↖ Topic Index

Emacs support for Python 🚧

Description	Keystroke	Function	Note
Python Support ○ Help & Customization • Tree-Sitter (@GitHub)	Emacs provides support for Python via the built-in **python.el** module which provides the python-mode command. Emacs also support a **tree-sitter** version for python. • There is another, older and buggy external package, **python-mode** which also provide a python-mode command. Do **not** use that! ⚠️ PEL Python 🎛️ Important aspects of Python source code management are customizable with PEL user option variables. **PEL customization for Python**: • Emacs customization group: **pel-pkg-for-python** (To edit change, use **<f12> <f2>** , see below). • **pel-python-tab-width**: The width of a tab used for c-mode files. Defaults to 4. • This concept differs from indentation: you can have an indentation of 3 and tab width of 8: **M-i** will move point to columns that are multiple of 8 **<tab>** will indent to a column that is a multiple of 3. PEL stores this value inside the **tab-width** variable for python-mode buffers. • The values for those user option variables can also be stored inside directory local files and even as file local variables. You can also modify them for each buffer and view their current settings using the commands listed in the following set of rows. See **Σ File/Directory Variables** for more info. • None of the commands below change PEL default; they change the value for the current buffer only. ⌨️ PEL provides the following set of **mode-specific key prefixes**: **<f11> SPC p** , **<f12>** and **M-<f12>** The first one is always available. The other two prefixes are only available in c-mode buffers. The **M-<f12>** prefix helps the typing flow when the next key is a Meta key. For simplification, the **<f11> SPC p** prefix is normally omitted in the table. 🎛️ **Tree-Sitter Support** 🚧: when the **pel-use-tree-sitter** user-option is set to t, PEL downloads, installs and provides tree-sitter support via: 📦 **tree-sitter** and **tree-sitter-langs** external packages . These are installed automatically by PEL when **pel-use-tree-sitter** is on. **Other Requirements**: • Emacs with dynamic module loading, and built with tree-sitter support. tree-sitter library must be installed separately. • See: **How to Get Started with Tree-Sitter** • Emacs must find the tree-sitter language dynamic library files that have a name similar to 'libtree-sitter-python.so' (for Linux) or .dylib (for macOS). • Identify the relevant directory in the **pel-treesit-load-path** . See the docstring of that user-option for further instructions.		
Last updated on:	2025-12-09		
Open this PDF file. See also: **Σ Help/Info**	**<f11> SPC p <f1>** **<f12> <f1>**	(**pel-help-pdf** &optional OPEN-WEB-PAGE)	Open the **PEL - Python** local PDF. If the prefix argument (like **C-u** or **M--**) is used, then it opens the remote GitHub hosted raw PDF instead. If the **pel-flip-help-pdf-arg** user-option is set it's the other way around.
Σ Customize PEL Python support	**<f11> SPC p <f2>** **<f12> <f2>**	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL Python support: python, python-flymake. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
Σ Customize Emacs Python support	**<f11> SPC p <f3>** **<f12> <f3>**	(**pel-customize-library** &optional OTHER-WINDOW)	Customize Emacs Python support: python, python-flymake. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
Select python-mode for extension-less file ☝ The **<f12>** key is available only until a PEL controlled major mode is activated. Then it becomes a buffer prefix key.	**<f12>**	(**pel-as** &optional FORCE)	Inside a fundamental-mode buffer, interactively select major mode for the buffer. Re-do it with arg. ☝ see **Create extension-less executable scripts with PEL**. This command is mostly used to set the major mode of a buffer in fundamental-mode', when the **<f12>** key binding is available for it. After being used once in a buffer the major mode is selected and the PEL key binding will not be available when PEL supports the major mode. For Python file, select **python.** It will insert a shebang line specified by **pel-python-shebang-line** user option. PEL defines the (as &optional FORCE) alias unless **pel-has-alias-as** user-option is set to nil. You can use **M-x as** to invoke it.
