2.1	Pre-packaged Lisp+Emacs	15
2.2.2	Installing the Lisp and Emacs separately	15
2.3	Using Lisp Within Emacs	16
2.3.1	Starting and quitting Lisp	16
2.3.2	Editing and evaluating functions	17
2.3.3	Loading functions into Lisp	17
2.3.4	I/O in Lisp	18
2.3.5	Getting help	19
2.3.6	Debugging Lisp programs	20
2.4	Lisp libraries and packages	21
2.5	Resources for Lisp	21
2.6	Expectations for Lisp programs	21
2.6.1	Indentation	22
2.6.2	Capitalization/word separation	22
2.6.3	Commenting conventions	22

3	Python and Keras	23
3.1	IDE for Python	24
3.1.1	Elpy	24
3.1.2	Jupyter	24
3.1.3	Pyenv mode	24
3.1.4	Interpreter	24
3.2	TensorFlow and Keras	25
3.3	Jupyter notebooks and JupyterLab	25
4	Turning in programs	25
4.0.1	Written portion	25
4.0.2	Sample runs	26
5	Literate programming	26

Emacs, Lisp, and Python

Programming Information for UMaine COS 470

Welcome to COS 470/570, Introduction to Artificial Intelligence! This packet describes some of what you will need to know to do your AI programming and homework assignments. First, we will talk about how to access or get the software you need: Emacs, Lisp, and Python, as well as packages you will need. Emacs works very well as an integrated development environment (IDE) for AI programming using either language. Then we will describe how to use Emacs and Lisp them. (Hopefully, you already know how to use Python!) Finally, we will discuss what is expected of your assignments for this class.

I should mention that I am *not* a Windows user, and so if you choose to use Windows, you are more or less on your own: this information packet is targeted toward Linux and macOS. Both Lisp and Emacs are available for Windows, however, and you should have little trouble finding online resources to supplement what I provide here. You should also consider installing Linux, either as a second operating system on your computer, running in a virtual machine (e.g., using VirtualBox¹), or as a Docker² container. You can also install Linux directly onto your Windows machine from the Microsoft Store (I'd suggest Ubuntu) after configuring your computer to use the Windows Subsystem for Linux (WSL 2³).

1 Emacs

1.1 What is Emacs, anyway?

Only the best thing since sliced bread.⁴ It's a family of programs that can trace their origin to a set of editor macros (thus the name) created in the 1970s. We'll be using the Free Software Foundation's GNU Emacs, which is the most widespread variant. It consists of a small core written in C, which includes a Lisp interpreter (Emacs Lisp, or elisp). The remainder of the program, as well as many, many packages, are written in elisp.⁵

Over time, Emacs evolved from an editor to a...well...the joke is, it's an operating system. Only I'm not sure it's that much of a joke. You can do almost *anything* within Emacs, from editing (duh) to project management to organizing your to-do list to reading/writing email to playing Tetris to... Well, you get the idea.

But what it really does well is function as a *programmer's* editor and IDE. This is especially true for Lisp, since Emacs and Lisp grew up together, you might say; the Lisp Machines, for example, provided a version of Emacs (Zmacs) as their IDE (arguably the *first* IDE), and consequently, Emacs is *very* Lisp-aware. It turns out it is also good for just about any other programming language, including Python. Since both Lisp and Python are interpreted languages, Emacs supports running the respective interpreters inside of Emacs, with tight communication between the IDE and the interpreter.

Emacs is a modeless editor, unlike, e.g., Vi (or Vim, or...). Whereas when using a mode-based editor, you are either communicating with the editor or writing (code, for example), in Emacs, the

¹<https://www.virtualbox.org>

²<https://www.docker.com>

³<https://devblogs.microsoft.com/commandline/announcing-wsl-2/>

⁴I may be a tad biased.

⁵You *could* do your Lisp assignments in elisp, especially since you can load a package to make it conform mostly to Common Lisp. While I don't have anything really against this, it will be slower—much slower—than a Lisp such as SBCL that can be compiled.

two are interleaved. Editor commands are *bound* to key combinations, such as **C-e** (control-e) to go to the end of the current line, **C-n** to go to the next line, etc.⁶ These can sometimes be rather baroque, since Emacs has a *lot* of possible commands—for example, **C-M-t** (transpose words), **C-x C-f** (edit file), etc. Believe it or not, it starts to make sense and become second nature after a while. And, if not, there are always menus.

Or, even better, change things to suit yourself. Emacs is completely and (mostly) easily customizable (via an extensive customization facility as well as by writing elisp code; try **M-x customize**), and this includes which keys are bound to which editor commands. If you want Windows bindings, no problem; just a few lines in your initialization file, and you're all set.

1.2 Installing Emacs

Emacs runs on all major operating systems, though being from the FSF, it's developed on GNU/Linux. You can download the source from their Emacs website.⁷ That site also has links to pre-compiled versions of Emacs. Most package managers also know about Emacs, too. For example, on a Mac, I could use Homebrew or MacPorts to install it.⁸

1.3 Using Emacs

1.3.1 Starting and quitting Emacs

You will usually start Emacs just like you would any other application. However, it is possible to run Emacs within a terminal, i.e., with it not having its own window, which may occasionally be handy if you are logged in to your computer remotely. To start it this way, from a shell (terminal), type:

```
emacs -nw
```

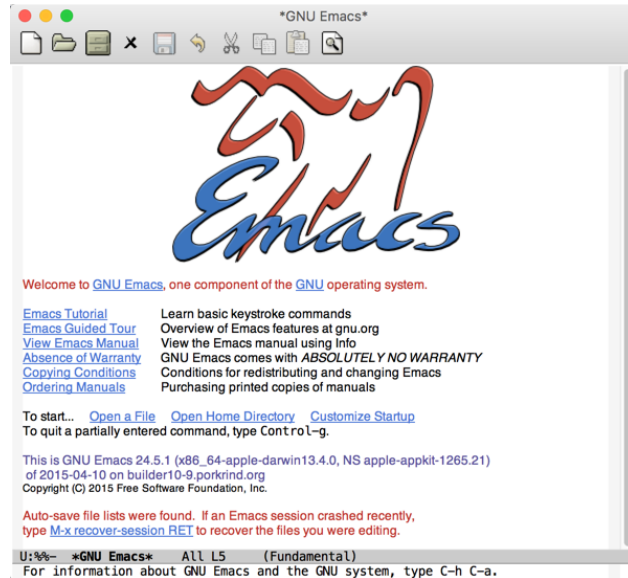
assuming, of course, that the `emacs` binary is on your load path.

When run as a regular app, Emacs should open a window that looks like:

⁶When you are cursing Emacs for its key bindings that don't match with the ones you learned from Windows or the Mac, just remember I'll be cursing the Windows/Mac bindings. Emacs came first, and I *still* haven't gotten used to the "newfangled" bindings. Seriously, though, you change bindings to suit yourself.

⁷<https://www.gnu.org/software/emacs>

⁸On the Mac, I'd suggest getting the pre-compiled binaries so that you can install it like a usual Mac application in `/Applications`. If you want to call it from the command line, then use Homebrew or MacPorts.



To quit Emacs, either use the menu (e.g., under the **File** menu item on macOS) or type the key combination `C-x` (control-x) followed by `C-c` (control-c).

1.3.2 Interrupting Emacs

If at any time you want Emacs to stop what it is doing, its own “interrupt character” is `C-g`. (The usual key combination to stop a process, `C-c C-c`, will not work, since Emacs rebinds and uses just about all the control characters.)

1.3.3 Learning Emacs basics

The best way to learn how to use Emacs is to take its built-in tutorial. Unless you have re-bound it to another key, typing the control-h (`C-h`) key will run the Emacs command `help-for-help`, which brings up Emacs’ help facility. Start Emacs and type `C-h`. You will see a line in the *minibuffer* at the bottom of the screen that looks like:

```
C-h (Type ? for further options)-
```

Type `?` now, and Emacs will show a list of different help commands. Type `t` now, and it will begin the tutorial.

