

Emacs – The Best Python Editor?

This is a PDF version of The Real Python Emacs Setup Guide.

Thus far, in our series of posts on Python development environments we've looked at Sublime Text and VIM:

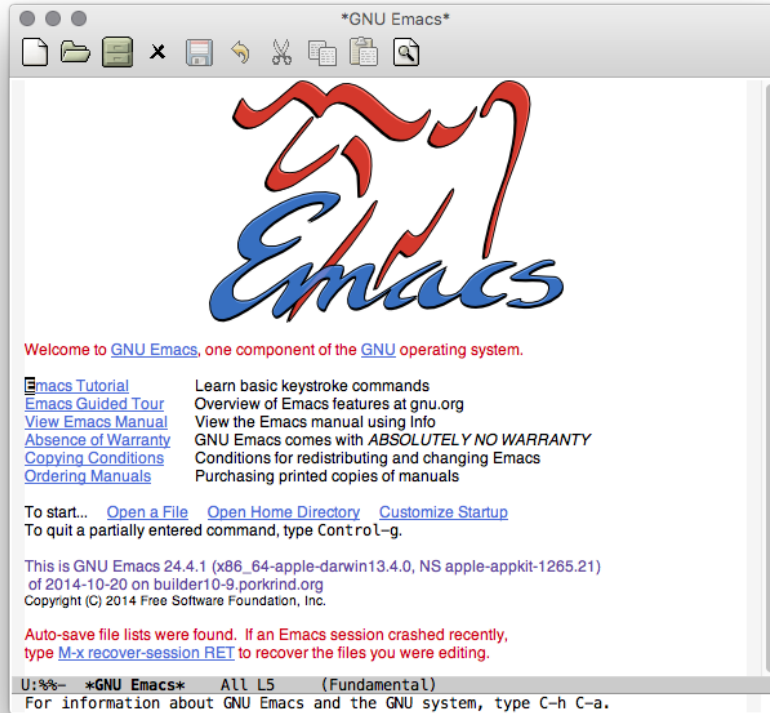
- Setting Up Sublime Text 3 for Full Stack Python Development
- VIM and Python - A Match Made in Heaven

In this post we'll present another powerful editor for Python development - Emacs. **While it's an indisputable fact that Emacs is the best editor, we'll (try to) keep an open mind and present Emacs objectively, from a fresh installation to a complete Python IDE so that you can make an informed decision when choosing your go-to Python IDE.**

Installation and Basics

Installation

Installation of Emacs is a topic that does not need to be covered in yet another blog post. This Guide, provided by ErgoEmacs, will get you up and running with a basic Emacs installation on Linux, Mac, or Windows. Once installed, start the application and you will be greeted with the default configured Emacs:



Basic Emacs

Another topic that does not need to be covered again is the basics of using Emacs. The easiest way to learn Emacs is to follow the built-in tutorial. The topics covered in this post do not require that you know how to use Emacs yet; instead, each topic highlights what you can do after learning the basics.

To enter the tutorial, use your arrow keys to position the cursor over the words “Emacs Tutorial” and press Enter. You will then be greeted with the following passage:

Emacs commands generally involve the CONTROL key (sometimes labeled CTRL or CTL) or the META key (sometimes labeled EDIT or ALT). Rather than write that in full each time, we'll use the following abbreviations:

C-<chr> means hold the CONTROL key while typing the character <chr>

Thus, C-f would be: hold the CONTROL key and type f.

M-<chr> means hold the META or EDIT or ALT key down while typing <chr>.

If there is no META, EDIT or ALT key, instead press and release the ESC key and then type <chr>. We write <ESC> for the ESC key.

So, key entries/commands like C-x C-s (which is used to save) will be shown throughout the remainder of the post. This command indicates that the CONTROL and X key are pressed at the same time, and then the CONTROL and S key. This is the basic form of interacting with Emacs. Please follow the built-in tutorial as well as the Guided Tour of Emacs to learn more.

Configuration and Packages

One of the great benefits of Emacs is the simplicity and power of configuration. The core of Emacs configuration is the Initialization File, `init.el`.

In a UNIX environment this file should be placed in `$HOME/.emacs.d/init.el`:

```
$ touch ~/.emacs.d/init.el
```

Meanwhile, in Windows, if the `HOME` environment variable is not set, this file should reside in `C:/.emacs.d/init.el`. See GNU Emacs FAQ for MS Windows > Where do I put my init file? for more info.

Configuration snippets will be presented throughout the post. So, create the init file now if you want to follow along. Otherwise you can find the complete file in the Conclusion.

Packages are used to customize Emacs, which are sourced from a number of repositories. The primary Emacs package repository is MELPA. All of the packages presented in this post will be installed from this repository.