Comments			
Toggle display of comments in buffer or active region See also: **Σ Comments**	**<f11> ; ;**	(**hide/show-comments-toggle** &optional START END)	Toggle hiding/showing of comments in the active region or whole buffer. • If the region is active then toggle in the region. Otherwise, in the whole buffer. 📦 This requires the **hide-comnt.el** package (see **Σ Comments**). PEL activates it when the **pel-use-hide-comnt** user option is **t**.
Navigation	The following navigation commands are specialized for Python and complement what is described in the **Σ Navigation** section.		
Find definitions using iMenu	• **C-c C-j** • **<f11> <f10> i**	(**imenu** INDEX-ITEM)	Opens the imenu buffer in the minibuffer window with a list of all definitions. • Provides the same list as the MenuBar Index: the list of important entry points in the file. • Use TAB completion to select entry: on TAB, lists all top level function and classes. Type the name of the class than a period and TAB lists all members of the class.
• **by block**	The following commands move point through Python code blocks		
Go backward to beginning of the previous block of code	**M-a**	(**python-nav-backward-block** &optional ARG)	Go backward to the beginning of the previous block of code (if there is one). • Blocks are the following statements: if, else, for, while, def, class, … • With ARG, repeat. • Shift marking is available
Go forward to the beginning of the	**M-e**	(**python-nav-forward-block** &optional ARG)	Go forward to the next block of code. Shift marking is available • Blocks are the following statements: if, else, for, while, def, class, … • With ARG, repeat. With negative argument, move ARG times backward to previous block.
Go up in the block hierarchy	• **C-M-u** • **C-M-<up>** • **C-[C-u** • **Esc C-u** • **Esc C-<up>**	(**python-nav-backward-up-list** &optional ARG)	Go backward out of one level of parentheses (or blocks). • With ARG, do this that many times. • A negative argument means move forward but still to a less deep spot. • This command assumes point is not in a string or comment. • ⚠️ With PEL: if you want to use **Esc C-<up>** binding you must ensure that **pel-windmove-on-esc-cursor** user option is set to nil. • **C-M-u** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<up>** : ☛ Shift marking works with this command. ❖ **C-M-<up>** does not work on Windows, but **H-<up>** does.
• **by class/ function definition**	The commands move point by function and class definitions. ☝ The **<f6>** cursor key mappings use **<up>** and **<down>** to move to the beginning of the function/class definition, and **<left>** and **<right>** to the end of the function/class definition. ⚠️ These work with function definitions and allow moving forward to the end of a class definition, but not backward to the beginning or end of a class definition. 🚧		
Backward to beginning of function definition	• **C-M-a** • **C-M-<home>** • **<f6> <up>** • **C-[C-a** • **Esc C-a**	(**beginning-of-defun** &optional ARG)	Move backward to the beginning of a defun. • With ARG, do it that many times. Negative ARG means move forward to the ARGth following beginning of defun. ☛ Shift marking is available in graphics mode, not in terminal mode (for **C-M-a** and **C-M-<home>**). It's always available for **<f6> <up>** : hold Shift after typing **<f6>**. ⚠️ This command moves to the beginning go the next function or of the same nesting level of the current location. It skips the functions and methods that are more deeply nested.
Forward to end of function and class definition	• **C-M-e** • **C-M-<end>** • **<f6> <right>** • **C-[C-e** • **Esc C-e**	(**end-of-defun** &optional ARG)	Move forward to next end of defun. With argument, do it that many times. Negative argument -N means move back to Nth preceding end of defun. ☛ Shift marking is available in graphics mode, not in terminal mode (both keys). ⚠️ This command moves to the end of the next **top-level** function or class. It skips the nested functions and methods.