You should take the tutorial now. I’ll wait right here. 😊

1.3.4 Customizing Emacs

In order to have Lisp and Emacs communicate well, there are a few things that need to be done. To get you started, I will provide a sample `.emacs` file you can copy into your home directory. I will explain it as needed in this document. There may also be other things you want to add to the file over time as you get more conversant with Emacs; you will also likely want to explore the customizations you can make using the `M-x customize` extended command.

Let’s look at the sample `.emacs` file. The first line looks like:

```
1   ;;; -*- Mode: emacs-lisp -*-
```

This just tells Emacs what kind of file it is, in this case, one containing Emacs Lisp expressions. Emacs is pretty good at determining what kinds of things files contain when they have the appropriate extensions (`.lisp` for Lisp files, `.py` for Python, etc.), but in the case of `.emacs`, since there is no extension, it's best to tell Emacs. This is not required; it just lets Emacs be smarter about syntax highlighting, etc., when the file is being edited.

```
2   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
3   ;;;
4   ;;; Expand the list of packages shown with M-x list-packages. This should,
5   ;;; for example, allow SLIME to be loaded via the package list mechanism.
6   ;;;
7   ;;; NOTE: If the "slime" package doesn't show up when you do M-x
8   ;;; list-packages, try M-x package-list-packages.
9   ;;;
10
11  (require 'package)
12
13  (add-to-list 'package-archives
14              ("melpa" . "http://melpa.org/packages/") t)
15
16  (package-initialize)
```

This just adds the MELPA elisp repository⁹ to the list of repository Emacs checks for packages to make it easier to install packages.

Next comes some customizations to allow to use Lisp easily from within Emacs. The package you'll use for this is called SLIME, the Superior Lisp Interaction Mode for Emacs (for more information, including pointers to the documentation, see the WikEmacs page¹⁰). It is a very good Emacs \leftrightarrow Lisp interface that works with virtually all modern Lisps, including ACL, CCL, and SBCL. If you are going to use CCL or SBCL, then this is the interface you want to use.

What SLIME does is control Lisp from within Emacs. It lets you interact with the read-eval-print loop (REPL, i.e., the interpreter) in a buffer, and it provides useful functionality when editing Lisp programs, such as finding symbols and definitions, displaying argument lists, completion, indentation, and syntax highlighting. It also helps debug and inspect your Lisp functions and objects. It is tightly integrated with Emacs via one or more modes, so once you install it, you just interact with Emacs in a normal way.

SLIME is available via a GIT repository¹¹; however, the easiest way to install SLIME is to just let Emacs do it itself. You can do this manually via the M-x `list-packages` command, then selecting `slime` to install, but here, we'll add something to the `.emacs` file to install it and configure it for you:

```
17  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
18  ;;;
19  ;;; SLIME and Lisp setup
```

⁹MELPA = Milkypostman's Emacs Lisp Package Archive, at melpa.org.

¹⁰<https://wikemacs.org/wiki/SLIME>

¹¹<https://github.com/slime/slime>

```

20 ;;
21
22 ;;; NOTE: Change this to point to where your Lisp is installed.
23
24 (setq inferior-lisp-program "/usr/local/bin/sbcl")
25
26 (package-install "slime")
27
28 (require 'slime-autoloads)
29 (slime-setup '(slime-fancy slime-presentations))
30
31 (global-set-key "\C-cs" 'slime-selector)
32
33 ;;; Set fill column, autofill for Lisp buffers:
34 (add-hook 'lisp-mode-hook
35   '(lambda ()
36     (slime-mode)
37     (setq fill-column 80)
38     (turn-on-auto-fill)))

```

This tells SLIME (and Emacs in general) where the Lisp is it is to use. In the case of the sample file, I just used where SBCL (Steel Bank Common Lisp; see below) is installed on my machine. The rest of the customization just sets up a few things to be nicer for you.

For example, although Emacs is happy to let you have lines as long as you want, even hiding the fact that a line has wrapped (if you do `M-x visual-line-mode`), it is good practice when writing a program to keep the lines a reasonable length. That way, when looking at code on the Web or in a non-Emacs editor, or when printing the code, the lines aren't wrapped or don't vanish off to the right. I would strongly recommend keeping the length of lines in Lisp files ≤ 80 characters. The next to the last line in the `lisp-mod-hook` definition above does this. Hooks are Lisp functions that you can set to be run by Emacs when a particular thing happens; in this case, the new anonymous function (a `lambda`; you'll learn about these if you don't already know what they are) is run whenever you edit a file in Lisp mode. It sets the line length and tells Emacs to automatically start a new line if the current one exceeds that length. (If you are happy with Emacs' default fill column of 70, you can delete that `setq` line.)

It's usually handy to highlight matching parentheses, especially in Lisp, but also in other languages, too. This customization makes Emacs parenthesis matching better:

```

39 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
40 ;;;
41 ;;; Parenthesis matching -- this can be done in customizations, too. Matching
42 ;;; is done by default, but setting show-paren-mode will cause the matching
43 ;;; parenthesis to be highlighted when the cursor is next to a parenthesis
44 ;;;
45
46 (show-paren-mode t)

```

Okay. If you've taken the tutorial, your pinky finger is likely tired of hitting the escape key ever time you wanted to type a `M-` (meta) key. There are some other customizations you can do

from within Emacs to create a real meta key on your keyboard, or rather, to make one of the keys function as a meta key. This is straightforward for macOS using `M-x customize-group <ret> ns <ret>`. This will take you to a customization screen where you can change your keyboard around like you want. For example, I change the Mac’s Command keys to be meta keys by changing the “NS Command Modifier” value to “meta”. Since I come from an era where the control key was where it’s *meant* to be, i.e., where the caps-lock key is on modern keyboards, I swap caps-lock and control. This makes it *much* easier on the fingers for Emacs commands. The Mac’s Option key is mapped to the “super” modifier.¹² Similar workarounds can be found for other keyboards, too.

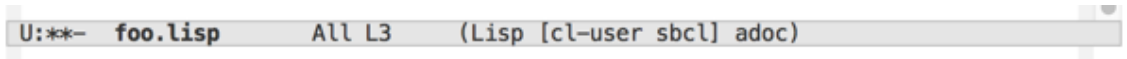
On Linux and Windows computers, and actually on Macs, too, it is easy to change the keys around globally, which I do on my Mac and Windows machines. (Yes, doing machine learning has me using Windows, alas, although with WLS 2 and Linux on top of it.) Of course, doing this will mean you have to get used to the rearranged keys for everything; and while you’re at it, you might as well change your input functionality on your computer to use Emacs key bindings for everything, right?

That’s about all you need to get started!

1.3.5 Editing Lisp files

To edit a Lisp file, just type `C-x C-f` and give Emacs the file name. You should give the file the extension `.lisp` or `.l`, unless you want to customize Emacs to have it understand that other extensions also signify Lisp files. I would suggest `.lisp`, however, as that is a standard way to identify Lisp files that most Lisp interpreters also understand.

If the file is new, an empty buffer will be displayed. Note that the mode line looks something like:



Shown is the coding style (`U` for Unix), that the file has been changed since being opened (`***`; I had already changed this file), what is being shown (all of the file), what line the cursor is on (my cursor was on line 3), and the major and any minor modes. In this case, `Lisp` is the major mode for editing lisp files. The odd-looking string:

```
[cl-user sbcl]
```

is from SLIME; it is telling you that the file’s Lisp code will be placed in the `cl-user` package and that the Lisp interpreter is “`sbcl`”.¹³

The `adoc` means that a completion/documentation package has been loaded by SLIME to help you as you program. To see that in action, type

¹²The super modifier opens up an entire new set of possible keys to bind commands to; for example, `super-n` on my machine runs `make-frame`, which creates a new frame (window) from the current buffer. If you think having control, meta, and super (and shift) as modifiers is extreme—you can have, for example, a key combination of `C-M-super-shift a`, thus playing Twister with your fingers—on the Lisp Machines, where most of this comes from, there was also a “hyper” key, and on precursors of that, there were in total *seven* modifier keys. And no, *I* don’t use anything but control, meta, and shift on a regular basis.

