Elpy Documentation

Release 1.30.0

Jorgen Schäfer

Sep 06, 2019

Contents

Elpy is the Emacs Python Development Environment. It aims to provide an easy to install, fully-featured environment for Python development.

Contents:

CHAPTER 1

Introduction

1.1 Overview

Elpy is an extension for the Emacs text editor to work with Python projects. This documentation tries to explain how to use Elpy to work on Python project using Emacs, but it does not aim to be an introduction to either Emacs or Python.

You can read a quick tour of Emacs, or read the built-in tutorial by running `C-h t` in the editor. That is, you hold down the `control` key and hit `h` (the canonical *help* key in Emacs), release both, and hit `t` (for tutorial).

For Python, you can read the basic tutorial. If you already know Python, you should check out some best practices.

Once you have these basics, you can go on to install Elpy.

1.2 Installation

1.2.1 With use-package

Simply add the following lines to you `.emacs`:

```
(use-package elpy
  :ensure t
  :init
  (elpy-enable))
```

Or if you want to defer Elpy loading:

```
(use-package elpy
  :ensure t
  :defer t
  :init
  (advice-add 'python-mode :before 'elpy-enable))
```

1.2.2 Manually from Melpa

The main Elpy package is installed via the Emacs package interface, `package.el`. First, you have to add Elpy's package archive to your list of archives, though. Add the following code to your `.emacs` file and restart Emacs:

```
(require 'package)
(add-to-list 'package-archives
             '("melpa-stable" . "https://stable.melpa.org/packages/"))
```

Now you can run *M-x package-refresh-contents* to download a fresh copy of the archive contents, and `M-x package-install RET elpy RET` to install elpy. If you want to enable Elpy by default, you can simply add the following to your .emacs:

```
(package-initialize)
(elpy-enable)
```

Congratulations, Elpy is now successfully installed!

1.2.3 From apt (Debian 10 an Ubuntu 18.10)

Users of Debian 10 or Ubuntu 18.10 can skip the instructions above this line and may simply install Elpy and all of its recommended dependencies with the following command::

```
sudo apt install elpa-elpy
```

Elpy can then be activated by running `M-x elpy-enable`. This can be made automatic by adding the following to your `.emacs`:

```
(elpy-enable)
```

In order to use all the features (such as navigation with `M-.`), you'll need to install some python libraries. You can do that easily by typing `M-x elpy-config RET`, and following the instructions.

CHAPTER 2

Quickstart

Once installed, Elpy will automatically provide code completion, syntax error highlighting and function signature (in the modeline) for python files.

2.1 Useful keybindings

Elpy has quite a lot of keybindings, but the following ones should be enough to get you started:

C-c C-c evaluates the current script (or region if something is selected) in an interactive python shell. The python shell is automatically displayed aside of your script (if not already there).

C-RET evaluates the current statement (current line plus the following nested lines).

C-c C-z switches between your script and the interactive shell.

C-c C-d displays documentation for the thing under cursor (function or module). The documentation will pop in a different buffer, that can be closed with *q*.

2.2 Go further

Elpy offers a lot of features, including code navigation, debugging, testing, refactoring, profiling and support for virtual environments. Feel free to explore the documentation, everything is there !

CHAPTER 3

Concepts

3.1 Configuration

You can easily configure Elpy to your own preferences. All *Customize Options* below are accessible via this interface. Elpy builds heavily upon existing extensions for Emacs. The configuration interface tries to include the options for those as well.

M-x elpy-config
: Show the current Elpy configuration, point out possible problems, and provide a quick interface to relevant customization options.

  Missing packages can be installed right from this interface. Be aware that this does use your currently-selected virtual env. If there is no current virtual env, it will suggest installing packages globally. This is rarely what you want.

3.2 The RPC Process

Elpy works by starting a Python process in the background and communicating with it through a basic *Remote Procedure Call* (RPC) interface. Ideally, you should never see this process and not worry about it, but when things don't work as expected, it's good to know what's going on in the background.

Every project and virtual env combination gets their own RPC process. You can see them both in the process list (`M-x list-process`) as well as in the buffer list (`C-x C-b`) as buffers named `*elpy-rpc[...]*`.

By default, Elpy will also find the library root of the current file and pass that to the RPC functions. The library root is the directory from which the current file can be imported.

There are a few options and commands related to the RPC process.