Styling (Themes & More)

To begin, the following configuration snippet provides package installation and installs a theme package:

```
;; init.el --- Emacs configuration

;; INSTALL PACKAGES
;; -----

(require 'package)

(add-to-list 'package-archives
  '("melpa" . "http://melpa.org/packages/") t)

(package-initialize)
(when (not package-archive-contents)
  (package-refresh-contents))
```

```

(defvar myPackages
  '(better-defaults
    material-theme))

(mapc #'(lambda (package)
  (unless (package-installed-p package)
    (package-install package)))
  myPackages)

;; BASIC CUSTOMIZATION
;; -----

(setq inhibit-startup-message t) ;; hide the startup message
(load-theme 'material t) ;; load material theme
(global-linum-mode t) ;; enable line numbers globally

;; init.el ends here

```

The first section of the configuration snippet, `;; INSTALL PACKAGES`, installs two packages called *better-defaults* and *material-theme*. The *better-defaults* package is a collection of minor changes to the Emacs defaults that makes a great base to begin customizing from. The *material-theme* package provides a customized set of styles.

My preferred theme is *material-theme*, so we'll be using that for the rest of the post.

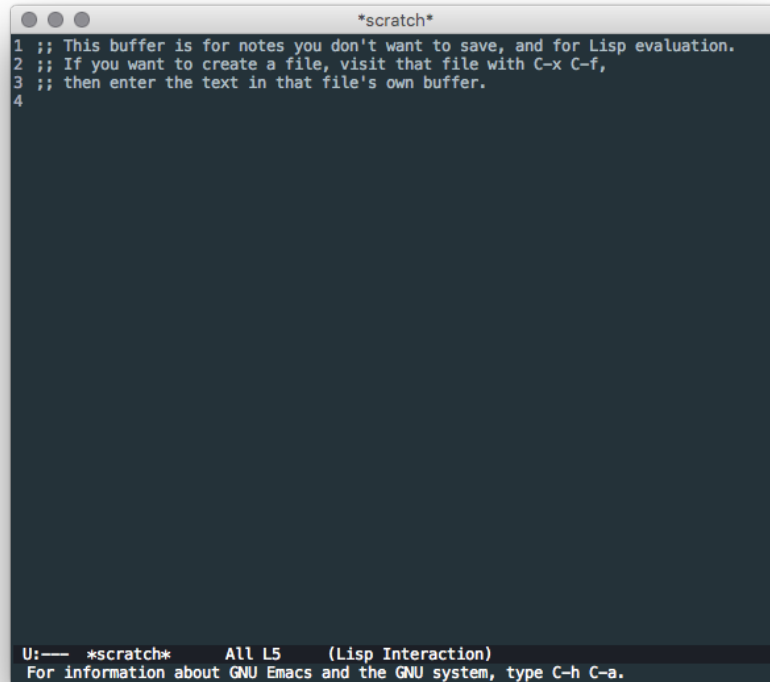
The second section `;; BASIC CUSTOMIZATION`:

1. Disables the startup message (this is the screen with all the tutorial information). *You may want to leave this out until you are more comfortable with Emacs.*
2. Loads the material theme.
3. Enables line numbers globally.

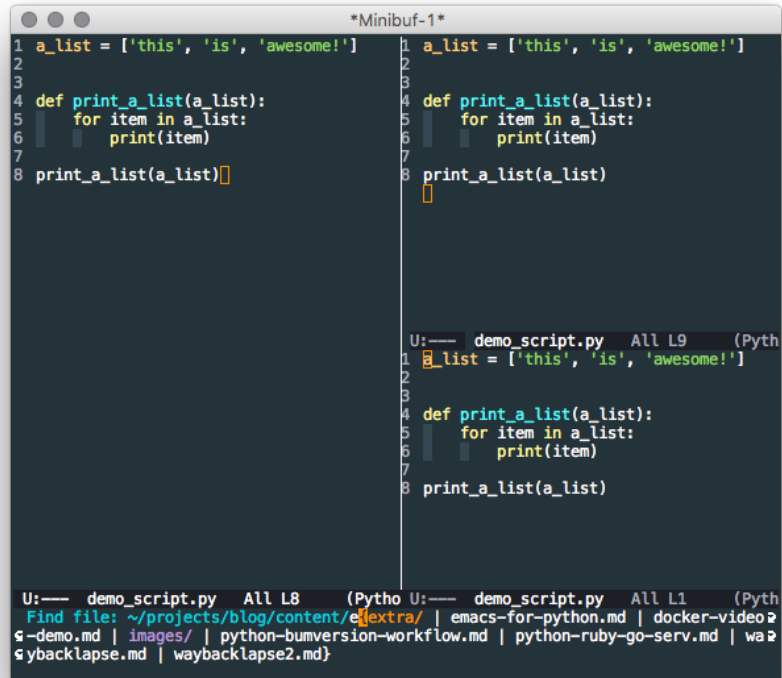
Enabling something globally means that it will apply to all buffers (open items) in Emacs. So if you open a Python, markdown, and/or text file, they will all have line numbers shown. You can also enable things per mode - e.g., `python-mode`, `markdown-mode`, `text-mode`. This will be shown later when setting up Python.

Now that we have a complete *basic* configuration file we can restart Emacs and see the changes. If you placed the `init.el` file in the correct default location it will automatically be picked up.