¹³A Lisp *package* is a namespace, or symbol table; `cl-user` is the default package for you, the Common Lisp user, that holds references to all the symbols you create. Think of it as similar to how Python splits up the namespace based on its packages. When you `import time`, for example, you can access the `sleep` function as `time.sleep(1)`, etc. If you need to access something in another package in Lisp, you can do the same sort of thing; e.g., to get the system time from the operating system, you could call `(sb-posix:time)`, i.e., use the `time` function from the `sb-posix` package.

You will likely not have to worry much about creating your own packages, but if you do, a quick trip to a good Lisp text should help. For now, don’t worry about it.


```
(defun
```

in the new buffer and press the space bar. You should see in the minibuffer:

```
(defun name args &body body)
```

which is a description of the `defun` special form: it takes a name, an argument list, and then the body of the function being defined.

Go ahead and type in a function, for example:

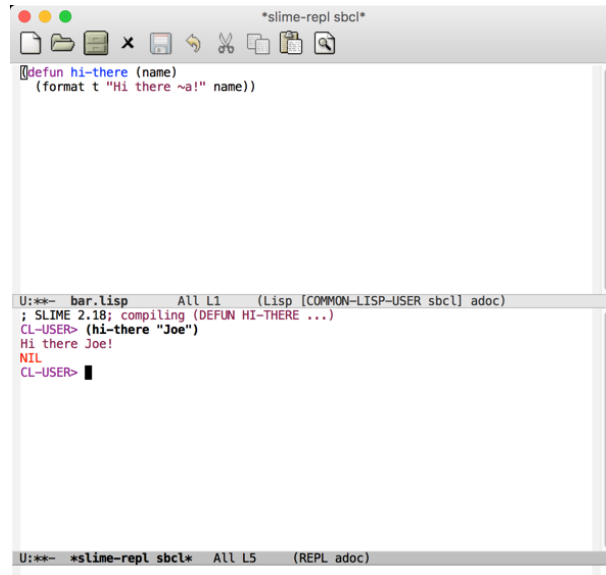
```
(defun hi-there (name)
  (format t "Hi there, ~a!" name))
```

You could save the file (`C-x C-s`), then go to the Lisp buffer and load it, then run the function. However, that is rather slow. Go ahead and start Lisp (we'll talk more about this below) by typing `M-x slime` and pressing the return key. You should see the screen split, with your file's buffer in the top and a Lisp interaction buffer in the bottom portion of the window. The cursor is in the bottom buffer.

Move the cursor to the top buffer, either by clicking in it or by typing `C-x o` ("other buffer", i.e.). Put the cursor somewhere in the function you just created and type `C-c C-c`. This runs the function `slime-compile-function`, which evaluates the function in the Lisp *image* you have running. If you put the cursor in the Lisp buffer now and type

```
(hi-there "Joe")
```

you should see something like this:



The screenshot shows an Emacs window titled `*slime-repl sbcl*`. The top buffer contains the function definition:

```
(defun hi-there (name)
  (format t "Hi there ~a!" name))
```

The bottom buffer shows the following output:

```
U:*- bar.lisp All L1 (Lisp [COMMON-LISP-USER sbcl] adoc)
; SLIME 2.18; compiling (DEFUN HI-THERE ...)
CL-USER> (hi-there "Joe")
Hi there Joe!
NIL
CL-USER> █
```

The status bar at the bottom indicates the current buffer is `*slime-repl sbcl*` and the window is titled `U:*- *slime-repl sbcl* All L5 (REPL adoc)`.

This shows that the `hi-there` function was, indeed, evaluated in Lisp.

This ability to incrementally evaluate (or compile) functions as you write a program is incredibly useful, and it also allows you to incrementally change part of the program as you debug it later.

Eventually, of course, you'll complete a file and want to save it, so that you can load it in Lisp using Lisp's own `load` function. To do that, or to save the file before you are done, type `C-x C-s`.

I should mention here that if you for some reason kill Emacs or your computer crashes, etc., before you have saved your file, all is not lost! Emacs keeps "autosave" files for every file that is

being edited after the file has been modified. While this file may not contain everything you have put into the file, it will contain all but the last few characters you have typed.

In order to restore a file that was lost or not saved, use `M-x recover-file`. This will show you a couple of lines in a temporary buffer showing the file and the autosave file so that you can decide if you really want to recover it. It asks you to answer “yes or no” about recovering it. If you answer yes, then it will open copy the autosave information into the file and open it. You should then save the file.

One other thing. SLIME and Emacs allows you to easily find the definition of functions you, or others, have written. Suppose you are working in Lisp and you realize that a function you wrote, say `foobar`, has a bug, but you can’t remember in which of the files you loaded it’s defined. If you type `M-.`—that is meta and the period—then tell it the function name, Emacs will edit the file where the function is defined.

One other last thing. SLIME comes with documentation about Lisp. From the Lisp buffer or any buffer editing a Lisp file, type `M-x slime-documentation` and tell what you want information about. For example,

```
M-x slime-documentation <ret> format <ret>
```

where `<ret>` means “press the return key”, will display quite a bit of documentation about the `format` function, which does formatted output.

1.3.6 Finding and installing Emacs packages

There are many, many Emacs packages, as you probably noted when you installed SLIME, if you did it via the Emacs package system. When you type `M-x list-packages`, you get a buffer listing all the packages Emacs can find. Typing return on any of the lines will give a bit more information about the file, as well as buttons in the buffer giving the option to install the file. There is usually also a link to the home page of the package; clicking on this will (depending on the default setup on your computer) send the URL to a browser.

Spend some time looking through the packages to see if there are any you might enjoy installing. I guarantee you will find something to pique your interest.

1.3.7 Useful modes and commands

Working with directories (folders) Emacs has a very nice way to access and edit directories (much better than the Finder or the equivalent Windows utility), called `dired`. You can access this via `C-x d` or `M-x dired`. Typing this gives a buffer that looks like this:

```

ket
/Users/ket:
total used in directory 80 available 4289791888
-rw-r--r--  1 ket  staff   40 Jan 10 21:13 #foobar.py#
drwxr-xr-x+ 25 ket  staff  850 Jan 10 21:13
lrwxr-xr-x  1 ket  staff   33 Jan 10 21:12 #foobar.py -> root@Roys-MacBook
Pro.Local.20574
drwxr-xr-x  13 root  admin  442 Sep 21 21:35 ..
-r-----  1 ket  staff   7 Nov 27 10:51 .CFUserTextEncoding
-rw-----  1 ket  staff  240 Apr 15 2014 .bash_history
drwx-----  6 ket  staff  204 Jan 10 21:12 .emacs.d
drwx-----  9 ket  staff  306 Jan 10 19:41 .gnupg
-rw-----  1 ket  staff   0 Jan 10 21:03 .python_history
-rw-r--r--  1 ket  staff  343 Mar 11 2014 .rpwd.gpg
drwxr-xr-x  3 ket  staff  102 Jan 10 20:00 .slime
-rw-r--r--  1 ket  staff  149 Jan 10 20:55 .slime-history.eld
drwx-----  3 ket  staff  102 Nov 9 2014 Applications
drwx-----  5 ket  staff  170 Apr 15 2014 Desktop
drwx-----  4 ket  staff  136 Jun 6 2014 Documents
drwx-----  4 ket  staff  136 Mar 11 2014 Downloads
drwx----- 51 ket  staff 1734 Nov 27 10:52 Library
drwx-----  3 ket  staff  102 Mar 11 2014 Movies
drwx-----  3 ket  staff  102 Mar 11 2014 Music
drwx-----  3 ket  staff  102 Mar 11 2014 Pictures
drwxr-xr-x+  5 ket  staff  170 Mar 11 2014 Public
-rw-r--r--  1 ket  staff 1225 Jan 10 20:56 foo.log
-rw-r--r--  1 ket  staff   52 Jan 10 21:11 foo.py
-rw-r--r--  1 ket  staff   20 Jan 10 21:11 foo.tex
-rw-r--r--  1 ket  staff   19 Jan 10 20:56 foo.tex~

U:%%- ket          ALL L5      (Dired by name)

```

In addition to giving a lot of information about the files and subdirectories in a directory, positioning the cursor on any of the lines lets you perform quite a few actions, such as renaming a file, copying it, changing protections, editing it, etc., and editing subdirectories or super-directories (using the “.” line, which in Unix refers to the parent directory of the current one).

