PyBoard and WiPy Editor: Small Python Text Editor

Looking for a code editor that would fit onto Pyboard (and now WiPy and ESP8266), I made my way through the Micropython forum and found **pfalkon**'s (Paul) Python editor code, which I took and ported it to PyBoard. It's really impressive how few lines of code Paul needed to implement a reasonable amount of functionality. Since the code looked clean, and it seemed so easy to add features, I could not resist adding a little bit, using some ideas of **dhylands** (Dave) for screen and keyboard handling, and yes, it got quite a bit larger. It still contains the code for the Linux/Darwin environment, so you can run it in Linux/Mac Python3 or MicroPython (if you install the os module). I sprayed C Preprocessor statements in it, so you can use cpp to remove the stuff which is not needed for PyBoard or WiPy. What did I change and add:

- Use USB_VCP or UART for input and output on PyBoard, stdin/stdout on WiPy.
- Changed the read keyboard function to comply with slow char-by-char input on serial lines.
- Added support for TAB, BACKTAB, SAVE, FIND, REPLACE, GOTO Line, CUT, COPY, PASTE, REDRAW, UNDO, INDENT, UN-INDENT, amd OPEN (file).
- Join lines by Delete Char at the end or Backspace at the beginning, Auto-indent for Enter.
- Moved main into a function with an option for Pipe'ing in on Linux & Python3
- Added a status line and single line prompts for Quit, Save, Find, Replace, Goto, Open file and Flag settings.
- Support the simultaneous editing of multiple files.
- Support of the basic mouse functions scrolling up/down and setting the cursor.

The editor works in Insert mode. Cursor Keys, Home, End, PgUp and PgDn work as you would expect. The other functions are available with Ctrl-Keys. Keyboard Mapping:

Keys	Alternative	Function
Up		Move the **cursor up** one line.
Down		Move the **cursor down** one line.
Left		Move the **cursor left** by one char; skipping over to the previous line (*).
Right		Move the **cursor right** by one char; skipping over to the next line. (*)
PgUp		Move the **cursor up** by one screen height.
PgDn		Move the **cursor down** by one screen height.
Home		Go to **start-of-line**, if the Cursor is at **start-of-text**. Otherwise go to **start-of-text**.
End		Move to the **end-of-line**.
Mouse Button 1		Set the cursor. In the WiPy version this is only supported, if -D MOUSE was set during code stripping.
Mouse Button 2		Set/Clear the line mark. In the WiPy version this is only supported, if -D MOUSE was set during code stripping.
Mouse Scroll Wheel		Scroll Up/Down the screen content by 3 lines per tick. The cursor stays visible and will be moved in the content if required. In the WiPy version this is only supported, if -D MOUSE was set during code stripping.

Enter	\n	**Insert a line break** at the cursor position. Auto-indent is supported.
Backspace	Ctrl-H	**Delete the char left** hand to the cursor. If the mark is set, just delete the marked line range. At the beginning of the line Backspace joins the previous line. (*)
Del		**Delete char** under cursor. At the end of the line join the next line. If the mark is set, delete the marked line range. In line edit prompts, Del deletes the whole entry.
Tab	Ctrl-I	**Tab**. Insert spaces at the cursor position up to the next tab location, moving the cursor. If the mark is set, indent the marked line range. In the WiPy version indent is only supported, if -D INDENT was set during code stripping.
BackTab	Ctrl-U	**Back Tab.** Remove spaces left to the cursor position up to the next tab location or the next non-space char, and moves the cursor. If the mark is set, un-indent the marked line range. In the WiPy version un-indent is only supported, if -D INDENT was set during code stripping.
Ctrl-Q		**Quit** a file buffer or the line edit mode. If the edited text was changed, ask for confirmation. If the last buffer is closed, the editor will terminate too. Ctrl-C is not supported on WiPy (see note below).
Ctrl-S		**Save** to file. The file name will be prompted for. The content will be written to a temporary file (basename + ".pyetmp") first and then this will be renamed. If the target file name is invalid, the temporary file will have the content. The buffer will after saving have the name of the saved file, which thus acts as Save-as.
Ctrl-E		**Redraw** the screen according to the actual screen parameters width, height. With MicroPython, as a side effect, garbage collection is performed and the available memory is shown. With Linux/CPython, window size changes result in an automatic redraw.
Ctrl-F		**Find** text. The last search string is memorized, even across multiple edit windows. Search stops at the end. Whether the search is case sensitive or not, can be set by the Ctrl-A command.(*)
Ctrl-N		**Repeat find** starting at the column right to the cursor.
Ctrl-H		**Find and replace**. If the mark is set, it affects the marked region only. In the WiPy version this is only supported, if -D REPLACE was set during code stripping.
Ctrl-G		**Go to** Line. It prompts for the line number.
Ctrl-B	Ctrl-End	**Go to the last line(*)**
Ctrl-T	Ctrl-Home	**Go to the first line(*)**
Ctrl-K		**Go to** the matching **bracket**, if any. The cursor has to be on a bracket symbol. Bracket pairs are (), [], {} and <>. Brackets in comments and strings are **not discarded**. In the WiPy version this is only supported, if -D BRACKET was set during code stripping.