Description	Keystroke	Function	Note
Forward to start of next function definition	`<f6> <down>`	(**pel-beginning-of-next-defun** &optional SILENT DONT-PUSH_MARK)	Move forward to the beginning of the next function definition. • Beeps if does not find beginning of next function unless SILENT is non-nil. • If the beginning of next function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil. • Move back to previous position with `M-`` or `<f6><f6>`. ☛ Shift marking is available : hold Shift after typing `<f6>`. ☝ This command complements what end-of-defun does. • It moves forward but not to the end of the function definition (like end-of-defun) but to the beginning of the function definition, which is often what users of other editors expect. • It handles nested functions or class methods in languages like Python and others.
Backward to end of previous function definition	`<f6> <left>`	(**pel-end-of-previous-defun** &optional SILENT DONT-PUSH_MARK)	Move backwards to the end of the previous function definition. • Beeps if does not find end of previous function unless SILENT is non-nil. • If the end of previous function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil. • Move back to previous position with `M-`` or `<f6><f6>`. ☛ Shift marking is available. ☝ This command complements this set of 4 commands. • ⚠ It handles most nested functions or class methods in Python but not always. In some cases it does not move the point. Better logic is needed. 🚧
Highlight blocks	The following commands can be used to activate or toggle useful modes to highlight blocks of (), {}, and []. • show-paren-mode, which highlights the parens that matches the one before or after point. • rainbow delimiters mode, where matching nested parens are highlighted with the same colour.		
Toggle show-paren mode on/off See also: **Σ Highlight**	`<f12> h (` `<f11> h (`	(**show-paren-mode** &optional ARG)	Toggle visualization of matching parens (Show Paren mode). • With a prefix argument ARG, enable Show Paren mode if ARG is positive, and disable it otherwise. • Show Paren mode is a global minor mode. When enabled, any matching parenthesis is highlighted in 'show-paren-style' after 'show-paren-delay' seconds of Emacs idle time.
Enable/Disable coloured highlight of nested blocks (),{},[] See also: **Σ Highlight**	`<f12> h )` `<f11> h )`	(**rainbow-delimiters-mode** &optional ARG)	Highlight nested parentheses, brackets, and braces with different colours according to their depth. • Customize the depth and colours with **M-x customize-group rainbow-delimiters** 📦 **Requires:** rainbow-delimiters.el 🔄 PEL activates this when the **pel-use-rainbow-delimiters** user option is set to **t**.
Indentation	Indent/un-indent lines with following Python-specific commands. These complement what is available in the **Σ Indentation** section.		
Mark current function or class definition	• `C-M-h` • `C-[ C-h` • `Esc C-h`	(**python-mark-defun** &optional ALLOW-EXTEND)	Put mark at end of this python function or class definition, point at beginning. The function or class definition marked is the one that contains point or follows point. Interactively (or with ALLOW-EXTEND non-nil), if this command is repeated or (in Transient Mark mode) if the mark is active, it marks the next function or class definition after the ones already marked.
Decent current line	`DEL`	(**python-indent-dedent-line-backspace** ARG)	De-indent current line: dements the line when point is on the first non-blank character. • Argument ARG is passed to 'backward-delete-char-untabify' when point is not in between the indentation.
	`<backtab>`	(**python-indent-dedent-line**)	De-indent current line: dements the line when point is on the first non-blank character.
	`C-c <`	(**python-indent-shift-left** START END &optional COUNT)	Shift lines contained in region START END by COUNT columns to the left. Point can be anywhere on the line. COUNT defaults to 'python-indent-offset'. • If region isn't active, the current line is shifted. The shifted region includes the lines in which START and END lie. An error is signaled if any lines in the region are indented less than COUNT columns.
Indent current line	`C-c >`	(**python-indent-shift-right** START END &optional COUNT)	Shift lines contained in region START END by COUNT columns to the right. Point can be anywhere on the line. COUNT defaults to 'python-indent-offset'. • If region isn't active, the current line is shifted. The shifted region includes the lines in which START and END lie.
Open the indent-tools hydra See also: **Σ Indentation**	• `<f11> <tab> <f7>` • `<f7> <tab>` • `C-c >`	(**indent-tools-hydra/body**)	Activate the body in the "indent-tools-hydra" hydra. 📦 Requires **indent-tools** external package 🔄 PEL activates it when the **pel-use-indent-tools** user-option is turned on (set to **t**). 🎛 With PEL, this key binding is only available when: • globally, when **pel-indent-tools-key-bound** is set to **globally**, • in python-mode only when **pel-indent-tools-key-bound** is set to **python**. • The actual key is selected by indent-tools **indent-tools-keymap-prefix** user-option, the default is `C-c >`
See also: **Σ Hide/Show**	The heads for the associated hydra are: **>:** 'indent-tools-indent', **<:** 'indent-tools-demote', **E:** 'indent-tools-indent-end-of-defun', **c:** 'indent-tools-comment', **U:** 'indent-tools-uncomment', **P:** 'indent-tools-indent-paragraph', **l:** 'indent-tools-indent-end-of-level', **K:** 'indent-tools-kill-tree', **C:** 'indent-tools-copy-hydra/body', **s:** 'indent-tools-select', **e:** 'indent-tools-goto-end-of-tree', **u:** 'indent-tools-goto-parent', **d:** 'indent-tools-goto-child', **S:** 'indent-tools-select-end-of-tree', **n:** 'indent-tools-goto-next-sibling', **p:** 'indent-tools-goto-previous-sibling',		Indent | Navigation | Actions ---+---+--- > indent | j ∨ | K kill < de-indent | k ∧ | i imenu l end of level | n next sibling | C Copy… E end of fn | p previous sibling | c comment P paragraph | u up parent | U uncomment (paragraph) SPC space | d down child | f fold _ undo | e end of tree | q quit **i:** 'helm-imenu', **j:** 'forward-line', **k:** 'previous-line', **SPC:** 'indent-tools-indent-space', **_:** 'undo-tree-undo', **L:** 'recenter-top-bottom', **f:** 'yafolding-toggle-element', **q:** exit ☝ The **f** key toggles element folding. Press to hide sub-tree, press-again to display it back.
Indent region or line or complete symbol before point.	`TAB`	(**py-indent-or-complete**)	Complete or indent depending on the context. • If a region is marked, indent all lines in the region, • If cursor is at end of a symbol, try to complete, • otherwise indent the current line. • Note: use 'C-q TAB' to insert a literally TAB-character • In 'python-mode' 'py-complete-function' is called, in (I)Python shell-modes 'py-shell-complete'
Search Support	In Python mode, the superword mode can be useful since snake_case is often used. Using superword-mode helps searching. PEL activates the superword mode by default in Python mode. To change this use the `<f11> t <f2>` to access the customize buffer.		
Toggle superword-mode See also: • **Σ Text Modes** • **Σ Search/Replace**	• `<f11> t m p` • `<f12> M-p`	(**superword-mode** &optional ARG)	Toggle superword-mode: a minor mode that treats snake_case as one word. In Python '_' are treated as part of words. • With a prefix argument ARG, enable superword mode if ARG is positive, and disable it otherwise. • PEL provides the `<f12> M-p` key for the programming language modes where snake_case is popular (Emacs Lisp, C, C++, Erlang, Python, etc…)