M-x elpy-rpc-restart
: Close all running RPC processes. Elpy will re-start them on demand with current settings.

elpy-rpc-python-command (Customize Option)
The Python interpreter Elpy should use to run the RPC process. This defaults to `"python"`, which should be correct for most cases, as a virtual env should make that the right interpreter.

Please do note that this is *not* an interactive interpreter, so do not set this to `"ipython"` or similar.

elpy-rpc-large-buffer-size (Customize Option)
The size in character starting from which Elpy will transfer buffer contents via temporary files instead of via the normal RPC mechanism.

When Elpy communicates with the RPC process, it often needs to tell Python about the contents of the current buffer. As the RPC protocol encodes all data in JSON, this can be a bit slow for large buffers. To speed things up, Elpy can transfer file contents in temporary files, which is a lot faster for large files, but slightly slower for small ones.

elpy-rpc-pythonpath (Customize Option)
A directory to add to the `PYTHONPATH` for the RPC process. This should point to the directory where the `elpy` module is installed. Usually, there is no need to change this.

3.3 Backends

For introspection and analysis of Python sources, Elpy mainly relies on Jedi. Jedi is known to have some problems coping with badly-formatted Python.

3.4 Virtual Envs

Elpy has full support for Python's virtual envs. Every RPC process is associated with a specific virtual env and completions are done based on that environment.

Outside of RPC processes, though, it is not easy to have more than one virtual env active at the same time. Elpy allows you to set a single global virtual env and change it whenever you like, though.

M-x pyvenv-workon

M-x pyvenv-activate

M-x pyvenv-deactivate
These commands are the main interaction point with virtual envs, mirroring the normal **activate** and **deactivate** commands of virtual envs and the **workon** command of `virtualenvwrapper.sh`.

The *pyvenv-workon* command will allow auto-completion of existing virtual envs and also supports virtualenvwrapper's setup hooks to set environment variables.

Elpy won't pollute your Emacs command namespaces, but it might be an idea to create an alias for the workon command:

```
(defalias 'workon 'pyvenv-workon)
```

3.5 Modules

As the last concept, Elpy has a number of optional features you can enable or disable as per your preferences.

elpy-modules (Customize Option)

The list of modules to activate by default. See the section on *Writing Modules* for details on how to write your own modules.

CHAPTER 4

Editing

4.1 Emacs Basics

Elpy is an extension to Emacs, and as such the standard bindings in Emacs are available. This manual is not meant to be an introduction to Emacs, but this section will still highlight some features in Emacs that are especially useful for Python editing.

Movement keys in Emacs often use `fbnp` for forward, backward, next (down) and previous (up). `k` and `backspace` (`DEL`) are for deleting. These are combined with the `Control`, `Meta` and `Control-Meta` modifiers. Control generally refers to the simplest form. `C-f` moves one character forward. Meta changes this to affect *words*, that is, consecutive sequences of alphanumeric characters. The Control-Meta combination then affects whole expressions.

In the following table, `|` refers to the position of point.

Before	Key	After
`|hello_world`	`C-f`	`h|ello_world`
`|hello_world`	`M-f`	`hello|_world`
`|hello_world`	`C-M-f`	`hello_world|`

Expression-based commands will also work on strings, tuples, dictionaries, or any balanced groups of parentheses. This works for all movement keys (`f`, `b`, `n`, `p`), with *next* and *previous* moving to the next or previous group of parens. It also works with forward and backward deletion (`d` and `DEL`/`<backspace>`, respectively) for character and word groups, but not for expressions. To delete the expression after point, use `C-M-k`. For the expression before point, you can use `C-M-b C-M-k`.

If you enable `subword-mode`, Emacs will also consider CamelCase to be two words instead of one for the purpose of these operations.

In addition to the above, Emacs also supports moving up or down inside nested parentheses groups. `C-M-d` will move *down* into the next enclosed group of parentheses, while `C-M-u` will move *up* to the directly enclosing group of parentheses.

Finally, a lot of Elpy's commands change their behavior when the prefix argument is given. That is, hit `C-u` before the command. In Elpy, the prefix argument often disables any attempt by the command at being smart, in case it would

get it wrong.

4.2 Moving By Indentation

C-down (elpy-nav-forward-block)

C-up (elpy-nav-backward-block)
These commands are used to navigate between lines with same indentation as the current line. Point should be placed on the first non-whitespace character of the line and then use *C-down* to move forward or *C-up* to move backward.