As an alternative, you can start Emacs from the command-line with `emacs -q --load <path to init.el>`. When loaded, our initial Emacs window looks a bit nicer:



The following image shows some other basic features that come with Emacs out of the box - including simple file searching and split layouts:

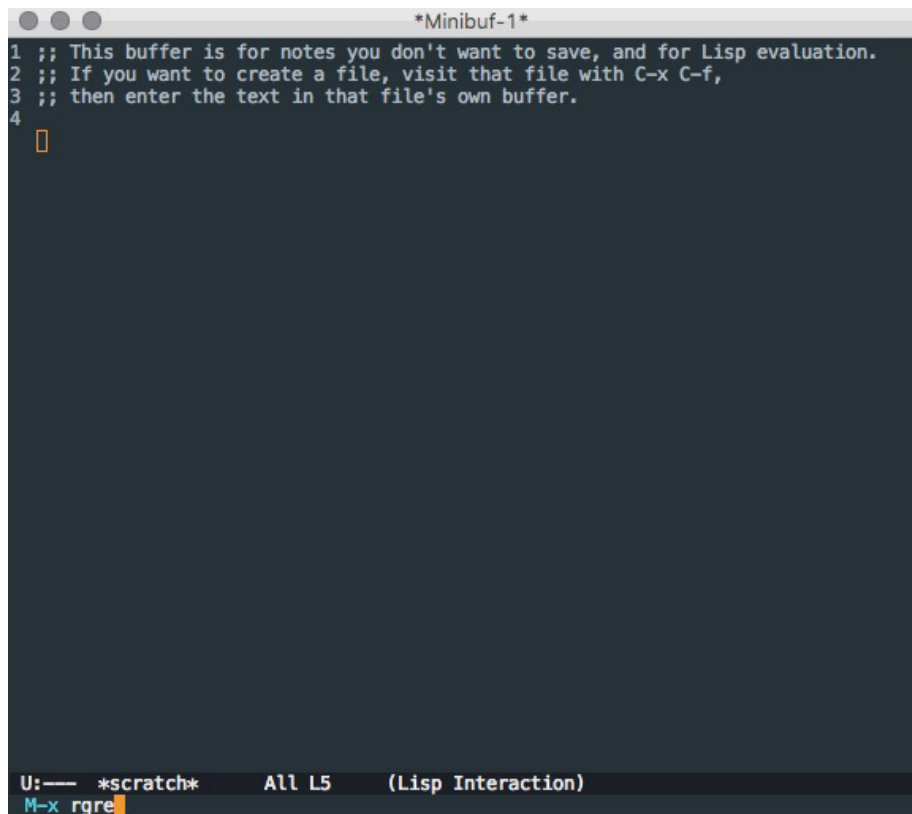


The image shows an Emacs window titled "*Minibuf-1*" with a dark background. It displays the results of a recursive grep search. The search was performed in the directory `~/projects/blog/content/extra/` for the file `demo_script.py`. The results are shown in two columns, each representing a different buffer (L8 and L9). Both buffers contain the same Python code snippet:

```
1 a_list = ['this', 'is', 'awesome!']
2
3
4 def print_a_list(a_list):
5     for item in a_list:
6         print(item)
7
8 print_a_list(a_list)
```

At the bottom of the window, a status bar shows the search command: `U:— demo_script.py All L8 (Pyth` and `U:— demo_script.py All L1 (Pyth`. Below this, a list of files is displayed, including `emacs-for-python.md`, `docker-video`, `demo.md`, `images/`, `python-bumversion-workflow.md`, `python-ruby-go-serv.md`, `waybacklapse.md`, and `waybacklapse2.md`.

One of my favorite simple features of Emacs is being able to do a quick recursive-grep search - `M-x rgrep` For example, say you want to find all instances of the word *python* in any *.md* (markdown) in a given directory:



```
1 ;; This buffer is for notes you don't want to save, and for Lisp evaluation.
2 ;; If you want to create a file, visit that file with C-x C-f,
3 ;; then enter the text in that file's own buffer.
4
U:— *scratch* All L5 (Lisp Interaction)
M-x rgre
```

With this basic configuration complete we can begin to dive into configuring the environment for Python development!

Elpy – Python Development

Emacs is distributed with a python-mode (`python.el`) that provides indentation and syntax highlighting. However, to compete with Python-specific IDE's (Integrated Development Environments), we'll certainly want more. The elpy (Emacs Lisp Python Environment) package provides us with a near complete set of Python IDE features, including:

- Automatic Indentation,
- Syntax Highlighting,
- Auto-Completion,
- Syntax Checking,
- Python REPL Integration,
- Virtual Environment Support, and
- Much more!

To install and enable elpy we need to add a bit of configuration and install

flake8 and jedi using your preferred method for installing Python packages (pip or conda, for example).