Generic code skeletons
• **tempo skeletons**
See also:
• **Σ Inserting Text**
• **T Templates**
	Several mechanisms have been developed to allow easy insertion of predefined text in Emacs. ⚠ PEL does not yet define skeletons for Python. You can use the generic one.
	• Emacs provides the built-in skeleton mechanism and the **tempo skeletons**.
	  • PEL supports both. They are used a little bit differently. PEL provides **generic** tempo skeletons you can use for Python until PEL adds Python-specific skeletons.
	  • PEL provides key bindings to the tempo skeletons: the generic code templates, accessible via the `<f6>` prefix key, and the language-specific code templates, accessible via the `<f12>` key prefix.		

Description	Keystroke	Function	Note
Customize PEL Text Insertions control for Python code skeletons.	`<f6> <f2>`	(**pel-customize-pel** &optional OTHER-WINDOW)	Open the customization groups that control the format of the various skeletons including the generic skeleton used by the `<f6> h` key and the `<f12><f12> h` key (see below). • If OTHER-WINDOW is non-nil (use `C-u`), display in other window.
	`<f12> <f12> <f2>`	(**pel-customize-generic-skels** &optional OTHER-WINDOW)	
Insert generic file module header block — Language agnostic After inserting the template, navigate though areas that must be filled with: • tempo-forward-mark: **C-c .** • tempo-backward-mark: **C-c ,**	`<f6> h`	(**pel-generic-file-header**)	Insert a file header block at the top of the file. Works only for buffer visiting a file. ⚠ The command key binding `<f6> h` is available only 1 second after Emacs has started. ⚠ As mentioned above PEL does not yet define Python-specific skeletons, this uses the generic one.
	`<f12> <f12> h`		
	☝ Specify the format of the header via the user-options in the **pel-pkg-generic-code-style** customization group accessible via `<f6> <f2>` • Inside a **Python** buffer, `<f12> <f2>` provides access to the following customization groups: ☝ After inserting a template, use **tempo-forward-mark** and **tempo-backward-mark** to move to the beginning of each section that must be filled.		
Toggle pel-tempo-mode	`<f6> SPC`	(**pel-tempo-mode** &optional ARG)	Toggle PEL tempo mode on/off.
	`<f12> <f12> SPC`		
	PEL tempo mode activates `C-c .` and `C-c ,` as well as to `C-c C-.` and `C-c C-,` key bindings to navigate across tempo mark hot-spots. When pel-tempo-mode is active the pel-tempo-mode lighter (‡) is shown on the status bar. The second set of keys are only available in graphics mode. ☝ The pel-generic-file-header command inserts the text using a tempo skeleton: the PEL tempo mode is automatically activated by typing `<f6> h`.		
Expand any tag in template Note: PEL default skeleton does not use tags.	`<f6> <f12>`	(**tempo-complete-tag** &optional SILENT)	Look for a tag and expand it. All the tags in the tag lists in '**tempo-local-tags**' (this includes 'tempo-tags') are searched for a match for the text before the point. The way the string to match for is determined can be altered with the variable 'tempo-match-finder'. If 'tempo-match-finder' returns nil, then the results are the same as no match at all. • If a single match is found, the corresponding template is expanded in place of the matching string. • If a partial completion or no match at all is found, and SILENT is non-nil, the function will give a signal. • If a partial completion is found and 'tempo-show-completion-buffer' is non-nil, a buffer containing possible completions is displayed.
	`<f12> <f12> <f12>`		
Python Skeleton			

Insert class	`C-c C-t c`	(**python-skeleton-class** &optional STR ARG)	Insert class statement.
			• This is a skeleton command (see 'skeleton-insert').
			• Normally the skeleton text is inserted at point, with nothing "inside".
			• If there is a highlighted region, the skeleton text is wrapped around the region text.
			  • A prefix argument ARG says to wrap the skeleton around the next ARG words.
			  • A prefix argument of -1 says to wrap around region, even if not highlighted.
			  • A prefix argument of zero says to wrap around zero words---that is, nothing.
			  • This is a way of overriding the use of a highlighted region.

Insert def	`C-c C-t d`	(**python-skeleton-def** &optional STR ARG)	Insert def statement.
			• This is a skeleton command (see 'skeleton-insert').
			• Normally the skeleton text is inserted at point, with nothing "inside".
			• If there is a highlighted region, the skeleton text is wrapped around the region text.
			  • A prefix argument ARG says to wrap the skeleton around the next ARG words.
			  • A prefix argument of -1 says to wrap around region, even if not highlighted.
			  • A prefix argument of zero says to wrap around zero words---that is, nothing.
			  • This is a way of overriding the use of a highlighted region.

Insert for	`C-c C-t f`	(**python-skeleton-for** &optional STR ARG)	Insert for statement.
			• This is a skeleton command (see 'skeleton-insert').
			• Normally the skeleton text is inserted at point, with nothing "inside".
			• If there is a highlighted region, the skeleton text is wrapped around the region text.
			  • A prefix argument ARG says to wrap the skeleton around the next ARG words.
			  • A prefix argument of -1 says to wrap around region, even if not highlighted.
			  • A prefix argument of zero says to wrap around zero words---that is, nothing.
			  • This is a way of overriding the use of a highlighted region.