C-left (elpy-nav-backward-indent)

C-right (elpy-nav-forward-indent)
These commands are used to navigate between indentation levels. *C-left* moves point to previous indent level or over previous word. *C-right* moves point to next indent level or over the next word.

4.3 Moving the Current Region

M-down (elpy-nav-move-line-or-region-down)

M-up (elpy-nav-move-line-or-region-up)

M-left (elpy-nav-indent-shift-left)

M-right (elpy-nav-indent-shift-right)
Elpy can move the selected region (or the current line if no region is selected) by using the cursor keys with meta. Left and right will dedent or indent the code, while up and down will move it line-wise up or down, respectively.

CHAPTER 5

IDE Features

5.1 Projects

Elpy supports the notion of *projects*, a related collection of files under a common directory. This common directory is called the project root. A number of Elpy's commands work on all files inside the project root.

C-c C-f (elpy-find-file)
Find a file in the current project. This uses a search-as-you-type interface for all files under the project root.

A prefix argument enables "do what I mean" mode. On an import statement, it will try to open the module imported. Elsewhere in a file, it will look for an associated test or implementation file, and if found, open that. If this fails, either way, it will fall back to the normal find file in project behavior.

If the current file is called `foo.py`, then this will search for a `test_foo.py` in the same directory, or in a `test` or `tests` subdirectory. If the current file is already called `test_foo.py`, it will try and find a `foo.py` nearby.

This command uses find-file-in-project under the hood, so see there for more options.

C-c C-s (elpy-rgrep-symbol)
Search the files in the current project for a string. By default, this uses the symbol at point. With a prefix argument, it will prompt for a regular expression to search.

This is basically a `grep -r` through the project.

In addition to these two commands, *`elpy-check`* also supports optionally checking all files in the current project.

Elpy's idea of the project root and which files belong to a project and which don't can be influenced as well.

M-x elpy-set-project-root
Set the current project root directory. This directory should contain all files related to the current project.

elpy-project-ignored-directories (Customize Option)
When Elpy searches for files in the current project, it will ignore files in directories listed here.

elpy-project-root-finder-functions (Customize Option)
To find the project root, Elpy can utilize a number of heuristics. With this option, you can configure which are used.

To configure Elpy specifically for a single project, you can use Emacs' Directory Variables. Elpy provides a simple interface to this.

M-x elpy-set-project-variable
Set or change the value of a project-wide variable. With a prefix argument, the value for the variable is removed.

This only takes effect in new buffers.

5.2 Completion

When you type Python code, Elpy will try and figure out possible completions and provide them in a suggestion window. If Elpy doesn't do so automatically, you can force it to complete right where you are.

M-TAB (elpy-company-backend)
Provide completion suggestions for a completion at point.

You can use cursor keys or `M-n` and `M-p` to scroll through the options, `RET` to use the selected completion, or `TAB` to complete the common part.

On any completion option, `C-d` or `<f1>` will display a temporary window with documentation. `C-w` will display a temporary window showing the source code of the completion to get some context.

Elpy uses Company Mode for the completion interface, so its documentation is a good place for further information.

elpy-get-info-from-shell (Customize Option)
If t, use the shell to gather docstrings and completions. Normally elpy provides completion and documentation using static code analysis (from jedi). With this option set to t, elpy will add the completion candidates and the docstrings from the associated python shell. This allows to have decent completion candidates and documentation when the static code analysis fails. the static code analysis fails.

5.3 Navigation

Elpy supports some advanced navigation features inside Python projects.

M-. (elpy-goto-definition)
Go to the location where the identifier at point is defined. This is not always easy to make out, so the result can be wrong. Also, the backends can not always identify what kind of symbol is at point. Especially after a few indirections, they have basically no hope of guessing right, so they don't.

C-x 4 M-. (elpy-goto-definition-other-window)
Same as *elpy-go-to-definition* (with the same caveats) but goes to the definition of the symbol at point in other window, if defined.

M-* (pop-tag-mark)
Go back to the last place where `M-.` was used, effectively turning `M-.` and `M-*` into a forward and backward motion for definition lookups.

C-c C-o (elpy-occur-definitions)
Search the buffer for a list of definitions of classes and functions.

If you use an Emacs version superior to 25, elpy will define the necessary backends for the xref package.

M-. (xref-find-definitions)
Find the definition of the identifier at point.

C-x 4 . (xref-find-definition-other-window)
Like `M-.` but switch to the other window.

