emacswiki.org

EmacsWiki: Python Programming In Emacs

10-13 minutes

This page collects information for creating a usable Python programming environment in Emacs.

Quick start

Emacs already has out-of-the-box Python support via `‘python-mode’`. The IDE packages listed below can be used to set up a more complete environment quickly.

Python Modes

There are a number of Python major modes for Emacs. As well as basic editing these all provide a range of IDE-like features, relying on a mix of native Emacs features and external Emacs/Python packages:

- python.el, which comes with Emacs 24.2 and up.
- python-mode.el.
- 'loveshack' python.el from Emacs 24.1 and before has a number of fans as well.

IDE packages

These provide powerful and relatively complete environments by combining and customizing other packages, both Emacs Lisp and Python, and resolving conflicts between them.

- Elpy
- anaconda-mode
- Python layer for Spacemacs. Builds on anaconda-mode and several other packages.
- python-mode.el. Replaces the `‘python-mode’` in python.el.

Support Features

Various features that can be added to or improved in Emacs. Note that the IDE packages listed above provide many of these using packages referred to below.

Virtual Environments

There is some built-in support for virtual environments in python.el, but these packages provide more features.

- pyvenv - used in elpy, written by the same author.
- python-environment - used in jedi, written by the same author
- pyenv-mode - uses pyenv, written by the author of anaconda-mode.
- virtualenvwrapper

- virtualenv.el - deprecated.

To automatically activate virtual environment you can use:

- auto-virtualenvwrapper - uses virtualenvwrapper
- auto-virtualenv - uses pyvenv

Indentation

The defaults should see you compliant with PEP8 but see IndentingPython for detail.

Comment/Uncomment Region

If you have `‘transient-mark-mode’` on, you can just use `‘comment-dwim’`: select a region and hit `M-;’. The DoWhatIMean means that it will comment or uncomment the region as appropriate. If you do not have `‘transient-mark-mode’` on by default, you can hit C-SPC twice to activate it temporarily.

python-mode.el also provides `py-comment-region and commands to comment/uncomment all known forms, def, block, clause etc.

Completion

Note that some IDE packages provide completion, e.g. Elpy provides completion using either jedi or rope as a backend to CompanyMode.

You can also configure completion by using either jedi or rope as a backend to either CompanyMode or AutoComplete.

For jedi:

- company-jedi provides a backend for CompanyMode.
- Jedi.el provides a backend for AutoComplete.

For rope:

- Ropemacs can be used as a backend for both CompanyMode and AutoComplete.

An experiment:

- AutoComplete support in IPython shell buffers

Code navigation

The tools built on jedi support using it to find definitions. jedi-direx provides a tree-style source code viewer for Python buffers but is not on MELPA.

Helm provides an interface for navigating buffers via `‘helm-semantic-or-imenu’`.

helm-cscope provides a Helm interface to xcscope. pycscope supports generating the required cscope index of Python source trees.

Code generation helpers

yasnippet

Yasnippet comes with a broad set of templates for Python.

sphinx-doc

sphinx-doc supports inserting and updating docstring skeletons as used by Sphinx.

python-docstring

python-docstring is a minor mode for intelligently reformatting (refilling) and highlighting Python docstrings. It understands both epytext and Sphinx formats (even intermingled!), so it knows how to reflow them correctly. It will also highlight markup in your docstrings, including epytext and reStructuredText.

Lint, style and syntax checkers

Both Flycheck and FlyMake can be used to wrap checkers such as pep8, pyflakes (flake8 includes both), pylint and pychecker.

Flycheck

Flycheck integrates flake8 and pylint. It can be used with multiple checkers per buffer.

Flymake

pylint

Pylint can be configured to use FlyMake, as documented at http://docs.pylint.org/ide-integration.

flake8 / pyflakes

flymake-python-pyflakes is a FlyMake handler using pyflakes or flake8, although the author suggests you might be better served using Flycheck.

Elpy supports using FlyMake with flake8.

Using multiple checkers

Flymake only natively supports a single checker per buffer. However, more than one code checker can be applied by using a wrapper script that runs the desired tools and combines their output. This has lots of issues, one being that flymake does not seem to show more than one error message per line of code, meaning that an error or warning which is intentionally left unfixed can mask an error or warning that would get more attention.

One of the best-maintained such scripts is lintrunner.py. There are others here, here, here, and here.

Refactoring

Refactoring libraries

Ropemacs is a library using Pymacs to talk with the Rope refactoring library.

Jedi.el also has some refactoring support.

Reformatting and PEP8 conformance

yapf

Yapf attempts to format Python code to the best formatting that conforms to a style guide, even if the original code didn’t violate the style guide. The style guide can be customized; predefined ones include pep8 and google.

Both py-yapf and Elpy support applying yapf to the current buffer.

autopep8

autopep8 formats Python code to conform to the PEP 8 style guide using the pep8 tool.

Both py-autopep8 and Elpy support applying autopep8 to the current buffer.

Tidying imports

py-isort using isort.

Elpy supports using importmagic.

Running tests

nose and nosemacs support running nose tests.