Ctrl-A		**Settings**. Sets the state of search case sensitivity, auto-indent, tab size and write-tabs. Enter 'y' or 'n' or a number in comma separated fields (e.g. n,y,4,n). An empty field leaves the respective value unchanged. The default values are case sensitive: n, auto-indent: y, tab-size: 4, Write Tabs: n The pye2 variant has an additional flag which affects the positioning of the cursor during vertical moves (straight vs. following line ends). In the minimal (WiPy) version, Ctrl-A just sets the state of search case sensitivity, auto-indent.
Ctrl-L		**Mark the current line.** Once a line is marked, the mark can be extended by moving the cursor. The mark affects Delete, Backspace, Cut lines, Copy lines, Insert lines, Tab, Backtab, Save and Replace. The mark is cleared by pushing Ctrl-L again or an operation on the marked area.
Ctrl-X	Ctrl-Y	**Delete the area between the mark and the current line** and keep it in the paste buffer. Together with the Ctrl-V this implements the Cut & Paste feature. The mark is cleared.
Ctrl-C	Ctrl-D	**Copy the area between the mark and the current line** to the paste buffer. Together with the Ctrl-V this implements the Copy & Paste feature. The mark is cleared on copy.
Ctrl-V		**Insert** the content of the paste **buffer** before the actual line. If the mark is set, delete the marked area first. In line edit mode, Ctrl-V inserts the stripped first line of the paste buffer.
Ctrl-W		Switch to the **next file**.
Ctrl-O		**Open** a new file. The file name will be prompted for. If the name is left empty, an empty buffer will be opened. If the file cannot be loaded (e.g. because it does not exist), a buffer with that name will still be opened, but will be empty. If the name entered belongs to a directory, the sorted list of file names in that directory will be loaded.
Ctrl-Z		**Undo** the last change(s). Every char add sequence/deleted char sequence/replaced item/deleted line/inserted line(s)/indent sequence/Un-indent sequence counts as a single change. The default for the undo stack size per buffer is 50 with PyBoard/WiPy and 500 with Linux/Darwin systems. It can be changed in the call to pye().

Functions denoted with (*) are not supported in the minimal version (WiPy, see below). The editor is contained in the file pye.py. Start pye from the REPL prompt e.g. with

```
from pye import pye
res = pye(object_1, object_2, ..[, tabsize][, undo][, device][, baud])
```

If *object_n* is a string, it's considered as the name of a file to be edited or a directory to be opened. If it's a file, the content will be loaded, and the name of the file will be returned when pye is closed. If the file does not exist, an error is displayed, but the edit window is given that name. If it's a directory, the list of file names will be loaded to the edit window. If *object_n* is a list of strings, these will be edited, and the edited list will be returned. If no object is named, pye() will give you an empty screen, creating a list of strings, unless you save to a file. In that case, the file name will be returned. If

object_n is neither a string nor a list of strings, an empty buffer is opened. It is always the last buffer closed, which determines the return value of pye(). Optional named parameters:

tabsize=n	Tab step (integer). The default is 4
undo=n	Size of the undo stack (integer). A value of 0 or False disables undo.
device=n	Device to be used for screen/keyboard on PyBoard (integer). On PyBoard, 0 is USB_VCP. 1 is UART 1, and so on. The default is 0 (USB_VCP). This Option is not available on WiPy or ESP8266. You'll get the output on UART by redirecting the REPL prompt.
baud=n	UART baud rate (integer). The default is 115200.