Typing “?” followed by “h” in a dired buffer will give you help for the mode.

Shell modes Something that I find very useful is to have a buffer that holds a shell so that I can interact with the operating system without having to open a terminal window. There are several modes for this, including:

- M-x `shell`
- M-x `terminal`
- M-x `eshell`

The major difference between the first two is that `shell` mode is more basic, while `terminal` mode emulates an actual terminal so that you could, if you were nuts, run Emacs within a `terminal` mode buffer inside of Emacs. (I’ve done that. Don’t do that.)

The last one, `eshell` mode, is interesting. It doesn’t open a real shell, such as you would in a terminal. Instead, it is a shell *emulator* written in Emacs Lisp. It has the useful property of being able to function as a read-eval-print loop for Emacs Lisp (and Emacs) itself. Typing an elisp form in that buffer and pressing return will evaluate the form, and you can access Emacs’ variables as if they were shell variables (e.g., prefacing them with \$):

```

Welcome to the Emacs shell

~/COS470/Docs $ echo "hi there"
hi there
~/COS470/Docs $ ls
#foo.lisp#      Figs  lisp-packet.bbl  lisp-packet.org
~/COS470/Docs $ # like shell commands so far
~/COS470/Docs $ (setq foo 3)
3
~/COS470/Docs $ $foo
3: command not found
~/COS470/Docs $ echo $foo
3
~/COS470/Docs $ (+ 15 (* 3 foo))
24
~/COS470/Docs $

```

Modes for L^AT_EX You have probably been exposed, whether you liked it or not, to L^AT_EX in other classes—especially if you have had me for a class beyond COS 140! L^AT_EX is *the* typesetting program for academics, and if you haven’t used it, you should. I *strongly* suggest you use L^AT_EX for the writing components of this course.

Anyway, if you do use L^AT_EX, you will want to install the AUCTeX package (M-x `list-packages`). This will provide you with additional help for writing L^AT_EX beyond what Emacs ships with. See the home page for AUCTeX¹⁴ for more information.

Python mode(s) Part of the course will involve writing Python code, and Emacs can greatly help you there, too. When you create or edit a Python file, Emacs recognizes it (from the file extension, e.g., `.py`) and enters Python mode. You can also start Python in a buffer using (from a Python file’s buffer) C-c C-p or (from anywhere) M-x `run-python`. You can now treat Python like you do Lisp: you can send the entire buffer to Python (C-c C-c), incrementally interpret parts of the buffer (M-x `python-shell-send-region`, e.g.), etc. If you type M-x `eldoc` to run the “electric documentation” function, then Emacs will show you information about the functions being called in your Python buffer.

For even an even more powerful Python IDE in Emacs, check out the package `elpy`. You can read about it at the Read the Docs website¹⁵, including what you need to do on the Python side of things to get it working the best.

Finally, if you want to use Jupyter¹⁶ notebooks for Python development, you can do *that* from within Emacs, too, using the package `ein` from the package list buffer. (Documentation is at the EIN homepage¹⁷, though.)

Org mode Emacs comes with a mode, called Org Mode, that is extraordinarily useful for...well, for just about everything, from literate programming; to writing documents that can be exported to L^AT_EX, HTML, e-books, Word, etc.; to maintaining To Do lists; to full-fledged project management; to...well. It’s useful, let’s just say that. It’s also quite addictive, once you start using it.

¹⁴<http://www.gnu.org/software/auctex>

¹⁵<https://elpy.readthedocs.io/en/latest/index.html>

¹⁶<https://jupyter.org>

¹⁷<https://tkf.github.io/emacs-ipynotebook/>

For example, this document (and the syllabus, and the slides...) were written using Org Mode. The source for this section looks like:

```
* Python mode(s)

Part of the course will involve writing Python code, and Emacs can greatly help you
there, too. When you create or edit a Python file, Emacs recognizes (from the file
extension, e.g., .py) it and enters Python mode. You can also start Python in a
buffer using (from a Python file's buffer) C-c C-p or (from anywhere) M-x
run-python. You can now treat Python like you do Lisp: you can send the entire
buffer to Python (C-c C-c), incrementally interpret parts of the buffer (M-x
python-shell-send-region, e.g.), etc. If you type M-x eldoc to run the "electric
documentation" function, then Emacs will show you information about the functions
being called in your Python buffer.

For even an even more powerful Python IDE in Emacs, check out the package elpy. You
can read about it at elpy.readthedocs.io/en/latest/index.html, including what you
need to do on the Python side of things to get it working the best.

* Org mode

Emacs comes with a mode, called Org Mode, that is extraordinarily useful for...well,
for just about everything, from literate programming; to writing documents that can
be exported to LaTeX, HTML, e-books, Word, etc.; to maintaining To Do lists; to
full-fledged project management; to...well. It's useful, let's just say that. It's
also quite addictive, once you start using it.

For example, this document (and the syllabus, and the slides...) were written using
Org Mode. The source for this section looks like:

#+ATTR_HTML: :height 200
#+ATTR_LATEX: :height 2in
./Figs/org-mode1.png

To check it out, see the home page, orgmode.org, and to install it, use M-x
list-packages.
```

To check it out, see the home page¹⁸, and to install it, use M-x list-packages. (Depending on your installation, it may already be installed.)

We'll come back to Org Mode later when we talk about literate programming, below.

RMAIL (and VM) and mail, etc. If you want to send and read mail from within Emacs, you can do that, too (RMAIL, VM). If you want to browse the Web, yep, you can do that, too (**eww**, possibly my favorite package name). Play chess? Sure. Want to play 2048, and you left your phone at home? No problem. You get the idea.

One last thing. If you want to play with a *very* simple AI program (or Lisp has made you need some bad psychotherapy), try M-x **doctor**, which is Emacs' implementation of Weizenbaum's old ELIZA¹⁹ program.

2 Lisp

2.1 What is Lisp, anyway?

LISP, or Lisp,²⁰ is a language for handling symbols. Since AI is all about symbol manipulation, Lisp is an obvious language to use. Lisp is much more, however. It is a rapid-prototyping language. It is a meta-language, that is, a language that can define other languages. It is an object-oriented language (one of the first). It is a hacker's language. It is a thing of beauty.²¹

Lisp has a long and illustrious history. It is the second oldest computer language still in use (after FOR). It is also one of the first (if not *the* first) languages for which an integrated development

¹⁸<http://orgmode.org>

¹⁹<http://www.wikiwand.com/en/ELIZA>

²⁰You may be wondering about here, "What gives with the capitalization?" Good question. LISP is an acronym for LISt Processing language. However, many people, myself included, think that "Lisp" looks nicer, just as some people (probably the same ones) prefer "Unix" to "UNIX" (and either to Windows, of course! ☺).

²¹I may be a tad biased.

environment (IDE) was created—in fact, it’s a good bet that many of the features you like about your favorite C, C++, Java, Python, or whatever environment came from and were available for Lisp 50+ years ago. At one time, the idea of a Lisp environment went so far that special-purpose hardware, *Lisp machines*, were built. These machines’ operating systems were written in Lisp. To a large extent, even their *microcode* was Lisp. Their editor, Zwei, was a variant of Emacs and, of course, was also written in Lisp and “understood” Lisp code very well. These were machines created by and for hackers.²² Indeed, the term “hacker” was first applied to the people who built Lisp and Lisp machines (see Steven Levy’s excellent book, *Hackers: Heroes of the Computer Revolution*). Many, if not most, AI programmers are still proud to be thought of as Lisp hackers, and most of us in my generation “grew up” using Lisp machines (and still miss them).

The last of the companies that made these machines, Symbolics, succumbed several years ago to market pressures from the big software and hardware companies. However, the development environments created for them live on in several descendants for general-purpose hardware and standard operating systems.²³

Most Lisp development environments center around the idea of a partnership between an editor and a Lisp interpreter. The editor is almost always Emacs or an Emacs-like editor. The idea is that one never leaves the editor, but rather there is a seamless interaction between the two programs: edit something, tell the editor you want it evaluated, and it ships it off to Lisp; ask Lisp where something is defined, it asks the editor to edit the file containing it; and so forth. You don’t have to use one of these environments—Lisp is perfectly capable of being used by itself—but it will make your life a lot, lot easier. This, by the way, was the root of most if not all integrated development environments (IDEs).