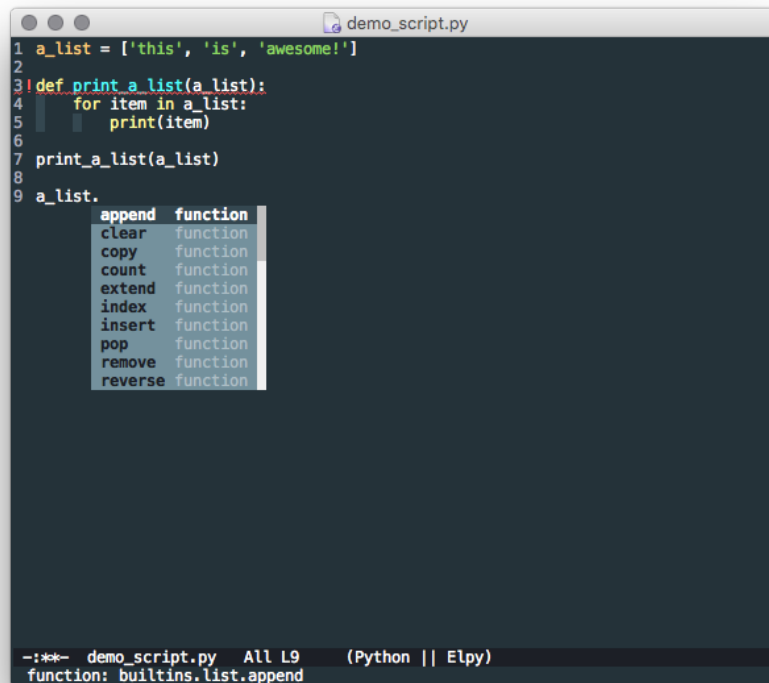
The following will install the elpy package:

```
(defvar myPackages
  '(better-defaults
    elpy ;; add the elpy package
    material-theme))
```

Now just enable it:

```
(elpy-enable)
```

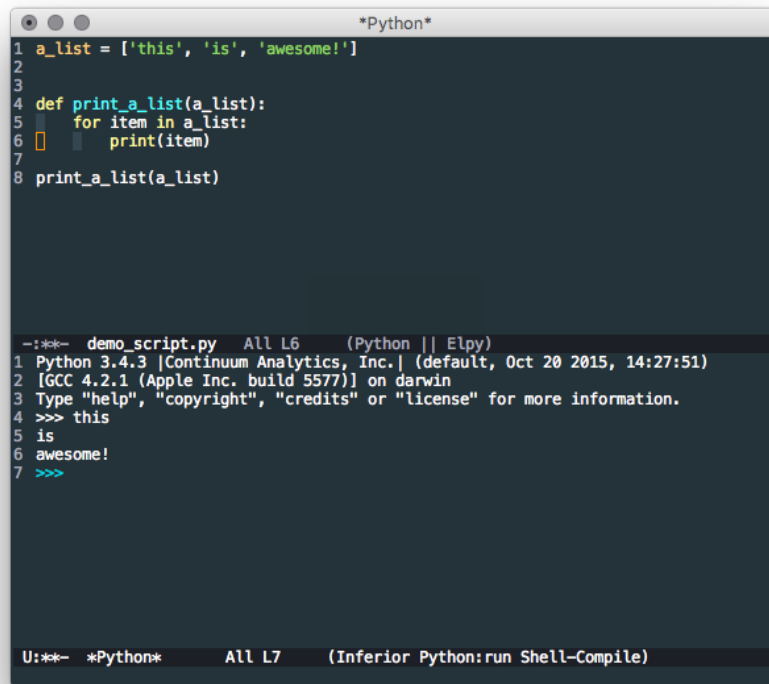
With the new configuration, we can restart Emacs and open up a Python file to see the new mode in action:



Shown in this image are the following features:

- Automatic Indentation (as you type and hit RET lines are auto-indented)
- Syntax Highlighting
- Syntax Checking (error indicators at line 3)
- Auto-Completion (list methods on line 9+)

In addition, let's say we want to run this script. In something like IDLE or Sublime Text you'll have a button/command to run the current script. Emacs is no different, we just type `C-c C-c` in our Python buffer and...



```
*Python*
1 a_list = ['this', 'is', 'awesome!']
2
3
4 def print_a_list(a_list):
5     for item in a_list:
6         print(item)
7
8 print_a_list(a_list)

-:***- demo_script.py  All L6  (Python || Elpy)
1 Python 3.4.3 [Continuum Analytics, Inc.] (default, Oct 20 2015, 14:27:51)
2 [GCC 4.2.1 (Apple Inc. build 5577)] on darwin
3 Type "help", "copyright", "credits" or "license" for more information.
4 >>> this
5 is
6 awesome!
7 >>>

U:***- *Python*  All L7  (Inferior Python:run Shell-Compile)
```