C-x 5 . (xref-find-definition-other-frame)
Like M-. but switch to the other frame.

M-, (xref-pop-marker-stack)
Go back to the last place where M-. was used, effectively turning M-. and M-, into a forward and backward motion for definition lookups.

M-? (xref-find-references)
Find references for an identifier of the current buffer.

C-M-. (xref-find-apropos)
Find all meaningful symbols that match a given pattern.

5.4 Interactive Python

Emacs can run a Python interpreter in a special buffer, making it much easier to send code snippets over. Elpy provides additional functionality to seamlessly work with interactive Python in a style similar to ESS.

5.4.1 Interpreter Setup

Elpy uses the Python interpreter setup from the Emacs `python` package. This section briefly summarizes some common setups; add the one you need to your `.emacs` file. Note that the code below (and Elpy in general) require at least Emacs 24.4.

Use the Python standard interpreter (default):

```
(setq python-shell-interpreter "python"
      python-shell-interpreter-args "-i")
```

Use Jupyter console (recommended for interactive Python):

```
(setq python-shell-interpreter "jupyter"
      python-shell-interpreter-args "console --simple-prompt"
      python-shell-prompt-detect-failure-warning nil)
(add-to-list 'python-shell-completion-native-disabled-interpreters
             "jupyter")
```

Use IPython:

```
(setq python-shell-interpreter "ipython"
      python-shell-interpreter-args "-i --simple-prompt")
```

Note that various issues with plotting have been reported when running IPython 5 in Emacs under Windows. We recommend using Jupyter console instead.

If you have an older version of IPython and the above code does not work for you, you may also try:

```
(setenv "IPY_TEST_SIMPLE_PROMPT" "1")
(setq python-shell-interpreter "ipython"
      python-shell-interpreter-args "-i")

As an IPython_ user, you might be interested in the `Emacs IPython
Notebook`_ or an `Elpy layer`_ for Spacemacs_, too.
```

5.4.2 The Shell Buffer

C-c C-z (elpy-shell-switch-to-shell)
Switch to buffer with a Python interpreter running, starting one if necessary.

By default, Elpy tries to find the root directory of the current project (git, svn or hg repository, python package or projectile project) and starts the python interpreter here. This behaviour can be suppressed with the option `elpy-shell-use-project-root`.

M-x elpy-shell-toggle-dedicated-shell
By default, python buffers are all attached to a same python shell (that lies in the **Python** buffer), meaning that all buffers and code fragments will be send to this shell. *elpy-shell-toggle-dedicated-shell* attaches a dedicated python shell (not shared with the other python buffers) to the current python buffer. To make this the default behavior (like the deprecated option *elpy-dedicated-shells* did), use the following snippet:

```
(add-hook 'elpy-mode-hook (lambda () (elpy-shell-toggle-dedicated-shell 1)))
```

M-x elpy-shell-set-local-shell
Attach the current python buffer to a specific python shell (whose name is asked with completion). You can use this function to have one python shell per project, with:

```
(add-hook 'elpy-mode-hook (lambda () (elpy-shell-set-local-shell (elpy-project-
↪root))))
```

C-c C-k (elpy-shell-kill)
Kill the associated python shell.

C-c C-K (elpy-shell-kill-all)
Kill all active python shells.

5.4.3 Evaluating code fragments

Elpy provides commands to send the current Python statement (`e`), function definition (`f`), class definition (`c`), top-level statement (`s`), group of Python statements (`g`), cell (`w`), region (`r`), or buffer (`b`) to the Python shell for evaluation. These commands are bound to prefix `C-c C-y`, followed by the single character indicating what to send; e.g., `C-c C-y e` sends the Python statement at point.

Each of the commands to send code fragments to the shell has four variants, one for each combination of: whether or not the point should move after sending ("step"), and whether or not the Python shell should be focused after sending ("go"). Step is activated by `C-`, go by `S-`. For example:

C-c C-y e (elpy-shell-send-statement)
Send the current statement to the Python shell and keep point position. Here statement refers to the Python statement the point is on, including potentially nested statements and, if point is on an if/elif/else clause, the entire if statement (with all its elif/else clauses).

C-c C-y C-e (elpy-shell-send-statement-and-step)
Send the current statement to the Python shell and move point to first subsequent statement.

Also bound to `C-RET`.