2.2 Installing Lisp

Most of the programming done in this course *must* be done in Lisp, both because it is in the long run easier and also to meet the School’s goal of exposing you to a range of programming languages and paradigms. Although you do not have to use Emacs, you are strongly encouraged to. Emacs is “the extensible, customizable, self-documenting real-time display editor”. It is a *programmer’s* editor, as compared with other editors you may have used in the past. However, it is much more than an editor, as we will see. You will benefit from using it not only for editing your Lisp code, but also for interacting with Lisp. You may find that you like it enough that it becomes your go-to editor (and it’s good for so much more than editing files).²⁴

There are several versions of Lisp available on the web. You will need to find one that is a version of Common Lisp. There are some very good free Lisp interpreters/compiler available, fortunately, as well as some that are for sale. The two that I would recommend are: Steel Bank Common Lisp (SBCL) and Allegro Common Lisp (ACL). The former is completely free, while the latter is available as a trial version that is adequate for the needs of this course. Code written in one should run in the other, unless you are using some of the version-specific extensions provided,

²²“Hacker” did not always have the perjorative connotation it does today, of people who break into computers; those people are actually better called “crackers” (with apologies to my Southern roots). The first hackers were people who had an intimate knowledge of programs, computers, and programming, and who enjoyed playing around with them and pushing their limits. This is the sense we use the term in AI when we refer to Lisp hackers, logic hackers, etc.

²³See symbolics.com, which is now a museum site for the Internet. This was the first Internet .com domain name, by the way.

²⁴For you Vim (vi) and/or Eclipse users: sorry. Emacs and Lisp go together extremely well; Vim and Lisp, not so much. All is not lost, though: you can make Emacs *emulate* Vim using the `viper` Emacs package. As for Eclipse...well, Emacs is less resource-hungry and in my opinion is the better and more flexible editor.

which I would recommend against.

2.2.1 Pre-packaged Lisp+Emacs

The absolute easiest way to get started with Lisp is to download and use one of the Lisp+Emacs packages. The best of these *used* to be LispBox, but the original author no longer supports it, and the newest version that is available at the project that took over support was last updated in 2011. That doesn't mean it's not useful, however: at worst, you'll likely need to update Emacs once you install and run the package. It is available for Windows, Linux, and macOS.

You can find LispBox at common-lisp.net/project/lispbox²⁵. Download the version for your distribution, unpack it, and you should be ready to run—though you may have to poke around a little bit to find out *how* to run it.

For example, for macOS, they hid the shell script that starts the program *inside* the Emacs app. If you unpack the files in the directory `/opt/local/LispBox`, then to run it, you would, in a terminal window, need to:

```
cd /opt/local/LispBox
cd Emacs.app/Contents/MacOS
./lispbox.sh
```

at this point, Emacs would start, so would Lisp, and you'd be all set.

The Emacs that ships with LispBox is old, however: Emacs 23 (current version is 25). This should cause no problem for the purposes of this class.

The Lisp that ships with LispBox is Clozure²⁶, which is as far as I know a perfectly fine Lisp—but I have not done more than start it.

2.2.2 Installing the Lisp and Emacs separately

If you don't want to go the Lisp+Emacs pre-packaged route, or if you already have Emacs installed, then you will need to install Lisp, Emacs, and SLIME (the interface between the two) separately.

Lisp There are three versions of Lisp that I can recommend: Allegro Common Lisp (ACL), Steel Bank Common Lisp (SBCL), and Clozure Common Lisp (CCL, which I have not used but seems quite popular).

ACL. Allegro Common Lisp is a product of a major Lisp and semantic web company, Franz Inc.^{27,28} The free version of their product, called Allegro CL Free Express Edition, can be downloaded from franz.com/products/allegro-common-lisp²⁹ by clicking on the “Free Download” link. Installation should be relatively straightforward. For example, on macOS, the download is a .dmg file that installs just fine.

On macOS and Linux, ACL comes with an IDE that may be sufficient for your purposes—however, I would still advise that you install Emacs and use it that way, since Emacs is very useful for a multitude of programming and other tasks.

²⁵<https://common-lisp.net/project/lispbox/>

²⁶<http://clozure.com/clozure-cl.html>

²⁷<http://franz.com/>

²⁸They are stuck with a non-punny name now, but a long time ago, the Lisp they produced was called Franz Lisp.

²⁹<http://franz.com/products/allegro-common-lisp/>

ACL can interact with Emacs via SLIME or using Franz’s own interface, ELI, which I *highly* recommend—it is much friendlier than SLIME in many ways. You can find information about this at franz.com/emacs³⁰.

ACL is an industrial-strength Lisp, though with limited tech support for the free version, and I would be using it myself if it weren’t for the licensing restrictions (the free version can’t be used for “university-sanctioned research”); in fact, in better (financial) days, we used to use the commercial version for research (and still do, to a limited extent).

One thing that should be mentioned about ACL is that it comes with some very nice add-ons, including recorded Lisp training sessions, and ELI is *nice*.

SBCL. Steel Bank Common Lisp is the descendant of CMU Lisp (thus the name: “steel” for Carnegie, “bank” for Mellon). It is a high-performance, open source/free Lisp. Unlike most other Lisps, this one compiles *everything*, including whatever you type into the Lisp listener (also called the “read–eval–print loop”) interactively. This means that it behaves like a traditional Lisp interpreter, but there is no separate compilation phase needed.

SBL runs well on Linux and macOS (it’s the one I use on my Mac, for example), and it runs “experimentally” on Windows—whatever *that* means.

You can download SBCL from www.sbcl.org³¹ and install it from there. You should also be able to install SBCL using your favorite package manager on Linux and macOS. For example, on macOS, you can use Homebrew³²:

```
brew install sbcl
```

CCL. Clozure Common Lisp is a project of Clozure Associates³³ and is the descendant of the original Macintosh Common Lisp (MCL). It is available for macOS (of course), Linux, and Windows. It is available at ccl.clozure.com³⁴

Although Clozure ships with a version of Emacs called Hemlock as its IDE, it is seriously out of date compared with GNU Emacs, and I would recommend installing GNU Emacs and using SLIME to communicate between the two. However, Clozure’s built-in IDE may be sufficient for your purposes, and it may provide some advantages that I’m not aware of. Since it builds on the Emacs used by the Lisp Machines³⁵, it may be of some historical interest (and nostalgic for us old-timers who used them).

2.3 Using Lisp Within Emacs

We have talked a tiny bit above about how to set up SLIME and use Lisp. In this section, we talk about it a bit more. (If you are using Franz’s ELI, you will need to look at its documentation on Franz’s web site.)

2.3.1 Starting and quitting Lisp

To start Lisp, type: `M-x slime`.

³⁰<http://franz.com/emacs/>

³¹<http://www.sbcl.org/>

³²<http://brew.sh/>

³³<http://www.clozure.com>

³⁴<http://ccl.clozure.com>

³⁵http://www.wikiwand.com/en/Lisp_machine

After a little bit of stuff printed on the screen and some buffers coming and going, you should have a new buffer called `*slime-repl-lisp*` that contains a Lisp prompt, e.g., `CL-USER>`. You can type forms there to interact with Lisp.

You can repeat Lisp forms using SLIME's history by typing `M-p`; repeating it will go through the history. If you pass the Lisp form you want, use `M-n` to go forward. To search for a prior form, use `M-r`.

Entering Lisp is easy. To exit Lisp, however, you first have to get SLIME's attention, so to speak. By default, typing a comma at the Lisp prompt in the SLIME buffer will cause SLIME to accept its own commands. If you then press the tab key, you can see a list of the commands SLIME will accept. The ones you may find most useful are `pwd` (print the current working directory), `cd` (change the working directory), and `quit`, to quit Lisp.

2.3.2 Editing and evaluating functions

See "Editing lisp files", above.