Often we'll want to be running a virtual environment and executing our code using the packages installed there. To use a virtual environment in Emacs, we type `M-x pyvenv-activate` and follow the prompts. You can deactivate a virtualenv with `M-x pyvenv-deactivate`. Elpy provides an interface for debugging the environment and any issues you may encounter with elpy itself. By typing `M-x elpy-config` we get the following dialog, which provides valuable debugging information:

```
*Elpy Config*
1 Elpy Configuration
2
3 Virtualenv.....: None
4 RPC Python.....: 3.4.3 (/Users/kpurdon/miniconda/bin/python)
5 Interactive Python: ipython (/Users/kpurdon/miniconda/bin/ipython)
6 Emacs.....: 24.4.1
7 Elpy.....: 1.8.1
8 Jedi.....: 0.8.1 (0.9.0 available)
9 Rope.....: Not found (0.10.3 available)
10 Importmagic.....: Not found (0.1.3 available)
11 Syntax checker....: flake8 (/Users/kpurdon/miniconda/bin/flake8)
12
13 You have not activated a virtual env. While Elpy supports this, it is
14 often a good idea to work inside a virtual env. You can use M-x
15 pyvenv-activate or M-x pyvenv-workon to activate a virtual env.
16
17 The directory ~/.local/bin/ is not in your PATH, even though you do
18 not have an active virtualenv. Installing Python packages locally will
19 create executables in that directory, so Emacs won't find them. If you
20 are missing some commands, do add this directory to your PATH.
21
22 There is a newer version of Jedi available.
23
24 [run] pip install --user --upgrade jedi
25
26 The importmagic package is not available. Commands using this will not
27 work.
28
29 [run] pip install --user importmagic
30
31 Options
32
33 'Raised' text indicates buttons; type RET or click mouse-1 on a button
U:*~ *Elpy Config* Top L1 (Custom)
```

With that, all of the basics of a Python IDE in Emacs have been covered. Now let's put some icing on this cake!

Additional Python Features

In addition to all the basic IDE features described above, Emacs provides some additional features for Python. While this is not an exhaustive list, PEP8 compliance (with `autopep8`) and integration with IPython/Jupyter will be covered. However, before that let's cover a quick syntax checking preference.

Better Syntax Checking (Flycheck v. Flymake)

By default Emacs+elpy comes with a package called *Flymake* to support syntax checking. However, another Emacs package, *Flycheck*, is available and supports realtime syntax checking. Luckily switching out for *Flycheck* is simple:

```
(defvar myPackages
  '(better-defaults
```

```

elpy
flycheck ;; add the flycheck package
material-theme))

```

and

```

(when (require 'flycheck nil t)
  (setq elpy-modules (delq 'elpy-module-flymake elpy-modules))
  (add-hook 'elpy-mode-hook 'flycheck-mode))

```

Now we get realtime feedback when editing Python code:



```

demo_script.py
1 a_list = ['this', 'is', 'awesome!']
2
3
4 def print_a_list(a_list):
5     for item in a_list:
6         print(item)
7
8?print_a_list(a_list)...
9 a_

-:***- demo_script.py All L9 (Python FlyC:0/1 || Elpy)

```

PEP8 Compliance (Autopep8)

Love it or hate it, PEP8 is here to stay. If you want to follow all or some of the standards, you'll probably want an automated way to do so. The *autopep8* tool is the solution. It integrates with Emacs so that when you save - **C-x C-s** - *autopep8* will automatically format and correct any PEP8 errors (excluding any you wish to).

First, you will need to install the Python package *autopep8* using your preferred method, then add the following Emacs configuration:

```

(defvar myPackages
  '(better-defaults
    elpy
    flycheck
    material-theme
    py-autopep8)) ;; add the autopep8 package

```

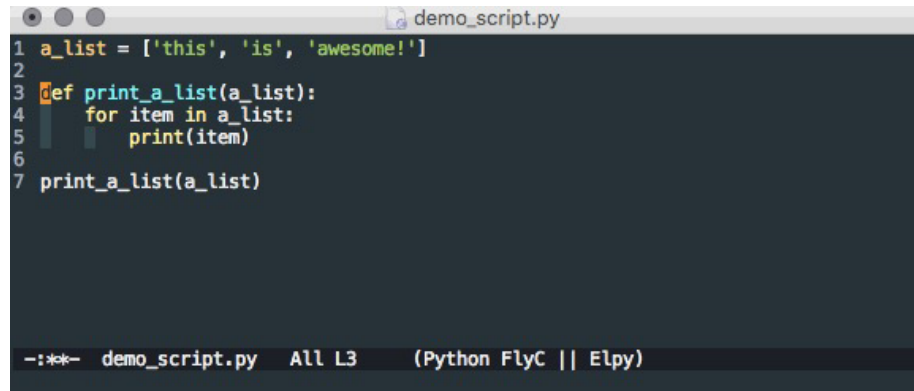
and

```

(require 'py-autopep8)
(add-hook 'elpy-mode-hook 'py-autopep8-enable-on-save)

```

Now (after forcing some pep8 errors) when we save our demo Python file, the errors will automatically be corrected:



```
1 a_list = ['this', 'is', 'awesome!']
2
3 def print_a_list(a_list):
4     for item in a_list:
5         print(item)
6
7 print_a_list(a_list)
```