C-c C-y E (elpy-shell-send-statement-and-go)
Send the current statement to the Python shell, keeping point position, and switch focus to the Python shell buffer.

C-c C-y C-S-E (elpy-shell-send-statement-and-step-and-go)
Send the current statement to the Python shell, move point to first subsequent statement, and switch focus to the Python shell buffer.

Elpy provides support for sending multiple statements to the shell.

C-c C-y O (elpy-shell-send-group-and-step)
Send the current or next group of top-level statements to the Python shell and step. A sequence of top-level statements is a group if they are not separated by empty lines. Empty lines within each top-level statement are ignored.

If the point is within a statement, send the group around this statement. Otherwise, go to the top-level statement below point and send the group around this statement.

C-c C-y W (elpy-shell-send-codecell-and-step)
Send the current code cell to the Python shell and step. A code cell is a piece of code surrounded by special separator lines; see below. For example, you can insert two lines starting with `##` to quickly send the code in-between.

elpy-shell-codecell-beginning-regexp (Customize Option)
Regular expression for matching a line indicating the beginning of a code cell. By default, `##.*` is treated as a beginning of a code cell, as are the code cell beginnings in Python files exported from IPython or Jupyter notebooks (e.g., `# <codecell>` or `# In[1]:`).

elpy-shell-cell-boundary-regexp (Customize Option)
Regular expression for matching a line indicating the boundary of a cell (beginning or ending). By default, `##.*` is treated as a cell boundary, as are the boundaries in Python files exported from IPython or Jupyter notebooks (e.g., `# <markdowncell>`, `# In[1]:`, or `# Out[1]:`).

Note that *elpy-shell-codecell-beginning-regexp* must also match the cell boundaries defined here.

The functions for sending the entire buffer have special support for avoiding accidental code execution, e.g.:

C-c C-y r (elpy-shell-send-region-or-buffer)
Send the the active region (if any) or the entire buffer (otherwise) to the Python shell and keep point position.

When sending the whole buffer, this command will also escape any uses of the `if __name__ == '__main__'` idiom, to prevent accidental execution of a script. If you want this to be evaluated, pass a prefix argument with `C-u`.

Also bound to `C-c C-c`.

The list of remaining commands to send code fragments is:

C-c C-y s (elpy-shell-send-top-statement)

C-c C-y S (elpy-shell-send-top-statement-and-go)

C-c C-y f (elpy-shell-send-defun)

C-c C-y F (elpy-shell-send-defun-and-go)

C-c C-y c (elpy-shell-send-defclass)

C-c C-y C (elpy-shell-send-defclass-and-go)

C-c C-y o (elpy-shell-send-group)

C-c C-y O (elpy-shell-send-group-and-go)

C-c C-y w (elpy-shell-send-codecell)

C-c C-y W (elpy-shell-send-codecell-and-go)

C-c C-y R (elpy-shell-send-region-or-buffer-and-go)

C-c C-y b (elpy-shell-send-buffer)

C-c C-y B (elpy-shell-send-buffer-and-go)

C-c C-y C-s (elpy-shell-send-top-statement-and-step)

C-c C-y C-S-S (elpy-shell-send-top-statement-and-step-and-go)

C-c C-y C-f (elpy-shell-send-defun-and-step)

C-c C-y C-S-F (elpy-shell-send-defun-and-step-and-go)

C-c C-y C-c (elpy-shell-send-defclass-and-step)

C-c C-y C-S-C (elpy-shell-send-defclass-and-step-and-go)

C-c C-y C-S-O (elpy-shell-send-group-and-step-and-go)

C-c C-y C-W (elpy-shell-send-codecell-and-step-and-go)

C-c C-y C-r (elpy-shell-send-region-or-buffer-and-step)

C-c C-y C-S-R (elpy-shell-send-region-or-buffer-and-step-and-go)

C-c C-y C-b (elpy-shell-send-buffer-and-step)

C-c C-y C-S-B (elpy-shell-send-buffer-and-step-and-go)

5.4.4 Shell feedback

When package eval-sexp-fu is loaded and `eval-sexp-fu-flash-mode` is active, the statements sent to the shell are briefly flashed after running an evaluation command, thereby providing visual feedback.