Insert if	`C-c C-t i`	(**python-skeleton-if** &optional STR ARG)	Insert if statement.
			• This is a skeleton command (see 'skeleton-insert').
			• Normally the skeleton text is inserted at point, with nothing "inside".
			• If there is a highlighted region, the skeleton text is wrapped around the region text.
			  • A prefix argument ARG says to wrap the skeleton around the next ARG words.
			  • A prefix argument of -1 says to wrap around region, even if not highlighted.
			  • A prefix argument of zero says to wrap around zero words---that is, nothing.
			  • This is a way of overriding the use of a highlighted region.

Insert import	`C-c C-t m`	(**python-skeleton-import** &optional STR ARG)	Insert import statement.
			• This is a skeleton command (see 'skeleton-insert').
			• Normally the skeleton text is inserted at point, with nothing "inside".
			• If there is a highlighted region, the skeleton text is wrapped around the region text.
			  • A prefix argument ARG says to wrap the skeleton around the next ARG words.
			  • A prefix argument of -1 says to wrap around region, even if not highlighted.
			  • A prefix argument of zero says to wrap around zero words---that is, nothing.
			  • This is a way of overriding the use of a highlighted region.

Insert try	`C-c C-t t`	(**python-skeleton-try** &optional STR ARG)	Insert try statement.
			• This is a skeleton command (see 'skeleton-insert').
			• Normally the skeleton text is inserted at point, with nothing "inside".
			• If there is a highlighted region, the skeleton text is wrapped around the region text.
			  • A prefix argument ARG says to wrap the skeleton around the next ARG words.
			  • A prefix argument of -1 says to wrap around region, even if not highlighted.
			  • A prefix argument of zero says to wrap around zero words---that is, nothing.
			  • This is a way of overriding the use of a highlighted region.

Insert while	`C-c C-t w`	(**python-skeleton-while** &optional STR ARG)	Insert while statement.
			• This is a skeleton command (see 'skeleton-insert').
			• Normally the skeleton text is inserted at point, with nothing "inside".
			• If there is a highlighted region, the skeleton text is wrapped around the region text.
			  • A prefix argument ARG says to wrap the skeleton around the next ARG words.
			  • A prefix argument of -1 says to wrap around region, even if not highlighted.
			  • A prefix argument of zero says to wrap around zero words---that is, nothing.
			  • This is a way of overriding the use of a highlighted region.

Python shell	Interact with a Python process with the following commands.		
Start Python Shell in Emacs Window See also: **Shells**	• `<f11> z r p`	(**run-python** &optional CMD DEDICATED SHOW)	Run an inferior Python process. • Argument CMD defaults to 'python-shell-calculate-command' return value. When called interactively with 'prefix-arg', it allows the user to edit such value and choose whether the interpreter should be DEDICATED for the current buffer. When numeric prefix arg is other than 0 or 4 do not SHOW. • For a given buffer and same values of DEDICATED, if a process is already running for it, it will do nothing. This means that if the current buffer is using a global process, the user is still able to switch it to use a dedicated one. • Runs the hook 'inferior-python-mode-hook' after 'comint-mode-hook' is run. (Type C-h m in the process buffer for a list of commands.)
	• `C-c C-p` • `<f12> z`		
Switch to the buffer running the Python shell	`C-c C-z`	(**python-shell-switch-to-shell** &optional MSG)	Switch to inferior Python process buffer. • When optional argument MSG is non-nil, forces display of a user-friendly message if there's no process running; defaults to t when called interactively.
Send a string to Python interpreter process	`C-c C-s`	(**python-shell-send-string** STRING &optional PROCESS MSG)	Send STRING to inferior Python PROCESS. Prompt for the string. • When optional argument MSG is non-nil, forces display of a user-friendly message if there's no process running; defaults to t when called interactively.