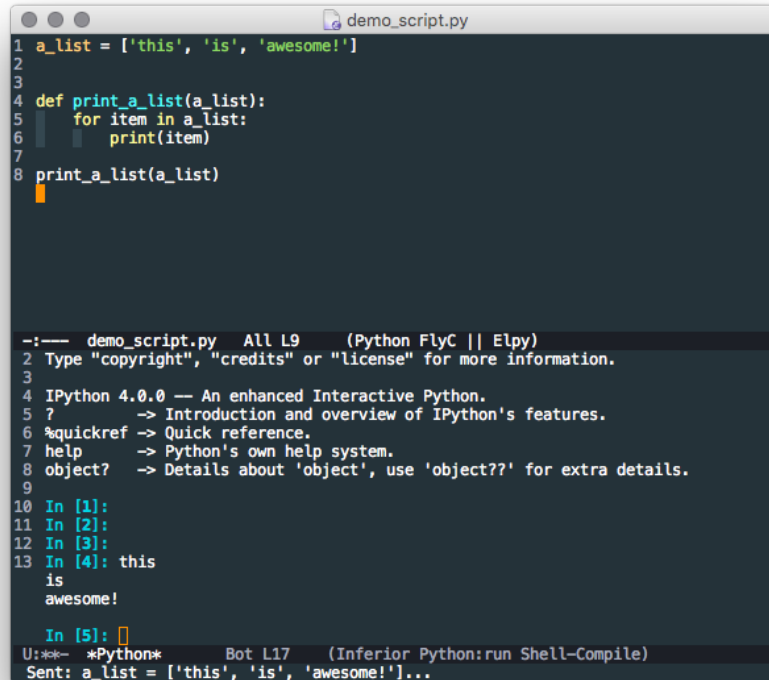
-:***- demo_script.py All L3 (Python FlyC || Elpy)

IPython/Jupyter Integration

Next up is a really cool feature: Emacs integration with the IPython REPL and Jupyter Notebooks. First, let's look at swapping the standard Python REPL integration for the IPython version:

(elpy-use-ipython)

Now when we run our Python code with `C-c C-c` we will be presented with the IPython REPL:

A screenshot of an Emacs window titled 'demo_script.py'. The window is divided into two main sections. The top section contains a Python script with the following code:

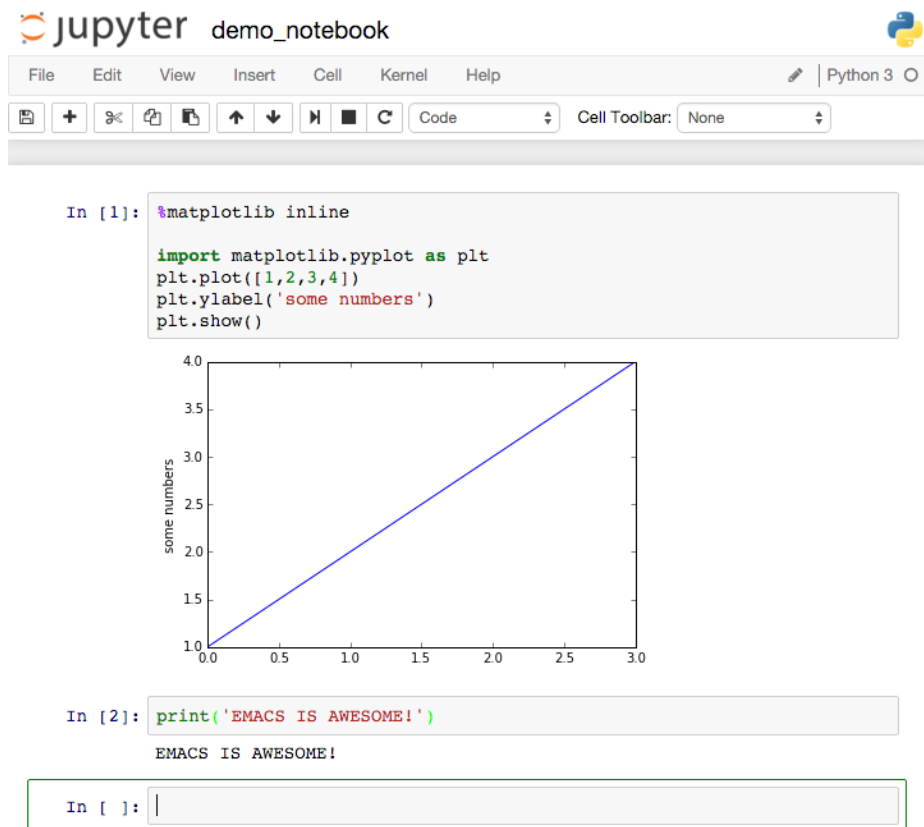
```
1 a_list = ['this', 'is', 'awesome!']
2
3
4 def print_a_list(a_list):
5     for item in a_list:
6         print(item)
7
8 print_a_list(a_list)
```

The bottom section shows the output of the script's execution. It starts with a header line: `-- demo_script.py All L9 (Python FlyC || Elpy)`. Below this is a list of help topics: `Type "copyright", "credits" or "license" for more information.`, `IPython 4.0.0 -- An enhanced Interactive Python.`, `? -> Introduction and overview of IPython's features.`, `%quickref -> Quick reference.`, `help -> Python's own help system.`, and `object? -> Details about 'object', use 'object??' for extra details.`. The execution results are shown as a series of input/output pairs: `In [1]:`, `In [2]:`, `In [3]:`, `In [4]: this`, `is`, `awesome!`, and `In [5]:`. At the bottom, there is a status bar with the text: `U:*** *Python* Bot L17 (Inferior Python:run Shell-Compile)` and `Sent: a_list = ['this', 'is', 'awesome!']...`