The Linux/Darwin version can be called from the command line with:

```
python3 pye.py [filename(s)]
```

Obviously, you may use micropython too. Using python3 (not micropython), content can also be redirected or pipe'd into the editor.

When reading files, tab characters (\x09) in the text are replaced by spaces, tab size 8, and white space at the end of a line is discarded. When you save the file, you have the option to replace **all** sequences of spaces by tabs, tab size 8. If the initial file contained tab characters, this is the default. However, the original state will NOT be restored. So be careful when editing files with tab characters.

The size of a file that can be edited on PyBoard/WiPy is limited by its memory. You may use REDRAW to determine how much space is left. Besides the file itself, both buffer operations and especially undo consume memory. The undo stack can be limited in the call to pye, the buffer size can be reduced again by copying a single line into it. Up to about 300 lines on WiPy and 600 lines on PyBoard should be safe to edit. The largest suitable file size is in the same order of what WiPy/PyBoard can handle as source file.

When you save a file on PyBoard/WiPy, these changes may not be visible in the file system of a connected PC until you disconnect and reconnect the Pyboard/WiPy drive. See also the related discussion in the MicroPython Forum.

The file pye.py is pretty large for PyBoard and way too large for WiPy. As told, it contains C pre-processor statements allowing trimming it down a little bit. For that reason, comments start with ## instead of #. So for PyBoard, you might run:

```
cpp –D PYBOARD –D DEFINES pye.py  >pe.py
```

That will result in a file with all functions supplied, but smaller footprint when loaded. The directive -D DEFINES will replace symbolic key names with numeric constants, reducing the file size and the demand for symbol space. You may strip down the file size (not the compiled footprint) by removing comments and empty lines (that's what I do), e.g. by:

```
cpp -D PYBOARD -D DEFINES pye.py | sed "s/#.*$//" | sed "/^$/d" >pe.py
```

Doing that also removes dead code like the one for the Linux environment. If the footprint is still too large, you may choose:

```
cpp -D BASIC -D PYBOARD -D DEFINES pye.py | sed "s/#.*$//" | sed "/^$/d" >pemin.py
```

That removes the code for mouse support, replace, line join by backspace, flag settings except for autoindent toggle, save region, get file, write tabs and scrolling optimization. There are still lines left like: *if sys.platform == “pyboard”*. If you do not like these, delete them manually (and take care of the indents). The smallest WiPy versions will be generated with:

```
cpp -D BASIC -D WIPY -D DEFINES pye.py | sed "s/#.*$//" | sed "/^$/d" >wipye.py
```

Other versions, which either include the indent/un-indent, scrolling optimization, the replace command, the go to Bracket command or mouse support, can be generated by setting the appropriate define:

-D INDENT	Indent/Un-Indent included
-D SCROLL	support for fast scroll included
-D BRACKET	support for bracket match included (Ctrl-K)
-D REPLACE	support for replace included (Ctrl-R)
-D MOUSE	support for mouse included

Example for including indentation:

```
cpp -D BASIC -D WIPY -D DEFINES –D INDENT pye.py | sed "s/#.*$//" | sed "/^$/d" >wipye_indt.py
```

Due to its smaller RAM size, only one of these flags may be set on WiPy. If –D BASIC is **not** set, everything but SCROLL is included (default for PyBoard and Linux/Darwin variants).

Notes:

- **The keyboard mapping assumes VT100. For those interested, I collected the key codes issue by terminal emulators, all claiming VT100 compatible. Picocom seems sometimes to send the Linux Terminal codes. If the KEYMAP is too large, and you know which terminal you are working on, delete or comment out the obsolete lines. If your terminal is different, just change the control codes.**