elpy-shell-echo-input (Customize Option)
Whenever a code fragment is sent to the Python shell, Elpy prints it in the Python shell buffer (i.e., it looks as if it was actually typed into the shell). This behavior can be turned on and off via the custom variable *elpy-shell-echo-input* and further customized via `elpy-shell-echo-input-cont-prompt` (whether to show continuation prompts for multi-line inputs) and `elpy-shell-echo-input-lines-head` / `elpy-shell-echo-input-lines-tail` (how much to cut when input is long).

elpy-shell-echo-output (Customize Option)
Elpy shows the output produced by a code fragment sent to the shell in the echo area when the shell buffer is currently invisible. This behavior can be controlled via *elpy-shell-echo-output* (never, always, or only when shell invisible). Output echoing is particularly useful if the custom variable *elpy-shell-display-buffer-after-send* is set to `nil` (the default value). Then, no window is needed to display the shell (thereby saving screen real estate), but the outputs can still be seen in the echo area.

elpy-shell-display-buffer-after-send (Customize Option)
Whether to display the Python shell after sending something to it (default `nil`).

5.5 Syntax Checking

Whenever you save a file, Elpy will run a syntax check and highlight possible errors or warnings inline.

C-c C-n (elpy-flymake-next-error)

C-c C-p (elpy-flymake-previous-error)
You can navigate between any error messages with these keys. The current error will be shown in the minibuffer.

Elpy uses the built-in Flymake library to find syntax errors on the fly, so see there for more configuration options.

C-c C-v (elpy-check)
Alternatively, you can run a syntax check on the current file where the output is displayed in a new buffer, giving you an overview and allowing you to jump to the errors from there.

With a prefix argument, this will run the syntax check on all files in the current project.

python-check-command (Customize Option)
To change which command is used for syntax checks, you can customize this option. By default, Elpy uses the `flake8` program, which you have to install separately. The *elpy-config* command will prompt you to do this if Elpy can't find the program.

It is possible to create a single virtual env for the sole purpose of installing `flake8` in there, and then simply link the command script to a directory inside your `PATH`, meaning you do not need to install the program in every virtual env separately.

5.6 Documentation

Elpy provides a single interface to documentation.

C-c C-d (elpy-doc)
When point is on a symbol, Elpy will try and find the documentation for that object, and display that. If it can't find the documentation for whatever reason, it will try and look up the symbol at point in pydoc. If it's not there, either, it will prompt the user for a string to look up in pydoc.

With a prefix argument, Elpy will skip all the guessing and just prompt the user for a string to look up in pydoc.

If the *autodoc* module is enabled (not the case by default) the documentation is automatically updated with the symbol at point or the currently selected company candidate.

elpy-autodoc-delay (Customize Option)
The idle delay in seconds until documentation is updated automatically.

5.7 Debugging

Elpy provides an interface to pdb, the builtin Python debugger. Note that this interface is only available for Emacs 25 and above.

C-c C-g g (elpy-pdb-debug-buffer)
Run pdb on the current buffer. If no breakpoints has been set using *elpy-pdb-toggle-breakpoint-at-point*, the debugger will pause at the beginning of the buffer. Else, the debugger will pause at the first breakpoint. Once pdb is started, the pdb commands can be used to step through and look into the code evaluation.

With a prefix argument `C-u`, ignore the breakpoints and always pause at the beginning of the buffer.

C-c C-g b (elpy-pdb-toggle-breakpoint-at-point)
Add (or remove) a breakpoint on the current line. Elpy adds a red circle to the fringe to indicate the presence of a breakpoint. You can then use *elpy-pdb-debug-buffer* to start pdb and pause at each of the breakpoints.

With a prefix argument `C-u`, remove all the breakpoints.

C-c C-g p (elpy-pdb-break-at-point)
Run pdb on the current buffer and pause at the cursor position.

C-c C-g e (elpy-pdb-debug-last-exception)
Run post-mortem pdb on the last exception.

5.8 Testing

Testing is an important part of programming. Elpy provides a central interface to testing, which allows for a good workflow for tests.

Elpy's test interface is built around Emacs' compilation framework. Elpy will run test commands as a compilation job, with all the advantages this brings.

C-c C-t (elpy-test)
Start a test run. This uses the currently configured test runner to discover and run tests. If point is inside a test case, the test runner will run exactly that test case. Otherwise, or if a prefix argument is given, it will run all tests.

M-x elpy-set-test-runner
This changes the current test runner. Elpy supports the standard unittest discovery runner, the Django discovery runner, nose and py.test. You can also write your own, as described in *Writing Test Runners*.