Elpy supports the standard unittest discovery runner, the Django discovery runner, nose and py.test

pytest supports the pytest Python package, thus tests written with pytest, nose, unittest and doctest style test suites, including Django and trial.

Reporting test coverage

pycoverage generates reports using coverage and provides a minor mode for displaying coverage by overriding `‘linum-mode’`. Alternatively, a Flycheck checker could be used to display coverage.

Debugging

gud

The native `‘python-mode’` supports pdb tracking via the GrandUnifiedDebugger: when you execute a block of code that contains some call to Python’s pdb (or ipdb) it will prompt the block of code and will follow the execution of pdb marking the current line with an arrow.

To debug a script run the Emacs command “M-x pdb” and invoke Python’s pdb as “python -m pdb foo.py”

realgud

realgud provides support for pdb, ipdb and trepanning debuggers. Its features not in the gud include single-keystroke debugger commands inside the source code.

Interactive environments - Shells, REPLs and notebooks

Using IPython as the Python shell

In Emacs's native python-mode, use:

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

This should work with any recent IPython, including on Windows.

python-mode.el also comes with IPython support.

Emacs IPython Notebook (EIN)

Emacs IPython Notebook (EIN) provides a IPython Notebook client and integrated REPL (like SLIME) in Emacs. It is available on MELPA as 'ein'. Replaces tkf's EIN, which it was forked from to keep up with IPython/Jupyter development.

ob-python

ob-python provides Org-Babel support for evaluating Python source code. Python source code blocks in Org Mode can be used to define functions, filter and analyze data, create graphics and figures, and produce reproducible research papers using a style similar to literate programming. It is included in the ‘org-plus-contrib’ package from the Org Mode ELPA or MELPA.

ob-ipython

ob-ipython provides Org-Babel support for evaluating Python source code using an IPython kernel. It provides similar features to ob-python (and tries to be more robust) as well as IPython-specific features like magics.

LaTeX or MarkDown

emacs-ipython, an Emacs extension that allows execution of python code inside a LaTeX or MarkDown buffer and display its results, text or graphic in the section below. The extension uses Pymacs to connect to an ipython kernel to execute code.

Live coding

live-py-mode is a Python minor mode supporting live coding, inspired by Bret Victor’s "Inventing on Principle" .

Cell-mode

Some packages provide MATLAB-like cells support: navigation between cell code blocks and evaluation.

python-x also provides some additional features

python-cell provides Matlab-like cells in python buffers

Viewing generated documentation

This refers to documentation generated by pydoc and similar tools.

Emacs' native python-mode supports ElDoc via `‘python-eldoc-at-point’`. This returns documentation for object at point by using the inferior python subprocess to inspect its documentation.

pydoc supports generating and viewing pydoc documentation either in an Emacs help buffer, with enhancements like linking to code and coloring for readability, or in a web browser.

helm-pydoc provides a helm interface to pydoc.

Elpy supports viewing documentation from jedi/rope, falling back to pydoc.

Viewing the official Python documentation

This refers to the documentation at https://www.python.org/doc/.

Using Info

You will first need to obtain Info files for the Python documentation. These can be downloaded pre-built from https://github.com/politza/python-info, or built using the instructions there (They are not available at https://www.python.org/doc/).

pydoc-info, available on MELPA, configures Info to use the Python docs, and provides customized support for Python.

For more basic configuration, so that `‘info-lookup-symbol’` searches the Python docs, customize Info as below.

```
(require 'info-look)

(info-lookup-add-help
 :mode 'python-mode
 :regexp "[[:alnum:]_]+"
 :doc-spec
 '(("(python)Index" nil "")))
```

Using a web browser

Pylookup mode allows searching the Python documentation from Emacs and viewing results in a web browser, either on- or off-line.

IronPython

- Install python-mode
- `(setq py-jython-command "c:/Program Files/IronPython 2.6 for .NET 4.0/ipy.exe")`
- Open a .py file. C-c C-t will toggle “jython”, C-c ! will start a

shell

Cython

See CythonMode

Unicode on Mac OS X

When using Emacs 24.1 on Mac OS X compiled via homebrew. The python-shell always used US-ASCII as encoding. To fix it I used:

```
(setenv "LC_CTYPE" "UTF-8")
(setenv "LC_ALL" "en_US.UTF-8")
(setenv "LANG" "en_US.UTF-8")
```

To determine your encoding in the python-shell use:

```
>>> import sys
'US-ASCII'
```

Not having the right encoding set leads to errors in ipython:

```
ERROR - failed to write data to stream:
<_io.TextIOWrapper name='<stdout>' mode='w'
encoding='US-ASCII'>
```

Editing pip requirements files

pip-requirements is a major mode for editing pip requirements files, with support for syntax highlighting, togglable comments and auto-completion of package names from PyPI.

How to improve this page

Round out support features in line with https://lists.gnu.org/archive/html/emacs-devel/2015-10/msg00669.html

Add sections on: profiling, semantic editing (?),

Extend sections on: refactoring, code navigation

Categories: CategoryProgramming, ProgrammingModes