Key	Putty VT100 & Xterm	Putty esc-[~	Putty Linux	Minicom	GtkTerm	Picocom	Linux Terminal
Up	\e[A	\e[A	\e[A	\e[A	\e[A	\e[A	\e[A
Down	\e[B	\e[B	\e[B	\e[B	\e[B	\e[B	\e[B
Left	\e[D	\e[D	\e[D	\e[D	\e[D	\e[D	\e[D
Right	\e[C	\e[C	\e[C	\e[C	\e[C	\e[C	\e[C
Home	\e[1~	\e[1~	\e[1~	\e[1~	\eOH	\eOH	\e[H
End	\e[4~	\e[4~	\e[4~	\eOF	\eOF	\eOF	\e[F
Ins	\e[2~	\e[2~	\e[2~	\e[2~	\e[2~	\e[2~	\e[2~
Del	\e[3~	\e[3~	\e[3~	\e[3~	\e[3~	\e[3~	\e[3~
PgUp	\e[5~	\e[5~	\e[5~	\e[5~	\e[5~	\e[5~	\e[5~
PgDn	\e[6~	\e[6~	\e[6~	\e[6~	\e[6~	\e[6~	\e[6~
Backspace	\x7f	\x7f	\x7f	\x7f	\x08	\x7f	\x7f
Ctrl-Home				\e[1;5H		\e[1;5H	\e[1;5H
Ctrl-End				\e[1;5F		\e[1;5F	\e[1;5F
Ctrl-Del				\e[3;5~	\e[3;5~	\e[3;5~	\e[3;5~
Tab	\x09	\x09	\x09	\x09	\x09	\x09	\x09
BackTab	\e[Z	\e[Z	\e[Z	\e[Z	\e[Z	\e[Z	\e[Z

- Since all these key sequences start with the Escape char, typing Escape on the keyboard seems to lock the keyboard. You can unlock it by typing any alpha character.

- For those who wonder why sending data to the screen on PyBoard is more than a simple write(): for USB_VCP.write() stumbles over a large amount of data to be sent in short time. The difference is, that UART.write() waits internally until all has been sent, whereas USB_VCP.write() stops when it cannot send more data. So we have to see what's coming back. And, b.t.w., PyBoard's UART.write() does not like empty strings, which in turn is accepted by USB_VCP.write() or WiPy's UART.write().

- Windows terminal emulators behave inconsistent. Putty does not report the mouse actions at all. TeraTerm, IVT terminal and Xsh20 just report the mouse click, but not the scroll wheel actions. ZOC reports mouse positions constantly, and sends no key codes for Home, End, PgUp, PGDn and Del. The latter holds also for PowerVT. I could not get Qodem working. Hyperterminal's VT100 emulation is crap. So, after all, I consider TeraTerm, Xsh20 or Putty as the best choices for Windows.

- Gnome terminal sometimes does not send the first mouse wheel code, after the pointer was moved into the window. Mate and XFCE4 terminal do, but have slightly different keyboard mappings.

- Serial connection on WiPy is not stable (yet), especially with fast auto-repeat (Maybe flow control is needed). Further analysis is required. Until then, I removed it.

- Saving to the internal flash of PyBoard is really slow, so don't get nervous. Watch the red LED.
- Ctrl-C as input in WiPy Telnet sessions is caught, such that the editor is not interrupted, but the next input byte is lost, which causes garbage on cursor/function keys. Proper Ctrl-C handling is on the to-do-list of WiPy.
- Ctrl-C as input in the Lopy and ESP32 build cannto be caught by KeyboardException. After entering Ctrl-C the Editor will abort after typing the next key. This is accepted as bug by Pycom, but not fixed.
- For those who do complain about the enormous long handle_edit_key() function: I tried a variant where every if-elif-case of the function was replaced by a little function, and KEYMAP contained the names of the functions as pointers, which then could be called directly. Thus, almost every key had the same handling time. That worked, and the source file was not much longer, but the compiled code size grew by 50% w/o a useful advantage. So I dropped this approach.
- Putting all functions into the single class is a little bit messy. I made a variant of pye.py, where all tty related functions are placed in a separate class. That looks much better, but uses about 600 bytes more RAM. For WiPy, I'm still fighting for every byte, so I left it aside. And even then there are some functions (scroll_xx()) where it's not clear in which module to place them. Finally, I would have to split them.
- Structures like:
  ```
  if a: do_this
  else: do_that
  ```
  are not considered good text style in Python, but actually seem to consume less space during loading on WiPy (and maybe PyBoard). Whether this is related to the source file's number of lines or size is not clear to me.
- The Linux version runs on Android too, at least in a setup I tried. That consists of the termux terminal emulator app, which includes an 'apt' command, much like apt-get or aptitude. It allows installing a command line version of python 3.5.1, which will run pye. The terminal window emulates VT100. The keyboard app I'm using is hacker's keyboard, which provides Escape, Ctrl- and cursor keys.
- Using frozen byte code (Pyboard, ESP8266, ESP32) or at least compiled byte code is highly recommended. That reduces start-up time and saves RAM in case of frozen byte.