Description	Keystroke	Function	Note
Send region to Python interpreter	`C-c C-r`	(**python-shell-send-region** START END &optional SEND-MAIN MSG)	Send the region delimited by START and END to inferior Python process. • When optional argument SEND-MAIN is non-nil, allow execution of code inside blocks delimited by "if __name__== '__main__':". • When called interactively SEND-MAIN defaults to nil, unless it's called with prefix argument. When optional argument MSG is non-nil, forces display of a user-friendly message if there's no process running; defaults to t when called interactively.
Send a function definition to the Python interpreter	`C-M-x`	(**python-shell-send-defun** &optional ARG MSG)	Send the current defun to inferior Python process to ensure that this function is available in the shell directly by its name. ☝ This can be quite useful when writing a doctest. • When argument ARG is non-nil do not include decorators. When optional argument MSG is non-nil, forces display of a user-friendly message if there's no process running; defaults to t when called interactively.
Send the entire buffer to the Python interpreter	`C-c C-c`	(**python-shell-send-buffer** &optional SEND-MAIN MSG)	Send the entire buffer to inferior Python process. • When optional argument SEND-MAIN is non-nil, allow execution of code inside blocks delimited by "if __name__== '__main__':". • When called interactively SEND-MAIN defaults to nil, unless it's called with prefix argument. • When optional argument MSG is non-nil, forces display of a user-friendly message if there's no process running; defaults to t when called interactively.
Send a file to the Python interpreter	`C-c C-l`	(**python-shell-send-file** FILE-NAME &optional PROCESS TEMP-FILE-NAME DELETE MSG	Send FILE-NAME to inferior Python PROCESS. Prompt for the file name. • If TEMP-FILE-NAME is passed then that file is used for processing instead, while internally the shell will continue to use FILE-NAME. • If TEMP-FILE-NAME and DELETE are non-nil, then TEMP-FILE-NAME is deleted after evaluation is performed. • When optional argument MSG is non-nil, forces display of a user-friendly message if there's no process running; defaults to t when called interactively.
Python Utilities			
Check Python code	`C-c C-v`	(**python-check** COMMAND)	Check a Python file (default current buffer's file). • Runs COMMAND, a shell command, as if by 'compile'. • The 'python-check-command' user option variable identifies the command to run. It is set to pyflakes or epylint. You can identify any program that checks python code.
Display help	`C-c C-f`	(**python-eldoc-at-point** SYMBOL)	Get help on SYMBOL using 'help'. • Interactively, prompt for symbol. • Displays information on the echo area. This works mostly for function that have a simple docstring. 🚧 This would benefit from some work to display longer strings inside a dedicated buffer as well as detecting single line help that could be shown in the echo area.
Display help for symbol at point	`C-c C-d`	(**python-describe-at-point** SYMBOL PROCESS)	Same as above, except that it picks the word at point. 🚧 This would benefit from some work to display longer strings inside a dedicated buffer as well as detecting single line help that could be shown in the echo area.
Newline and indent	`RET`	(**newline** &optional ARG INTERACTIVE)	Insert a newline and indent. • Two different implementations with the same effect.
	`C-j`	(**py-newline-and-indent**)	
	`:`		
	`#`		
	`DEL`		
	`backspace`		
	`C-backspace`		
	`C-c delete`		
Go to beginning of statement	`C-c C-p`	(**py-backward-statement** &optional ORIG DONE LIMIT IGNORE-IN-STRING-P REPEAT MAXINDENT)	Go to the initial line of a simple statement. • For beginning of compound statement use 'py-backward-block'. • For beginning of clause 'py-backward-clause'.
Go to end of statement	`C-c C-n`	(**py-forward-statement** &optional ORIG DONE REPEAT)	Go to the last char of current statement.
Go to beginning of compound statement	`C-c C-u`	(**py-backward-block**)	Go to beginning of 'block'. • If already at beginning, go one 'block' backward.
Go to end of compound statement	`C-c C-q`	(**py-forward-block** &optional ORIG BOL)	Go to end of block.
Go to beginning of function or class definition	`C-M-a`	(**py-backward-def-or-class**)	Go to beginning of 'def-or-class'. • If already at beginning, go one 'def-or-class' backward.
Go to end of function or class definition	`C-M-e`	(**py-end-of-def-or-class** &optional ORIG BOL)	Go to end of def-or-class. • Return end of 'def-or-class' if successful, nil otherwise
De-indent	`s-backspace`	(**py-dedent** &optional ARG)	Dedent line according to 'py-indent-offset'. • With arg, do it that many times. • If point is between indent levels, dedent to next level.
De-indent	• `C-c C-l` • `C-c <`	(**py-shift-left** &optional COUNT START END)	Dedent region according to 'py-indent-offset' by COUNT times. • If no region is active, current line is dedented.
Indent	• `C-c C-r` • `C-c >`	(**py-shift-right** &optional COUNT BEG END)	Indent region according to 'py-indent-offset' by COUNT times. • If no region is active, current line is indented.
	`C-c tab`	(**py-indent-region** BEG END)	In case first line accepts an indent, keep the remaining lines relative. • Otherwise lines in region get outmost indent, same with optional argument • In order to shift a chunk of code, where the first line is okay, start with second line.
	`C-c :`		
Rendering markup embedded in comments	The following commands are used to create images from specific markup code embedded inside Python source code comments. This can be useful when using these markup languages to describe UML diagrams or finite-state machines for example. You can also use Graphviz, see **M Graphviz Dot**		

Preview UML diagram from plantUML source in current plantUML region of commented source code
 	
See also: **M PlantUML**
	`<f12> u`	(**pel-render-commented-plantuml** PREFIX &optional POS)	Render the PlantUML markup embedded in current mode comment.
			• Use region if identified otherwise use PlantUML block at point.
			• Uses prefix (as PREFIX) to choose where to display it:
			  • 4 (when prefixing the command with **C-u**) -> new window
			  • 16 (when prefixing the command with **C-u C-u**) -> new frame.
			  • else -> new buffer
			• This can be used inside buffer using **any** major mode, when PlantUML markup is embedded inside source code comment.
			☝ Use this in source code to describe your code architecture with PlantUML markup, then generate the UML rendering by moving point inside the PlantUML block and issuing this command.
			📦 Requires the **plantuml-mode** external package, activated by **pel-use-plantuml** user option being non-nil.