2.3.3 Loading functions into Lisp

You can, as discussed previously, evaluate parts of a Lisp file buffer as you write it. However, the usual way to load a program into Lisp is to use the `load` Lisp function, e.g.:

```
(load "foo")
```

In most Lisp implementations, this will first try to find a current compiled version of the file, often named something like "foo.fasl"; if one is not found (perhaps because you haven't compiled the file yet), then Lisp will load the source file (e.g., "foo.lisp") as an interpreted file.

To compile a file, use the function `compile-file`

```
(compile-file "foo")
```

To compile a function that you have already evaluated, use

```
(compile 'fn)
```

where `fn` is the function name, and the single quote tells Lisp to use the symbol (the functions name) itself rather than the value of the symbol.

Some Lisps will compile, like Java, to a bytecode version of the function to speed up future processing. Most, however, compile to machine language. The result can't be loaded directly like a regular executable. However, it can be loaded by the Lisp interpreter and then it runs at machine-code speed (apart from any garbage collection and Lisp system overhead). Just for your entertainment, here's what the SBCL version of the function

```
(defun foo ()  
  (format t "This is a test"))
```

looks like when disassembled:

```

; disassembly for F00
; Size: 94 bytes. Origin: #x100595D144
; 44:      498B4C2460      MOV RCX, [R12+96]          ; thread.binding-stack-pointer
;                               ; no-arg-parsing entry point
; 49:      48894DF8      MOV [RBP-8], RCX
; 4D:      488B058CFFFFFF    MOV RAX, [RIP-116]        ; '*STANDARD-OUTPUT*'
; 54:      498BBC24C0220000  MOV RDI, [R12+8896]      ; tls: *STANDARD-OUTPUT*
; 5C:      83FF61      CMP EDI, 97
; 5F:      480F4478F9    CMOVEQ RDI, [RAX-7]
; 64:      83FF51      CMP EDI, 81
; 67:      7433      JEQ LO
; 69:      488D5C24F0    LEA RBX, [RSP-16]
; 6E:      4883EC18      SUB RSP, 24
; 72:      488B156FFFFFF    MOV RDX, [RIP-145]        ; "This is a test"
; 79:      488B0570FFFFFF    MOV RAX, [RIP-144]        ; #<FDEFINITION for WRITE-STRING>
; 80:      B904000000      MOV ECX, 4
; 85:      48892B      MOV [RBX], RBP
; 88:      488BEB      MOV RBP, RBX
; 8B:      FF5009      CALL QWORD PTR [RAX+9]
; 8E:      BA17001020      MOV EDX, 537919511
; 93:      488BE5      MOV RSP, RBP
; 96:      F8      CLC
; 97:      5D      POP RBP
; 98:      C3      RET
; 99:      0F0B10      BREAK 16                  ; Invalid argument count trap
; 9C: LO:  0F0B0A      BREAK 10                  ; error trap
; 9F:      02      BYTE #X02
; A0:      1B      BYTE #X1B                  ; UNBOUND-SYMBOL-ERROR
; A1:      1B      BYTE #X1B                  ; RAX

```

2.3.4 I/O in Lisp

There are many functions in Lisp to do I/O. Some of the more common ones are:

- `print`, `write`: Print a Lisp expression.
- `princ`: Print an expression, but omit double quotes if it is a string.
- `write-line`: Print an expression, then output a new line.
- `fresh-line`, `terpri`: Print a carriage return.
- `read`: Read a Lisp expression.
- `read-line`: Read a line as a string.
- `format`: Formatted I/O like C's `printf`. See a Lisp book or M-x `slime-documentation` `<ret> format <ret>` for information about this functions *many* features.

Lisp uses the abstraction of *streams* for input and output. The variables `*standard-output*` and `*standard-input*` point to the operating system-defined standard output and standard input, respectively. (Global “special” variables are often named like this, with asterisks.) The functions above default to work with streams contained in these two variables, but they will work equally well with streams that are associated with files, sockets, or what have you.

File I/O can be done with explicit calls to `open` and `close`, or more easily with the Common Lisp special form `with-open-file`, which is used like this (example is in ACL):

```
USER(36): (format t "This is a ~s.~%" 'test) ; ; t means *standard-output*
```

```

This is a TEST.
NIL ;; <- return value from format
USER(37): (with-open-file (out "/tmp/file.out" :direction :output)
           (format out "This is a ~s.~%" 'test))
NIL
USER(38): (shell "cat /tmp/file.out") ;; ask Unix to do a command
This is a TEST. ;; <- printed by Unix itself
0 ;; <- return code from the Unix command
USER(39):

```

For more information about I/O, consult a Lisp book or the Web.

2.3.5 Getting help

We have already seen one way to get information about Lisp functions, the `slime-documentation` Emacs command. Lisp has its own functions to get information, however; different Lisps may offer different ones, though.

One that is always available is `apropos`, which will look for all Lisp symbols containing a substring in their name. For example:

```

CL-USER> (apropos "print")
ASDF/FIND-SYSTEM::PRINT-PPRINT-DISPATCH
:ADDR-PRINT-LEN (bound)
:DEFAULT-PRINTER (bound)
;; many results omitted...
SB-PRETTY::PPRINT-ARRAY (fbound)
SB-PRETTY::PPRINT-BLOCK (fbound)
;; ...
*PRINT-LENGTH* (bound)
*PRINT-LEVEL* (bound)
*PRINT-LINES* (bound)
*PRINT-MISER-WIDTH* (bound)
*PRINT-PPRINT-DISPATCH* (bound)
*PRINT-PRETTY* (bound)
*PRINT-RADIX* (bound)
*PRINT-READABLY* (bound)
*PRINT-RIGHT-MARGIN* (bound)
;; ...
COPY-PPRINT-DISPATCH (fbound)
PPRINT (fbound)
...
PRINT (fbound)
;; ...
; No value
CL-USER>

```

Note that `apropos` will, by default, look in all packages known to Lisp. This is what a line like

```
SB-PRETTY::PPRINT-ARRAY (fbound)
```

is telling us: that there is a `PPRINT-ARRAY` function (that's what `fbound` means) in the `SB-PRETTY` package, and that the symbol is not exported from the package (thus the `::` rather than `:` had it been exported). Variables affecting the way `print` works are also listed, such as `*PRINT-PRETTY*`, which tells `print` to “pretty print”, or format nicely, the output (`bound` means that the symbol is a *bound* variable, i.e., one with a value). Finally, our old friend `print` is itself shown.

Another useful function is `describe`, which can be called on any Lisp object to give information about that object. For example:

```
CL-USER> (describe 'foo)
COMMON-LISP-USER::FOO
  [symbol]
```

`FOO` names a compiled function:

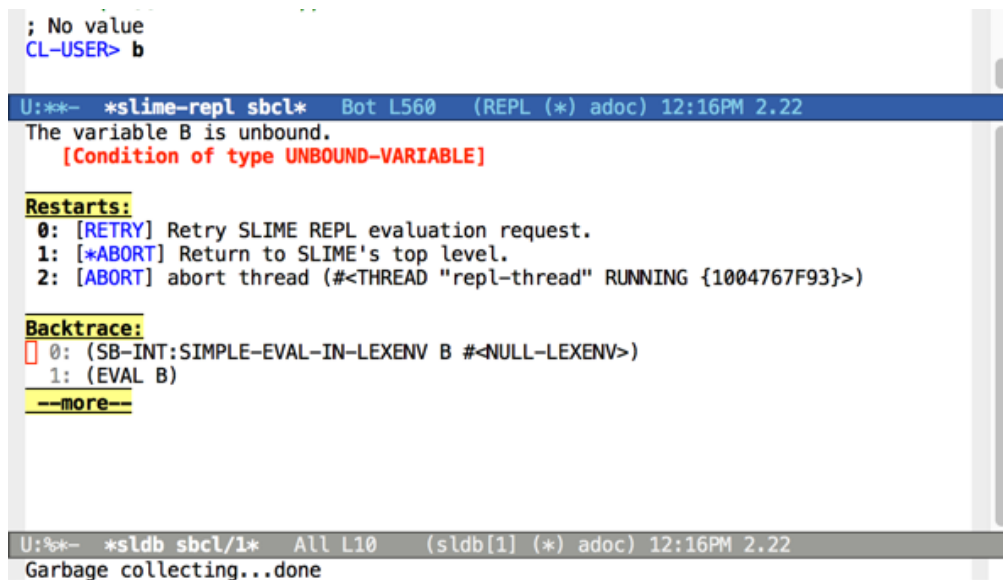
```
  Lambda-list: ()
  Derived type: (FUNCTION NIL (VALUES NULL &OPTIONAL))
  Source file: /Users/rmt/Classes/COS470/2017-Spring/foo.lisp
; No value
CL-USER>
```

SLIME itself provides a great deal of help for programming. For example, you can look up online documentation on any symbol using the “SLIME” menu item that is available whenever you are in a Lisp buffer or a Lisp file buffer.