Note on Django runners: Elpy tries to find *manage.py* within your project structure. If it's unable to find it, it falls back to *django-admin.py*. You must set the environment variable `DJANGO_SETTINGS_MODULE` accordingly.

This enables a good workflow. You write a test and use `C-c C-t` to watch it fail. You then go to your implementation file, for example using `C-u C-c C-f`, and make the test pass. You can use a key bound to `recompile` (I use `<f5>` for this) to just re-run that one test. Once that passes, you can use `C-c C-t` again to run all tests to make sure they all pass as well. Repeat.

For an even more automated way, you can use tdd.el, which will run your last compile command whenever you save a file.

5.9 Refactoring

Elpy supports various forms of refactoring Python code.

C-c C-e (elpy-multiedit-python-symbol-at-point)
Edit all occurrences of the symbol at point at once. This will highlight all such occurrences, and editing one of them will edit all. This is an easy way to rename identifiers.

If the backend does not support finding occurrences (currently only Jedi does), or if a prefix argument is given, this will edit syntactic occurrences instead of semantic ones. This can match more occurrences than it should, so be careful. You can narrow the current buffer to the current function using `C-x n d` to restrict where this matches.

Finally, if there is a region active, Elpy will edit all occurrences of the text in the region.

C-c C-r f (elpy-format-code)
Format code using the available formatter.

This command formats code using yapf , autopep8 or black formatter. If a region is selected, only that region is formatted. Otherwise current buffer is formatted.

yapf and autopep8 can be configured with style files placed in the project root directory (determined by `elpy-project-root`).

C-c C-r r (elpy-refactor)
Run the Elpy refactoring interface for Python code.

This command uses rope package and provides various refactoring options depending on the context.

5.10 Django

Elpy has basic Django support such as parsing either *manage.py* or *django-admin.py* (If it does not find *manage.py* it falls back to *django-admin.py*) for command completion assistance. Can also start *runserver* automatically and you can give an ip address and port.

C-c C-x c (elpy-django-command)
Choose what command you'd like to run via *django-admin.py* or *manage.py*.

C-c C-x r (elpy-django-runserver)
Start the development server command, *runserver*. Default arguments are *127.0.0.1* for ip address and *8000* for port. These can be changed via `elpy-django-server-ipaddr` and `elpy-django-server-port`.

5.11 Profiling

Elpy allows one to profile asynchronously python scripts using *cProfile*.

M-x elpy-profile-buffer-or-region
Send the current buffer or region to the profiler and display the result with `elpy-profile-visualizer`. The default visualizer is snakeviz, a browser-based graphical profile viewer that can be installed with *pip install snakeviz*. If the profiling fails, the python error output is displayed.

CHAPTER 6

Extending Elpy

6.1 Writing Modules

Modules are a way of easily extending Elpy with modular extensions. In essence, a module is a function which is called once to initialize itself globally, then once every time elpy-mode is enabled or disabled, and also once if elpy is disabled globally.

To achieve this, a module function receives one or more arguments, the first of which is the command specifier symbol, which can be one of the following:

`global-init` Called once, when Elpy is enabled using `elpy-enable`.

`global-stop` Called once, when Elpy is disabled using `elpy-disable`.

`buffer-init` Called in a buffer when `elpy-mode` is enabled.

`buffer-stop` Called in a buffer when `elpy-mode` is disabled.

To activate a module, the user has to add the function to *`elpy-modules`*.

6.2 Writing Test Runners

A test runner is a function that receives four arguments, described in the docstring of `elpy-test-at-point`. If only the first argument is given, the test runner should find tests under this directory and run them. If the others are given, the test runner should run the specified test only, or as few as it can.

Test runners should use an interactive spec of `(interactive (elpy-test-at-point))` so they can be called directly by the user. For their main work, they can use the helper function `elpy-test-run`. See the `elpy-test-discover-runner` for an example.

To make it possible to set the test runner as a file-, directory- or project-local variable, the function symbol should get the `elpy-test-runner` property with a value of `t`.

6.3 Running Tests:

You can set up a working environment for Elpy using `pip` and `cask`. After installing Cask, create a new virtual environment and run the `setup` script in it:

```
virtualenv ~/.virtualenvs/elpy
source ~/.virtualenvs/elpy/bin/activate
./scripts/setup
```

You can now run `./scripts/test` to run Elpy's test suite.

CHAPTER 7

Indices and tables

- genindex
- search

Index

L

P

X