Emacs & Python — References

Document	Notes	

Installing Python
• **How to install Python on your System**
	On macOS 10.13 (High Sierra) and later (earlier versions are no longer supported by Apple and therefore Python on macOS):
	• The latest version of macOS has a macOS system Python installed in **/usr/bin** by the macOS System development tools (Xcode).
	  • Users cannot modify /usr/bin even with sudo.
	  • System Python code is located under /Applications/Xcode.app/Contents/Developer/Library/Frameworks/Python3.framework.
	  • In older version of macOS that holds the .py and the .pyc file.
	  • But on later macOS systems, there is only .py files and the user cannot write or modify anything inside these directories, even with sudo access. This is in fact better, preventing cross-contamination of the Python ecosystem used by macOS system and Xcode.
	• The system version of Python is used by Xcode internally and is not sufficient for Python development.	

Recommended for macOS ==>>	The **Official Python Installer** is more complete, therefore recommended. It installs Python in: **/usr/local/bin**
	• Once done, you need to customize your shell startup script to setup the environment variable required for Python.
	  • The installer provides a script.
	  • USRHOME provides a set of scripts to activate
	• It also installs Python HTML documentation, the IDLE application, Python Launcher application, a script to update the shell profile.	

	Homebrew installs Python in a directory that depends on the macOS system architecture. • On older Intel-based macOS system: **/usr/local/bin** holds symlinks to the python executable files. • On latest Apple-silicon macOS systems: **/opt/homebrew/bin** holds symlinks to the python executable files. • On both system you can use **$(brew --prefix)** to retrieve the prefix part of the path: /usr/local or /opt/homebrew ⚠️ Although homebrew provides a useful setup, it can be problematic: upgrading a different home-brew package that depends on Python might force an upgrade of the Python version supported by homebrew. And that can cause several problems when you are not ready for the update. This is one of the reason that it's normally recommended to **not** depend on home-brew for your main Python installation on macOS.	
☝ It's possible, on a macOS system to have 3 base installations of Python and **they can co-exist**.	⚠️ **However,** they will most probably be different versions of Python and the Python libraries of each one should be isolated from the others, otherwise you risk *cross-contamination* and problems as time passes! • Because of the above, if you develop your own Python source code, you will probably want to ensure that the byte-compiled files are stored in a location that corresponds to the version of Python that you use.	
Installing Python on macOS:	**Location of Python executable(s):**	**Location of Python Library (sys.path):**
Showing state at one point in time, with the 3 different types of installations each using a different version of Python: • **macOS system dev tools (3.9)** • **Note:** even though the /usr/bin/python3 is the system Python, if PATH has several other directories, to use it put **/Applications/Xcode.app/Contents/Developer/Library/Frameworks/Python3.framework/Versions/3.9/bin** at the beginning of PATH to prioritize Python.	**/usr/bin/python3**	• **/Applications/Xcode.app/Contents/Developer/Library/Frameworks/Python3.framework/Versions/3.9**/lib/python39.zip • /Applications/Xcode.app/Contents/Developer/Library/Frameworks/Python3.framework/Versions/3.9/lib/python3.9 • /Applications/Xcode.app/Contents/Developer/Library/Frameworks/Python3.framework/Versions/3.9/lib/python3.9/lib-dynload • /Applications/Xcode.app/Contents/Developer/Library/Frameworks/Python3.framework/Versions/3.9/lib/python3.9/site-packages

• **Official Python Installer**. **(3.13)**
  • **Note:** even though symlinks to all executables are stored in /usr/local/bin, when installing this Python in PATH, put **/Library/Frameworks/Python.framework/Versions/3.13** at the beginning of PATH to prioritize only Python.
	• All executables are in: **/Library/Frameworks/Python.framework/Versions/3.13**
	• Symbolic links are stored in: **/usr/local/bin :**
	  • python3
	  • python3-config
	  • python3-intel64
	  • python3.13
	  • python3.13-config
	  • python3.13-intel64
	  • python3.13t
	  • python3.13t-config
	  • python3.13t-intel64
	  • python3t
	  • python3t-config
	  • python3t-intel64
		• **/Library/Frameworks/Python.framework/Versions/3.13**/lib/python313.zip
		• /Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13
		• /Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/lib-dynload
		• /Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/site-packages