2.3.6 Debugging Lisp programs

SLIME, in conjunction with the Lisp you are using, provides you with excellent debugging facilities. When you are in a Lisp or Lisp interpreter buffer, the “SLIME” menu provides you with debugging options, including cross-reference support (who calls this function, who does this function call, etc.), etc.

When Lisp encounters an error, SLIME brings up a debugging buffer. For example, this is what the buffer looks like when I try to evaluate an unbound symbol (I'm using SBCL):



```
; No value
CL-USER> b

U:*~ *slime-repl sbcl* Bot L560 (REPL (*) adoc) 12:16PM 2.22
The variable B is unbound.
  [Condition of type UNBOUND-VARIABLE]

Restarts:
  0: [RETRY] Retry SLIME REPL evaluation request.
  1: [*ABORT] Return to SLIME's top level.
  2: [ABORT] abort thread (#<THREAD "repl-thread" RUNNING {1004767F93}>)

Backtrace:
  0: (SB-INT:SIMPLE-EVAL-IN-LEXENV B #<NULL-LEXENV>)
  1: (EVAL B)
  --more--

U:%~ *sldb sbcl/1* All L10 (sldb[1] (*) adoc) 12:16PM 2.22
Garbage collecting...done
```

Note that it describe the problem (as well as telling you what condition was “raised”, so that you can find additional information or know what condition to catch, should you want to do that, using Lisp’s exception-handling facilities), then offers several things you can to do “restart” the misbehaving function. Typing 0, for example, will retry the evaluation; this might be useful if you change the offending function, then evaluate it in Lisp, then want to try the problem-causing function again from where it generated an error previously. If you just want to abort back to the REPL (read-eval-print loop, or *listener*), then just type “q”.

Note that the debugger also gives you a backtrace that shows the current stack frames. You can examine any of these by placing the cursor on the line and pressing return. You can move back through the stack and retry previous frames, as well.

2.4 Lisp libraries and packages

There are quite a few Lisp packages available (well, thousands, actually). One of the most useful packages is a package manager, Quicklisp³⁶; it’s written and maintained by a Mainer! This is available from www.quicklisp.org³⁷. Installing it is easy, and once installed, you can have it easily download and install any of about 1400 Lisp libraries. (There are other libraries, too, but these are the ones Quicklisp currently supports.)

Another very useful package is ASDF. ASDF (Another System Definition Facility) is a widely-used way to define *systems*, which are roughly equivalent to “projects” in other language IDE terms. Using it, groups of files or other systems that should be loaded together can be defined, as can their order, etc. It can keep track of dependencies, determine when a source file has been changed and needs to be compiled, and so forth.

See common-lisp.net/project/asdf³⁸ for how to install and use ASDF. It can also be installed via Quicklisp.

The resources listed in the next section are good starting points to find packages to do what you want to do in Lisp.

2.5 Resources for Lisp

The Common Lisp wiki, CLiki www.cliki.net³⁹ is an excellent resource for tutorials books, libraries, and even humor about Lisp. Unlike Common-Lisp.net (below), the libraries are organized by function and are described nicely. Of especial interest to some in the class may be the pointers to Lisp-based GUI and graphics libraries.

Common-Lisp.net⁴⁰ is another great resource for Lisp programmers, since it contains pointers to many libraries, tutorials, etc.

2.6 Expectations for Lisp programs

Your programs need to be both well-formatted and well-documented. There are conventions that have arisen over the years for Lisp programs that I expect you to follow, and that Emacs/SLIME will help you with.

³⁶<http://www.quicklisp.org>

³⁷<https://www.quicklisp.org/beta/>

³⁸<https://common-lisp.net/project/asdf/>

³⁹<http://www.cliki.net>

⁴⁰<http://common-lisp.net>

2.6.1 Indentation

Unlike Python, white space, including indentation, does not matter to Lisp apart from delimiting symbols, etc. However, your code should be indented for readability. If you are using Lisp and SLIME modes when editing your Lisp files—which is the default—then Emacs will automatically indent correctly when you press return or when you press tab on a line. In addition, you can select an entire region and use the command `M-x indent-region`.

Resist the urge to put too much on a line; use indentation and white space to make your code more readable.

2.6.2 Capitalization/word separation

In most modern languages, names that are really multiple words are formatted for readability by either “camel casing” them (e.g., `eventCounter`) or by using underscores to imitate spaces (e.g., `event_counter`). The convention in Lisp is to instead use dashes (e.g., `event-counter`). While it doesn’t matter much which you use, I prefer dashes, since that is more traditional and easier to type, and since some Lisps are case-insensitive.

2.6.3 Commenting conventions

Comments are very important to any program. Production-quality code spends most of its life being maintained, and comments are crucial to this process.

Comments are important in your programming assignments as well, not just so that I can figure out what you are trying to do in your code (and so give you a reasonable grade), but also so that *you* can figure out what you are trying to do during debugging.

Comments in Lisp begin with a semicolon. Everything after a semicolon to the end of the line is ignored by Lisp.

The number of semicolons starting a comment are somewhat standardized by tradition based on what the function of the comment are. A comment block that refers to the entire file, or to a complete section of the file, usually starts with `;;;;;`. Comment blocks preceding a function (or method, variable definition, etc.) usually start with `;;;`. Both of these are usually flush with the left margin. Comments taking up an entire line within a function usually start with `;;` and are indented along with the code. And comments appearing at the end of a line of code usually start with just `;` and begin in a standard “comment column”, if possible.

Emacs knows about some of these conventions, and typing `M-;` will insert semicolons in the right place and with the correct number (except for larger blocks; you may have to do those yourself).

I would suggest the same kind of file headers and function headers in all your code. For the file header, you might consider something like:

```
(in-package cl-user)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;;
;;;;; This file contains the code for a neural network to predict the stock
;;;;; market. If you are reading this file, and I am not around, then it
;;;;; works, and I'm in Hawaii.
;;;;
;;;;; Created: 1/2/2017
;;;;; Author: Roy M. Turner <rturner@maine.edu>
;;;;; Modifications:
```

```

;;;;; - 1/4/2017 - added deep learning for better predictions
;;;;;   (rtturner@maine.edu)
;;;;; - 1/7/2017 - changed it all into a random number generator; seems to
;;;;;   work better (rtturner@maine.edu)
;;;;;
;;;;;

```

For functions (or methods, or variables), you might consider headers that record the salient information you would like to have if you were encountering someone else's function—or if it has been a long time since you last looked at your own function. For example:

```

;;;
;;; Function: omit
;;; Arguments:
;;;   - list: a list
;;; Returns: nil
;;; Description:
;;;   This function takes a list and for each element outputs
;;;   the element if it is an atom, else it outputs "...".
;;; Author: Roy M. Turner <rtturner@maine.edu>
;;; Created: 1/7/2017
;;; Modifications: none
;;;

(defun omit (list)
  (let ((chars "..."))
    ; chars is set to ellipses
    (cond
      ; if at end of the list, return nil:
      ((null list) nil)
      ; if first thing on list is an atom, print it:
      ((atom (car list))
       (princ (car list))
       (princ " ")
       ; followed by a space
       (omit (cdr list)))
       ; ...and recursively call fcn
      ; otherwise, print the replacement characters
      (t (princ chars)
         (princ " ")
         (omit (cdr list)))))))

```

3 Python and Keras

You all should be familiar with Python at this point in your studies, so I won't spend much time on it here. However, if you haven't already installed Python on your computer, or if you want a very good distribution that includes a visual package manager, a Jupyter notebook system, and a bunch of data science/machine learning, then I would highly recommend installing the individual edition of Anaconda⁴¹. It should also provide the ability install to PyCharm, a very good IDE for Python.