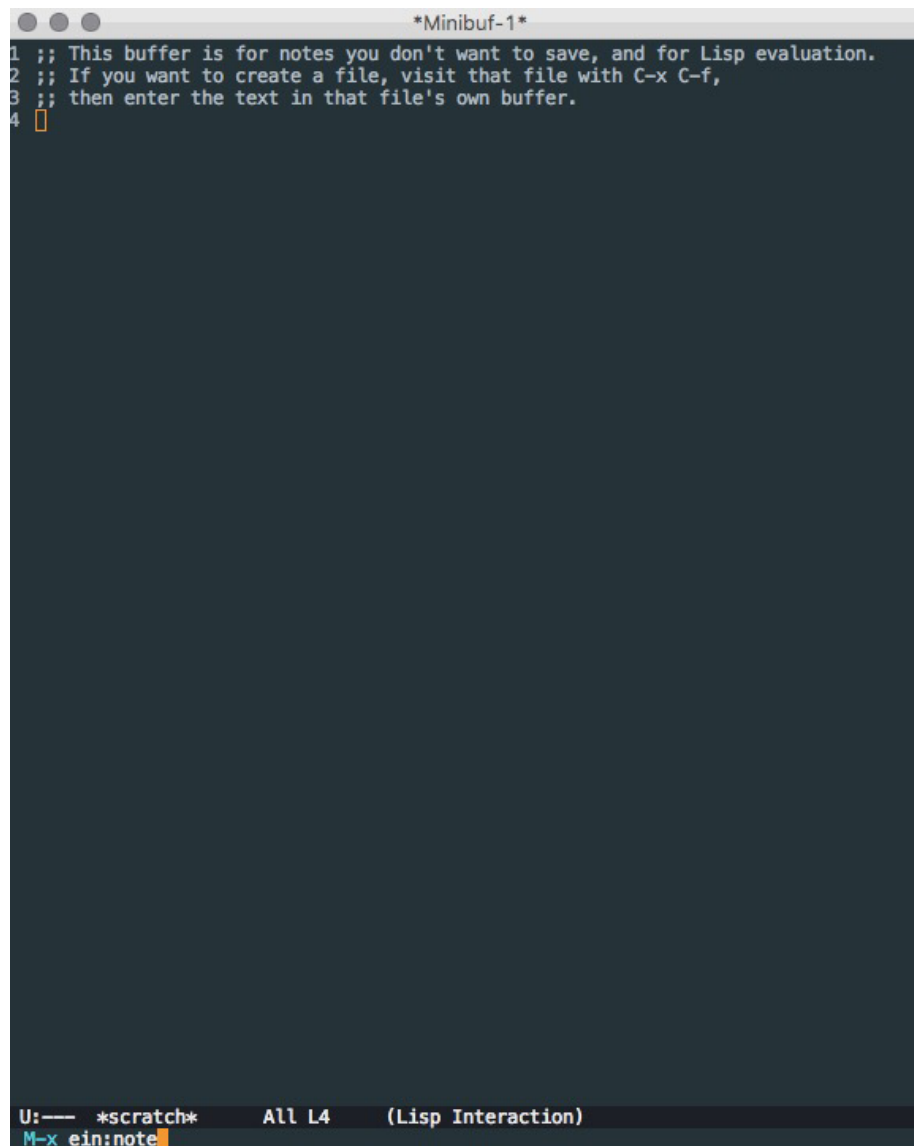
While this is pretty useful on its own, the real magic is the notebook integration. We'll assume that you already know how to launch a Jupyter Notebook server (if not check this out). Again we just need to add a bit of configuration:

```
(defvar myPackages
  '(better-defaults
    ein ;; add the ein package (Emacs ipython notebook)
    elpy
    flycheck
    material-theme
    py-autopep8))
```

The standard Jupyter web interface for notebooks is nice but requires us to leave Emacs to use:



However, we can complete the exact same task by connecting to and interacting with the notebook server directly in Emacs.