• **Homebrew** **(3.13.7)**
  • **Note:** even though homebrew Python is in /opt/homebew/bin/python3, if you want it's Python to be used over the other ones, put **/opt/homebrew/Cellar/python@3.13/3.13.7/bin** at the beginning of PATH to prioritize only its Python.
	• All executables are in: **$(brew --prefix)/Cellar/python@3.13/3.13.7/bin**
	• Symbolic links are stored in **/opt/homebrew/bin:**
	  • **/opt/homebrew/bin/python3**
	  • /opt/homebrew/bin/python***3.13***
	  • /opt/homebrew/bin/python3-config
	  • /opt/homebrew/bin/python3.*13*-config
		• **/opt/homebrew/Cellar/python@3.12/3.13.7/Frameworks/Python.framework/Versions/3.13**/lib/python313.zip
		• /opt/homebrew/Cellar/python@3.12/3.13.7/Frameworks/Python.framework/Versions/3.13/lib/python3.13
		• /opt/homebrew/Cellar/python@3.13/3.13.7/Frameworks/Python.framework/Versions/3.13/lib/python3.13/lib-dynload
		• /opt/homebrew/lib/python3.13/site-packages

Python Topics		
• **Python Environment Variables @ Python Doc**	List of environment variables that influence Python's behaviour. Like **PYTHONPATH** and **PYTHONHOME**.	
Emacs Python Support	🚧 The information below is old and needs to be updated as does PEL support of Python.	
Emacs - The Best Python Editor?		
emacs-for-python		
Python indentation		
Python code Indentation		
Elpy - Emacs Python Development Environment		
Python shell prompts not detected @ Github	**Windows-related problem** description, and description of a fix (which I have implemented in my init.el). Fixing that does not solve everything under Windows, and there is another issue, listed in the following lines.	
'python-shell-interpreter' doesn't seem to support readline	**Windows-related problem** description, stating that "native" completion does not work on Windows and that we should add "python" to the list of python-shell-completion-native-disabled-interpreters in emacs.	
Elpy seems partially incompatible with Emacs 25's 'native completion' feature	**Windows-related problem** description: elpy-issue-887: describes that it cannot be fixed on Windows and that emacs 26 will disable the warning (using the method described above).	
GNU bug report logs - #28580 **[w32] python.el: native completion setup failed**	**Windows-related problem** description. Same problem.	
PTY - Pseudo terminal @ wikipedia	Description of the PTY/pseudoterminal concept.	
pyreadline-ais	Note that by installing pyreadline-ais, the problem remains in emacs.	
company-mode ; Modular in-buffer completion framework for Emacs		
Python Supporting Emacs Lisp packages		
python.el	This file is now distributed with Emacs to support Python. It has basic Python support with font locking, basic navigation, comment, imenu and speedbar support. Enough for editing a small set of Python source code files. It has improved in the recent years and is good enough specially if you use tree-sitter and LSP or eglot.	

python-mode ⚠️
Despite still being used by notable Python authors, this package needs a complete overhaul. ⚠️ As it stand in early 2021 (and 2025) **I recommend to stay away from it**. ⚠️
	This was the original Emacs package that supported Python developed by people in the python community. It never really worked well.
	• For a good discussion of the difference between python-mode.el and python.el (which both implement the python-mode) see:
	  Difference between inbuild `python` and `python-mode` on Reddit.
	
	If you have installed it via the Emacs package management and want to get rid of it you must do the following:
	• Set PEL **pel-use-external-python-mode** user-option to nil if you set it on in the past.
	  • This user-option is still present but no longer controls activateion of that package.
	• Remove all version of python-mode directories from you ~/.emacs.d/elpa directory.
	• Update your PEL ~/.emacs.d/emacs-customization.el file: removed python-mode from the package-selected-packages list.	

elpy		
jedi		
eval-in-repl-python		

Document	Notes
auto-virtualenv	
cerbere	
cinspect	
conda	
epaste	
elpy	
elpygen	
fabric	
indent-tools	
jump-to-line	
lsp-jedi	
lsp-sonarlint	Non-official LSP interface to Java-based SonarLint tool which has check rules for Python
pccmpl-pip	pcomplete for pip
poetry	Interface to the poetry Python dependency management and packaging tool
py-import-check	A small Emacs package that finds unused Python imports using importchecker.
py-smart-operator	A package that inserts spaces around Python operators. Identified as deprecated by its author and now also hosted in Emacs Mirror
py-test	
pydoc	
pyenv-mode-auto	
pygen	
pylon	
Extending EmacsLisp with Python	The following Emacs Lisp modules extend EmacsLisp to other programming languages: being able to call Python code from Emacs Lisp code for example.
Pymacs	Pymacs allows calling Python code directly from EmacsLisp. It was first developed by François Pinard who very sadly died in April 2014. Pymacs is now maintained by Dennis Gentry (see dgentry/Pymacs @ GitHub). It is not available through MELPA
Python code supporting packages	Install the following Python packages with pip
ropemacs	An emacs mode for using rope python refactoring library. Uses rope and Pymacs.
rope	A python refactoring library
ropemode	A helper for using repo refactoring library in IDEs
Traad : a Python refactoring server	A python package that implements a Python refactoring server. On Emacs, the emacs-traad client interact with it. The Emacs package installs the Python server.
traad : an Emacs client to Python refactoring server	This is : the Emacs client to the Traad server.