⁴¹<http://anaconda.com>

3.1 IDE for Python

Chances are, you already have a favorite IDE for Python from your previous courses. If not, I would suggest using either Emacs, PyCharm, or Jupyter (see below).

If you decide to use Emacs, it includes a Python mode for editing your source files as well as a command, `run-python`, that will start a Python interpreter in a buffer. However, there are some customizations you will probably want to include to make it a better IDE.

Here is what I use, although there are probably better setups than what I use. (I'm a Lisp programmer, after all...)

3.1.1 Elpy

A better Python mode:

```
;; Use use-package to make loading packages easier:
(when (not (package-installed-p 'use-package))
  (package-install 'use-package))

(require 'use-package)

;; Now load elpy
(message "Loading elpy")
(use-package elpy)

;; Change this to point to where your Python is installed:
(customize-set-variable 'elpy-rpc-python-command
  "/home/rmt/anaconda/bin/python")
```

3.1.2 Jupyter

Emacs allows interaction with a Jupyter notebook server; since I'm most comfortable in Emacs, I use the notebooks from within Emacs itself:

```
;; Change to point to your own Jupyter executable:
(customize-set-variable 'ein:jupyter-default-server-command
  "/home/rmt/anaconda/envs/PyTorch/bin/jupyter")
```

3.1.3 Pyenv mode

Pyenv allows you to use multiple environments, for example, if you want a Python 2 environment for some things and a Python 3 environment with Tensorflow installed for this course:

```
(customize-set-variable 'pyenv-mode t)
```

3.1.4 Interpreter

This just points the normal Python interpreter to the right place:

```
(customize-set-variable 'python-shell-interpreter "/home/rmt/anaconda/bin/python")
```


3.2 TensorFlow and Keras

TensorFlow and PyTorch are two of the most commonly-used deep learning toolkits in use; in this course, we'll use TensorFlow. You can find information about it, including tutorials and other documentation, at tensorflow.org⁴².

Mostly, we won't be using "bare" TensorFlow in this class to build neural networks, but rather will be using the Keras front-end. Although Keras can be used with TensorFlow or Theano, TensorFlow itself comes with Keras built-in, so there isn't much you need to do to prepare it.

To install TensorFlow, you can either use Anaconda's package system or install it via pip, Python's built-in manager. For pip, do:

```
pip install tensorflow
```

Note that if you have a high-performance GPU in your computer, you may want to use it to speed up the machine learning assignment. For that, you will need to install CUDA⁴³ (for NVIDIA GPUs) and install the GPU variant of TensorFlow (`tensorflow-gpu`).

3.3 Jupyter notebooks and JupyterLab

Below, I discuss *literate programming* in the context of Lisp (and Python, too, for that matter). Another way of approaching this is to use Jupyter notebooks⁴⁴ or their next iteration, JupyterLab⁴⁵. Both of these allow you to intersperse formatted text (using Markdown⁴⁶), plots, and other things with your Python code. I'd encourage you to check it out; for one thing, I will accept notebooks instead of Python + runs and write-ups for the Python assignments.

4 Turning in programs

Unless told otherwise, all programs will be turned in electronically, usually via Blackboard. A programming assignment should include a written document describing the program (or as otherwise specified in the programming assignment), the program source code, and usually some sample runs of the program. Generally, this means that you will have to bundle these pieces together to turn them in as an archive using either `tar` and `gzip` (or `bzip2`) or `zip`.⁴⁷

4.0.1 Written portion

Usually the programming assignments handed out in class will give instructions about what is expected in the write-up for the program. In general, however, at least the following should be included:

- A description of the program's purpose and what it does.
- Any information about special features of your program.
- Information about how to run the program, sufficiently detailed that I can run it based on that information.
- An explanation of the results/sample runs.

⁴²<http://tensorflow.org>

⁴³<https://developer.nvidia.com/CUDA-zone>

⁴⁴<http://jupyter.org>

⁴⁵<http://jupyter.org>

⁴⁶<https://daringfireball.net/projects/markdown/>

⁴⁷But please, not `.rar`. Please. Really. No.

- An explanation of anything unusual you encountered while writing the program.
- An explanation of anything your program does *not* do that it should do.
- Answers to any questions asked in the assignment.

The written portion *must* be a PDF file; I will not accept anything else. I **strongly** suggest that you write and format the document using L^AT_EX (and you might experiment with Org Mode), since that is ultimately a skill you should acquire, and it produces *much* better output.

You will be graded on the quality of your writing. The written document must be well-formatted and free from grammatical and spelling errors. If you have trouble with writing, you should seek help from the Writing Center in Neville Hall.

4.0.2 Sample runs

The programming assignments (and probably your project) will require sample runs of your code to be turned in. The runs should show off your program the best you can. If there's a neat feature that your program has, make sure that it's shown in the run, or else I might miss it. If your program is best demonstrated via a video, upload that video to YouTube or elsewhere and include the URL in your write-up.

There are at least two ways of obtaining sample runs from Lisp. If you are running Lisp from within Emacs, which is *strongly* encouraged, then you can just cut and paste from Emacs' Lisp buffer.

An easier way, however, is to use Lisp's built-in `dribble` function. This will cause all output that usually goes to the standard output to also go to a file you specify, starting when `dribble` is called and continuing until `dribble` is called again without an argument.

You may want to comment on the sample runs; if you do, make sure that your comments are easily seen by me.

5 Literate programming

Literate programming refers to creating a program and its documentation as a seamless whole. It views a program as a piece of work with two audiences, humans and computers. There are many advantages of this, not least of which is keeping the documentation and code in sync with each other.

The best-known literate programming system is `cweb`, a descendant of Donald Knuth's WEB literate programming system. A `cweb` document is written with special markup to allow code to be written and discussed out of order, if desired, then the document is *tangled* to produce a source file and *woven* to produce a written document (well, a document in a formatting language such as L^AT_EX or `nroff`, anyway).

While this works well for compiled languages that have a write-compile cycle anyway, it is not ideal for interpreted languages. Consequently, some time back, I wrote a Lisp program, LP/Lisp, that instead uses the comments of a Lisp file as the source for the documentation; in some ways, it is similar to, though a bit more capable than, some of the documentation-producing programs used with Java or Python. (For example, fragments of code can be presented in the written version out of order from what is in the Lisp version.) The Lisp file *is* the LP file, in other words, and only the *weave* is necessary. This allows debugging directly from the LP file (of course).

If you are using Allegro Common Lisp (I have not ported it to SBCL or CCL) and are interested in using this system, I have made it available on the course website and at my lab's website, Maine-

SAIL.umcs.maine.edu⁴⁸ (under the “Software” menu). Be warned, though: LP/Lisp is tailored to L^AT_EX users.

I have also created a simpler LP system that should work in almost any Lisp. It is a simple converter between a marked-up Lisp file and an Org Mode file. The idea is that you could write your program in either, then convert as needed. It is not as fancy as LP/Lisp, however; for example, it can't split up Lisp functions for discussion. Unlike LP/Lisp, it is tailored for Org Mode rather than L^AT_EX. (But then, Org Mode can produce L^AT_EX.) This is available on the course web site.⁴⁹

Finally, Org Mode itself has facilities for literate programming, including evaluating and debugging directly from an Org Mode (rather than a Lisp) file. For example, the .emacs file we discussed above was created from this file using markup that looks like:

```
It's usually handy to highlight matching parentheses, especially in Lisp,
but also in other languages, too. This customization makes Emacs
parenthesis matching better:
#+BEGIN_SRC emacs-lisp +n -i :tangle yes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Parenthesis matching -- this can be done in customizations, too. Matching
;;; is done by default, but setting show-paren-mode will cause the matching
;;; parenthesis to be highlighted when the cursor is next to a parenthesis
;;;

(show-paren-mode t)
#+END_SRC
```

See the Org Mode documentation for more information about this.

⁴⁸<http://MaineSAIL.umcs.maine.edu>

⁴⁹<http://bit.ly/umaineCOS470>