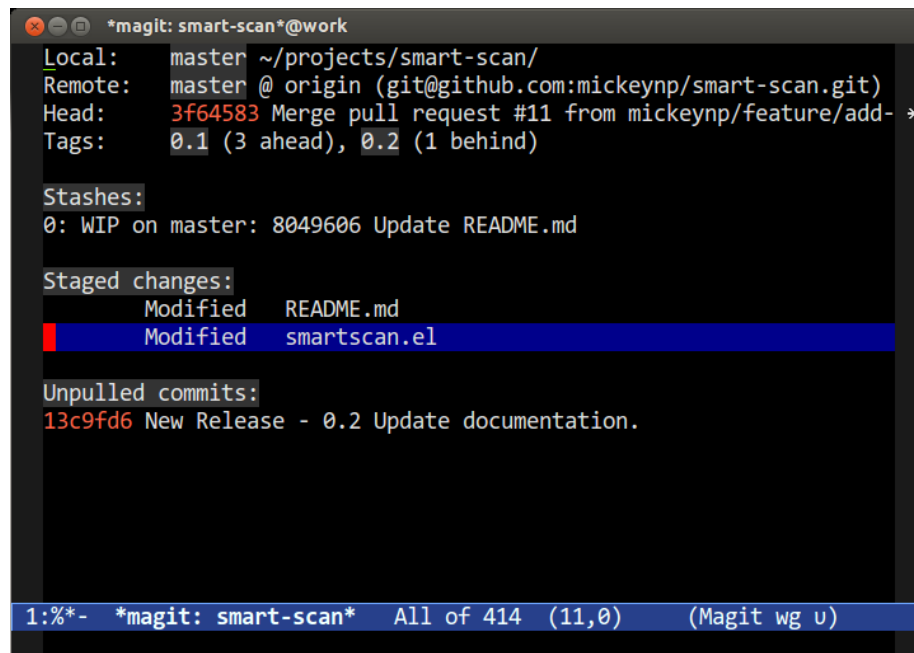
```
*Minibuf-1*
1 ;; This buffer is for notes you don't want to save, and for Lisp evaluation.
2 ;; If you want to create a file, visit that file with C-x C-f,
3 ;; then enter the text in that file's own buffer.
4 
U:— *scratch* All L4 (Lisp Interaction)
M-x ein:note
```

Additional Emacs Features

Now that all of the basic Python IDE features (and some really awesome extras) have been covered, there are a few other things that an IDE should be able to handle. First up is git integration...

Git Integration (Magit)

Magit is the most popular non-utility package on MELPA and is used by nearly every Emacs user who uses git. It's incredibly powerful and far more comprehensive than we can cover in this post. Luckily Mastering Emacs has a great post covering Magit here. The following image is from the Mastering Emacs post and gives you a taste for what the git integration looks like in Emacs:



```
*magit: smart-scan*@work
Local: master ~/projects/smart-scan/
Remote: master @ origin (git@github.com:mickeynp/smart-scan.git)
Head: 3f64583 Merge pull request #11 from mickeynp/feature/add-
Tags: 0.1 (3 ahead), 0.2 (1 behind)

Stashes:
0: WIP on master: 8049606 Update README.md

Staged changes:
    Modified README.md
    Modified smartscan.el

Unpulled commits:
13c9fd6 New Release - 0.2 Update documentation.

1:%*- *magit: smart-scan* All of 414 (11,0) (Magit wg u)
```

Other Modes

One of the major benefits of using Emacs over a Python-specific IDE is that you get compatibility with much more than just Python. In a single day I often work with Python, Golang, JavaScript, Markdown, JSON, and more. Never leaving Emacs and having complex support for all of these languages in a single editor is very efficient. You can check out my personal Emacs configuration here. It includes support for:

- Python
- Golang
- Ruby
- Puppet
- Markdown
- Dockerfile
- YAML

- Web (HTML/JS/CSS)
- SASS
- NginX Config
- SQL

In addition to lots of other Emacs configuration goodies.

Emacs In The Terminal

After learning Emacs you'll want Emacs keybindings everywhere. This is as simple as typing `set -o emacs` at your bash prompt. However, one of the powers of Emacs is that you can run Emacs itself in headless mode in your terminal. This is my default environment. To do so, just start Emacs by typing `emacs -nw` at your bash prompt and you'll be running a headless Emacs.

Conclusion

As you can see, Emacs is clearly the best editor... To be fair, there are a lot of great options out there for Python IDEs, but I would absolutely recommend learning either Vim or Emacs as they are by far the most versatile development environments possible. I said I'd leave you with the complete Emacs configuration, so here it is:

```
;; init.el --- Emacs configuration

;; INSTALL PACKAGES
;; -----

(require 'package)

(add-to-list 'package-archives
  '("melpa" . "http://melpa.org/packages/") t)

(package-initialize)
(when (not package-archive-contents)
  (package-refresh-contents))

(defvar myPackages
  '(better-defaults
    ein
    elpy
    flycheck
    material-theme
    py-autopep8))
```

```

(mapc #'(lambda (package)
  (unless (package-installed-p package)
    (package-install package)))
  myPackages)

;; BASIC CUSTOMIZATION
;; -----

(setq inhibit-startup-message t) ;; hide the startup message
(load-theme 'material t) ;; load material theme
(global-linum-mode t) ;; enable line numbers globally

;; PYTHON CONFIGURATION
;; -----

(elpy-enable)
(elpy-use-ipython)

;; use flycheck not flymake with elpy
(when (require 'flycheck nil t)
  (setq elpy-modules (delq 'elpy-module-flymake elpy-modules))
  (add-hook 'elpy-mode-hook 'flycheck-mode))

;; enable autopep8 formatting on save
(require 'py-autopep8)
(add-hook 'elpy-mode-hook 'py-autopep8-enable-on-save)

;; init.el ends here

```

Hopefully this configuration will spark your Emacs journey!